Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués

Nicolas Moro

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Nicolas MORO

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués

Soutenue à Paris le 13 novembre 2014

M.	Pascal BERTHOMÉ	Rapporteur	INSA Centre Val de Loire, Bourges
M.	Jean-Louis LANET	Rapporteur	INRIA, Rennes
M.	Jean-Claude BAJARD	Examinateur	UPMC, Paris
M.	Christophe GIRAUD	Examinateur	Oberthur Technologies, Bordeaux
M.	Sylvain GUILLEY	Examinateur	Telecom ParisTech, Paris
M.	Jean-François LALANDE	Examinateur	INSA Centre Val de Loire, Bourges
M.	Pascal PAILLIER	Invité	CryptoExperts, Paris
Mme	Emmanuelle ENCRENAZ	Directrice	UPMC, Paris
M.	Bruno ROBISSON	Co-directeur	CEA, Gardanne
Mme	Karine HEYDEMANN	Encadrante	UPMC, Paris

Remerciements

Le lecteur averti aura noté un nombre anormalement élevé de logos en première page, et pourrait se demander pourquoi tant d'encre couleur a été utilisée. Si ce même lecteur effectue une recherche sur Internet et tombe sur mon CV, il risque de se poser quelques questions... une thèse CEA à Gardanne, dans une équipe d'abord rattachée au LETI de Grenoble, puis ensuite rattachée à la division CEA-Tech PACA de Cadarache, tout en étant inscrit comme étudiant de l'Université Pierre et Marie Curie à Paris et pour au final travailler à l'École des Mines de Saint-Étienne... mais pas à Saint-Étienne. La quintessence de la complexité administrative à la française ! Mais concrètement, j'ai passé le plus clair de mon temps à Gardanne et ce manuscrit de thèse représente l'aboutissement de 6 années passées (dont 3 comme élève-ingénieur) sur le site Georges Charpak de l'École des Mines de Saint-Étienne, à Gardanne.

Après cet avant-propos nécessaire pour clarifier un peu ma situation, je souhaiterais tout d'abord remercier Pascal Berthomé et Jean-Louis Lanet pour avoir accepté d'être rapporteurs pour ce travail de thèse. Un grand merci également à Jean-Claude Bajard, Christophe Giraud, Sylvain Guilley, Jean-François Lalande et Pascal Paillier pour avoir accepté de faire partie de mon jury de thèse. J'adresse également mes remerciements à Jean-Luc Danger et Franck Wajsbürt, qui avaient bien généreusement accepté d'être les membres de mon jury d'évaluation de mi-thèse.

Ensuite, je souhaiterais bien évidemment remercier tous ceux qui, par leur soutien, ont contribué à la réussite de cette thèse. En premier lieu, je pense en particulier à Bruno Robisson, qui m'a accompagné au quotidien pour mes travaux pendant ces 3 années, ainsi qu'Emmanuelle Encrenaz qui, malgré la distance, a toujours assuré un excellent suivi de mes travaux et m'a beaucoup aidé notamment pour tout ce qui concerne la vérification formelle. Je les remercie tous deux pour la confiance qu'ils m'ont accordée et pour avoir dirigé mes travaux pendant ces trois années. Un remerciement spécial va à Karine Heydemann, qui a également encadré cette thèse pendant les deux dernières années, avec qui j'ai beaucoup apprécié travailler et pour qui l'art de la compilation ou la transformation de code assembleur n'ont plus de secrets, en plus d'être au moins aussi perfectionniste que moi dans la rédaction d'articles. Je souhaite par ailleurs remercier également Bruno, Emmanuelle et Karine pour leurs relectures attentives des différentes versions de ce manuscrit et sans

qui celui-ci serait de bien moindre qualité. Un autre remerciement très spécial va à Amine Dehbaoui, avec qui j'ai également beaucoup apprécié travailler et qui a été pendant près de deux ans mon co-encadrant *de facto*, grand virtuose du banc d'injection électromagnétique qui connaît à peu près tout ce qui va du silicium au buffer overflow.

Je remercie également tous ceux qui ont partagé mon quotidien pendant ces années, qui ont partagé une bière au Tibbar avec moi ou m'ont écouté leur parler longuement de ma vision de l'organisation de l'école ou encore des plans machiavéliques de l'association des anciens. Ce sont d'autres thésards, d'anciens professeurs, des stagiaires, et plus généralement des personnes qui ont contribué à rendre fort sympathique le quotidien à l'école. Je pense notamment à Olivier Vallier, Guillaume Reymond (qui a la réponse à toutes les questions du type "Comment on fait pour faire proprement . . . ?"), Jacques Fournier, Jean-Max Dutertre, Driss Aboulkassimi, Amir-Pasha Mirbaha, Jean-Pierre Nikolovski, Philippe Maurine, Laurent Freund, Jérôme Quartana, Anne-Lise Ribotta, Jean-Baptiste Rigaud (quand il est énervé, on entend la marche impériale), Romain Wacquez, Kamil Gomina, Alexandre Sarafianos, Cyril Roscian, Florian Praden (alias JJ, un danger perpétuel pour tout système sécurisé qui tombe entre ses mains), Franck Courbon (comme quoi on peut être thésard et bien habillé), Stéphan De Castro, Sébastien Tiran (qui m'impressionne par sa zénitude face aux débats de ses voisins de bureau), Yanis Linge, Maxime Lecomte (un Breton pas chauvin, si, ça existe), Nicolas Borrel, Clément Champeix, Alexis Krakovinski, Serge Seyroles, Thierry Vaschalde, Claire Pechikoff, Loïc Lauro, Romain Ferrand, Marie Paindavoine, Jean-Yves Zie, Victor Murillo, Assia Tria, Marc Ferro (mon adversaire de baby-foot préféré), Cassandre Vuillaume, Scherrine Tria, Eloise Bihar, Manuelle Bongo, Pierre Leleux, David Elbaze, David Cambon, Thomas Lonjaret, Sylvain Blayac, Clément Talagrand, Henda Basti, Etienne Drahi, Mathieu Leroux, Thierry Camilloni, Oussama Benzaim, Cyril Calmes, Roger Delattre, Jean-Michel Li, Philippe Lalevée, Bénédicte Franceschi, Catherine Dessi, Christian Buatois, François Jaujard, Pascal Gelly, Karine Canivet, Michel Fiocchi, Véronique Villaréal, Georges Loubière, Jean-Michel Li, Nabil Absi, Dominique Feillet, Claude Yugma, Stéphane Dauzère-Pérès, Hervé Jacquemin, Jakey Blue, Gracien Counot, Manon Leoni, Florent Bitschÿ, Sabine Salmeron, Richard Lagaize, Michelle Gillet, Thierry Ricordeau, Phillipe Caillouet, Pierre Joubert. . . et j'en oublie très certainement ! Et bien entendu, il n'y avait pas que des thésards ou des professeurs dans cette école : il y avait aussi des élèves-ingénieurs. Je pense notamment à Stéphane Collot, Bertrand Rossignol, Erwan Henry, Laurent Legrigeois, Charles Nguyen, Geoffrey Walter, Mikael Marin, Ludovic Lourenço, Kévin Arth, Vincent Caravéo, Loïc Bois, Franck Valdenaire, Margaux Raffoul, Umar Saleem, Louiza Khati, Eleonore Hardy, . . . bref, à tous ceux avec qui j'ai pris beaucoup de plaisir à refaire le monde au Tibbar tous les jeudis soirs pendant deux ans (et j'en oublie encore !). Parmi les personnages marquants de ces six années passées à l'école, je me dois également de remercier Bernard Dhalluin, qui m'a fourni de précieux

conseils sur le monde de la recherche alors que j'étais encore élève-ingénieur et avec qui j'ai beaucoup aimé collaborer dans le cadre de l'association des anciens, ainsi que Jean-Paul Ramond également pour ses conseils sur la thèse et en partie grâce à qui j'ai pu effectuer mon stage de dernière année d'école à Taïwan. Cette thèse aurait également été bien plus difficile sans d'abord Bénédicte Messina, puis ensuite Patricia Murrisch, qui m'ont aidé à m'y retrouver dans les strates administratives plus que nombreuses du CEA et qui ont toujours fait preuve d'une grande patience face à mes appels au secours ou mes formulaires mal remplis.

Un remerciement spécial également à Hélène Le Bouder et Ronan Lashermes (nos deux Bretons un peu chauvins sur les bords) ainsi qu'à Thomas Sarno (mon grand adversaire de débats idéologiques sur à peu près tous les sujets), Marc Lacruche (ouais ouais ouais) et Ingrid Exurville (championne du monde du macaron au caramel et beurre salé). Ah, et j'allais justement oublier Patrick Haddad, notre fournisseur officiel de saucissons et de petits coussins lyonnais ! Je tiens également à remercier tous les proches et amis qui n'étaient pas présents au quotidien mais à qui j'ai tellement parlé de ce fameux quotidien que c'est un peu comme si finalement ils y avaient été. Je pense en particulier à Adrien, Pierre-Yves, Béné, Gaby, Laure, Sophie, . . . mais là, si je dois tous les lister ça risque de prendre encore plusieurs pages.

J'aimerais également remercier tout particulièrement les membres de ma famille, pour m'avoir toujours accompagné et soutenu pendant toutes ces années d'études supérieures et qui m'ont toujours permis de suivre la voie que je désirais. Je pense bien évidemment tout particulièrement à mes parents et mes deux frères Arnaud et Mr Big, ainsi que mes grand-parents et ma grand-mère, toutes ces personnes à qui je dois beaucoup et sans qui je ne serais pas là où j'en suis aujourd'hui.

Enfin, mes tous derniers remerciements ne peuvent qu'aller à Loïc et Yuting. Pendant ces trois années de thèse, Loïc a occupé à la fois les casquettes de co-thésard, voisin de bureau, colocataire et bien entendu interlocuteur privilégié pour refaire le monde (avec quand même une petite touche de mauvaise foi quant au tapis de la salle de bains). . . bref, heureusement qu'il était là ! Et Yuting, bien évidemment, que je remercie pour son soutien pendant toutes ces années, d'abord depuis Taïwan puis directement en France. Je sais que ces années de thèse auraient été beaucoup plus difficiles si elle n'avait pas été là. Et maintenant que la thèse est finie, comme elle me dit si souvent. . . « Il faut travailler le Chinois ! »

Résumé

Cette thèse s'intéresse à la sécurité des programmes embarqués face aux attaques par injection de fautes. La prolifération des composants embarqués et la simplicité de mise en œuvre des attaques rendent impérieuse l'élaboration de contre-mesures.

Un modèle de fautes par l'expérimentation basé sur des attaques par impulsion électromagnétique a été élaboré. Les résultats expérimentaux ont montré que les fautes réalisées étaient dues à la corruption des transferts sur les bus entre la mémoire Flash et le pipeline du processeur. Ces fautes permettent de réaliser des remplacements ou des saut d'instructions ainsi que des modifications de données chargées depuis la mémoire Flash.

Le remplacement d'une instruction par une autre bien spécifique est très difficile à contrôler ; par contre, le saut d'une instruction ciblée a été observé fréquemment, est plus facilement réalisable, et permet de nombreuses attaques simples. Une contre-mesure empêchant ces attaques par saut d'instruction, en remplaçant chaque instruction par une séquence d'instructions, a été construite et vérifiée formellement à l'aide d'outils de model-checking.

Cette contre-mesure ne protège cependant pas les chargements de données depuis la mémoire Flash. Elle peut néanmoins être combinée avec une autre contre-mesure au niveau assembleur qui réalise une détection de fautes. Plusieurs expérimentations de ces contre-mesures ont été réalisées, sur des instructions isolées et sur des codes complexes issus d'une implémentation de FreeRTOS. La contre-mesure proposée se révèle être un très bon complément pour cette contre-mesure de détection et permet d'en corriger certains défauts.

Mots-clés : attaques par injection de fautes ; injection électromagnétique ; modèle de fautes ; contre-mesures vérifiées ; assembleur ; saut d'instruction

Abstract

This thesis focuses on the security of embedded programs against fault injection attacks. Due to the spreadings of embedded systems in our common life, development of countermeasures is important.

First, a fault model based on practical experiments with a pulsed electromagnetic fault injection technique has been built. The experimental results show that the injected faults were due to the corruption of the bus transfers between the Flash memory and the processor's pipeline. Such faults enable to perform instruction replacements, instruction skips or to corrupt some data transfers from the Flash memory.

Although replacing an instruction with another very specific one is very difficult to control, skipping an instruction seems much easier to perform in practice and has been observed very frequently. Furthermore many simple attacks can carried out with an instruction skip. A countermeasure that prevents such instruction skip attacks has been designed and formally verified with model-checking tool. The countermeasure replaces each instruction by a sequence of instructions.

However, this countermeasure does not protect the data loads from the Flash memory. To do this, it can be combined with another assembly-level countermeasure that performs a fault detection. A first experimental test of these two countermeasures has been achieved, both on isolated instructions and complex codes from a FreeRTOS implementation. The proposed countermeasure appears to be a good complement for this detection countermeasure and allows to correct some of its flaws.

Keywords : fault injection attacks ; electromagnetic injection ; fault model ; verified countermeasures ; assembly ; instruction skip

Table des matières

Introduction générale

Cette thèse s'intéresse à la sécurité des processeurs embarqués et s'inscrit dans le cadre de la protection de ceux-ci face aux attaques physiques.

La sécurité des systèmes embarqués représente un enjeu majeur voire critique pour un grand nombre de secteurs de l'industrie ou d'organisations étatiques. Ce sous-domaine du vaste ensemble qu'est la sécurité des systèmes d'informations s'intéresse au cas particulier de la protection des systèmes embarqués face aux attaques dont ils peuvent faire l'objet. De par leur statut d'*embarqués*, les systèmes à protéger peuvent se retrouver entre les mains d'éventuels attaquants, qui chercheront à porter atteinte à la confidentialité, à l'intégrité ou à l'authenticité de données protégées. De tels systèmes peuvent alors être l'objet d'attaques dites *physiques* qui visent à exploiter des faiblesses dans leur implémentation.

Le principal marché concerné par les problématiques liées à la sécurité des systèmes embarqués est pour l'instant celui de la carte à puce, avec ses nombreuses déclinaisons comme les cartes de paiement, les cartes SIM ou la télévision à péage (Pay-TV). De par l'utilisation de la carte à puce dans des domaines toujours plus nombreux, d'après (EUROSMART, 2012) les ventes de carte à puce devraient atteindre 7 milliards d'unités en 2013, soit une croissance de 8% par rapport à leur niveau de 2012, avec une croissance de 25% en ce qui concerne les cartes à puce munies d'une technologie sans contact.

Toutefois, la carte à puce n'est pas le seul secteur pour lequel une vulnérabilité matérielle peut s'avérer critique : l'utilisation d'attaques physiques a montré son efficacité dans le débridage (plus connu sous le nom anglais de *jailbreak*) de produits grand public comme l'iPhone d'Apple [1] ou la console XBOX 360 de Microsoft [2], pouvant ainsi entraîner des préjudices financiers importants aux industriels dont les produits présentent des vulnérabilités exploitables.

À ces besoins en systèmes sécurisés sont venus s'ajouter d'autres utilisations des cartes à puce ces dernières années, avec la généralisation de l'ajout de composants électroniques sécurisés aux documents d'identité. En France, cette numérisation des pièces justificatives d'identité a d'abord commencé avec l'utilisation de la carte à

1. Full hardware unlock, 2007
2. DEBUSSCHERE et MCCAMBRIDGE, 2012.

puce comme document pour l'assurance maladie (Carte Vitale, 1998). Plus tard, les passeports (2005) et permis de conduire (2013) ont également été numérisés et utilisent des circuits sécurisés proches de ceux de la carte à puce. Enfin, les cartes nationales d'identité devraient également être concernées par cette numérisation dans les prochaines années.

L'ajout de modules de communication basés sur des technologies sans contact de radio-identification (plus connues sous l'acronyme RFID, Radio Frequency IDentification) a également entraîné l'utilisation de circuits sécurisés pour de nouveaux usages comme les titres de transport ou autres badges d'accès. L'apparition de nouvelles vulnérabilités exploitables face à ces circuits a donc logiquement suivi, avec par exemple le cassage des cartes MIFARE Classic dès 2008 (vendues à plus d'un milliard d'exemplaires aujourd'hui et encore largement utilisées pour du contrôle d'accès).

Parmi les attaques physiques que peut subir un circuit, les attaques par injection de fautes visent à modifier son comportement en injectant une faute lors d'un calcul. Ces attaques par injection de fautes peuvent être utilisées sur des implémentations cryptographiques mais également sur d'autres parties de circuits de sécurité, par exemple pour contourner des tests de code PIN ou empêcher l'exécution de fonctions essentielles. La définition de contre-mesures face à ces attaques physiques est difficile car elle requiert une bonne estimation des moyens d'un éventuel attaquant. Dans le cas particulier des attaques par injection de faute, l'effet lui-même de certains moyens couramment utilisés pour injecter des fautes reste assez mal connu. Il est donc extrêmement difficile d'avoir de bons modèles qui représentent l'ensemble des fautes qui peuvent être produites par un attaquant.

Cette thèse se place dans le cas de programmes embarqués et vise à améliorer leur résistance face à des attaques visant les processeurs qui les exécutent. Pour cela, une bonne connaissance du modèle de fautes réalisables par un éventuel attaquant est indispensable. Une présentation du contexte d'étude ainsi qu'un état de l'art des différentes méthodes d'attaque de circuits de sécurité et des contre-mesures existantes sont présentés dans le chapitre 1. Nous y précisons également les objectifs de la thèse. Le chapitre 2 présente ensuite le montage expérimental d'attaque utilisé dans cette thèse, ainsi que plusieurs résultats préliminaires d'utilisation de ce dispositif. Le chapitre 3 présente ensuite une étude détaillée des effets du dispositif expérimental utilisé sur le processeur embarqué choisi pour cette thèse. Le chapitre 4 propose ensuite une contre-mesure au niveau assembleur basée sur le modèle défini dans le chapitre précédent. Enfin, le chapitre 5 propose une première évaluation expérimentale de la contre-mesure proposée sur plusieurs exemples de codes embarqués.

Une partie des travaux présentés dans cette thèse apparaissent dans les publications suivantes : (Dehbaoui, Mirbaha et al., 2013), (Moro, Dehbaoui et al., 2013),

(Moro, Heydemann, Encrenaz et al., 2014), (Moro, Heydemann, Dehbaoui et al., 2014).

Contexte et motivations

1

Sommaire

1.1 Introduction

Cette thèse étudie le cas des attaques physiques, pour lesquelles un attaquant dispose d'un accès au circuit, celui-ci devant continuer à assurer ses propriétés d'intégrité, de confidentialité et d'authenticité en cas d'attaque. Ce chapitre commence tout d'abord par présenter une classification des types d'attaques et une brève description des techniques expérimentales existantes pour l'attaque de circuits en 1.2. Un état de l'art sur les attaques par observation et leurs contre-mesures est présenté en 1.3 suivi d'un état de l'art sur les attaques par injection de faute et leurs protections en 1.4. Enfin, le chapitre s'achève sur une présentation des objectifs de la thèse et de l'approche choisie pour les atteindre en 1.5.

1.2 Principes généraux des attaques physiques

Tous les algorithmes cryptographiques reposent sur des propriétés mathématiques qui ramènent la cryptanalyse à un problème reconnu comme mathématiquement difficile. Ce type de construction garantit ainsi la sécurité théorique des données chiffrées. Toutefois, la résistance théorique d'un algorithme ne garantit pas l'absence de failles concernant le circuit sur lequel il est implémenté. Dans cette thèse, on utilisera pour désigner les attaques qui exploitent ce type de vulnérabilités les expressions d'*attaques par canaux auxiliaires* ou d'*attaques physiques*. Le but de ces attaques physiques est généralement d'obtenir une donnée secrète comme une clé de chiffrement, ou bien d'obtenir des informations sur un circuit dans un but de rétro-ingéniérie, ou encore de contourner une protection comme un contrôle d'accès. Certaines attaques nécessitent également la réalisation préalable d'une étape de préparation du composant. Celle-ci consiste généralement en une ouverture du boîtier à l'aide d'acide. Selon le type d'attaque qui sera réalisé ensuite, la préparation peut également impliquer la pose de micro-sondes ou encore l'ouverture du boîtier en face arrière.

Cette section présente plusieurs principes généraux liés aux attaques physiques et commence par une présentation du cas de la cryptographie embarquée en 1.2.1. Ensuite, les canaux auxiliaires pouvant être utilisés pour des attaques sont présentés en 1.2.3. Enfin, les principes généraux utilisés pour les différentes attaques physiques existantes sont présentés en 1.2.2.

1.2.1 Cryptographie embarquée

La cryptologie se définit étymologiquement comme la science du secret. Elle regroupe deux disciplines liées : la cryptographie, étudiant l'écriture de messages secrets, et la cryptanalyse, visant au déchiffrement de messages secrets sans posséder la clé de chiffrement. La cryptographie est utilisée depuis l'antiquité, bien qu'elle soit restée principalement dans le domaine militaire jusqu'au milieu du vingtième siècle (STERN, 1998). A titre d'exemple, en France la cryptographie a été considérée comme une arme jusqu'à très récemment : il a fallu attendre 1999 pour que son usage soit autorisé aux particuliers. La deuxième moitié du vingtième siècle a vu l'émergence de nouveaux principes cryptographiques sur lesquels reposent les algorithmes modernes. Parmi ceux-ci, on peut notamment citer la sécurité théorique absolue (SHANNON, 1949), les protocoles d'échanges de clés (DIFFIE et HELLMAN, 1976) ou la cryptographie à clé publique (RIVEST et al., 1978).

La cryptographie moderne utilise plusieurs types de briques de base qui sont utilisées dans des cryptosystèmes plus complexes. Parmi ces briques de base, on peut mentionner les fonctions de hachage, les générateurs de nombres aléatoires, les algorithmes de chiffrements dits à *clé secrète* (également appelés algorithmes *symétriques*) et ceux dits à *clé publique* (également appelés algorithmes *asymétriques*). Certains

algorithmes ont acquis à la suite de concours internationaux le statut de standard (DES, AES, DSA, SHA-3), mais beaucoup d'autres algorithmes non-standardisés ont été publiés et sont couramment utilisés (RC4, RSA, MD5, SHA-1, ...). Néanmoins, un grand nombre d'algorithmes propriétaires dont la sécurité repose en partie sur la non-connaissance de l'algorithme par un attaquant sont encore largement utilisés.

1.2.2 Canaux auxiliaires et grandeurs observables

Les attaques physiques exploitent les failles dans le circuit sur lequel est implémenté un algorithme. Pour cela, elles se basent sur la mesure par un attaquant d'une grandeur relative aux données manipulées par le circuit. Dans la suite de cette thèse, on utilisera pour désigner cette grandeur les termes de *grandeur observable* ou de *grandeur mesurable*. La consommation de courant ou encore un résultat fauté après une injection de fautes peuvent par exemple être des grandeurs observables. De même, les données internes au circuit auquel l'attaquant n'a pas d'accès direct seront appelées *grandeurs internes*.

La notion de *canal auxiliaire* indique l'existence d'une relation entre une grandeur interne visée par un attaquant et une grandeur observable. Cette relation n'a pas forcément une expression mathématique et peut être dépendante du circuit visé ou des conditions expérimentales dans lesquelles l'attaque est réalisée. Elle peut néanmoins être approchée à l'aide de modèles. Ces modèles sont utilisés par un attaquant pour représenter le lien entre les grandeurs observables à sa disposition et les grandeurs internes visées par l'attaque. Ensuite, une technique d'exploitation du canal auxiliaire basée sur ce modèle est utilisée afin d'extraire les grandeurs internes visées. Les techniques d'exploitation des différents canaux auxiliaires seront présentées plus en détails dans les parties 1.3 et 1.4.

1.2.3 Canaux auxiliaires utilisés pour la réalisation d'attaques

D'une manière générale, il existe un aspect bidirectionnel pour les canaux utilisés pour réaliser des attaques physiques, et la plupart des canaux utilisés pour une attaque par injection de fautes peuvent également être utilisés d'une façon différente dans le cas d'une attaque par observation. Plus précisément, le tableau 1.1 présente plusieurs canaux physiques qui peuvent être utilisés pour les deux types d'attaque.

1.3 Attaques par observation et contre-mesures

La notion d'attaque par observation a été introduite dans la communauté scientifique à la fin des années 1990, d'abord en utilisant le temps d'exécution d'un programme (KOCHER, 1996) comme canal auxiliaire, puis la consommation de courant (KOCHER et al., 1999). Si certaines attaques par observation peuvent nécessiter une décapsulation préalable du circuit, la majorité de celles présentées dans la littérature ne

Table 1.1. : Exemples de canaux physiques pouvant être utilisés pour des attaques

	Observation	Injection de fautes
Alimentation du circuit	Mesure du courant consommé	Perturbation de la tension d'alimentation
Temps	Mesure du temps d'exécution	Perturbation du signal d'horloge et violation des contraintes temporelles
Rayonnement électromagnétique	Analyse des émissions électromagnétiques	Injection électromagnétique
Lumière	Analyse des émissions de photons	Lumière blanche focalisée ou laser
Température	Analyse de la température du circuit	Chauffage du circuit
Ondes acoustiques	Analyse du bruit émis par le circuit	(Pas encore de technique d'exploitation proposée)

requiert aucune modification, diminuant donc le coût de ces attaques et les rendant donc possibles pour un plus grand nombre d'attaquants potentiels. Cette section commence par lister les grandeurs observables par un attaquant pour réaliser des attaques par observation et expliciter le lien entre celles-ci et certaines grandeurs internes du circuit en 1.3.1. La réalisation d'une attaque par observation peut nécessiter le choix d'un modèle de consommation. Les principaux modèles de consommation utilisés dans la littérature scientifique sont présentés en 1.3.2. Ensuite, les techniques connues d'exploitation de ces données mesurées, pour obtenir une grandeur secrète comme une clé de chiffrement sont présentées en 1.3.3. Ces données mesurées peuvent également être utilisées dans un but de rétro-ingéniérie, ce cas est présenté en 1.3.4. Enfin, les différentes approches à la base des contre-mesures face aux attaques par observation sont présentées en 1.3.5.

1.3.1 Données observables par un attaquant

Les paragraphes qui suivent présentent les différentes grandeurs observables, explicitent leurs liens avec certaines grandeurs internes du circuit et présentent les différents moyens d'observation de ces grandeurs.

1.3.1.1. Sondage d'une donnée

La mesure directe par l'attaquant d'une donnée sur un des fils du circuit est appelée *sondage*, mais l'expression *micro-probing* est couramment utilisée. Dans ce type de procédé expérimental, l'attaquant vient placer une aiguille métallique sur un ou plusieurs des fils du circuit attaqué (généralement sur les bus) afin de pouvoir directement obtenir les valeurs logiques qui y transitent (Anderson et M. Kuhn, 1996 ; Kömmerling et M. G. Kuhn, 1999).

1.3.1.2. Consommation de courant

La mesure de la consommation de courant peut se faire en insérant une résistance de *shunt* (dont la valeur est connue) en série avec le composant entre celui-ci et la masse. La tension entre les bornes de cette résistance, qui peut être mesurée par un oscilloscope, est ainsi directement proportionnelle à l'intensité du courant consommé par celui-ci.

Une grande majorité des circuits intégrés actuels sont fabriqués à l'aide de la technologie Complementary Metal-Oxyde Semiconductor (CMOS). En technologie CMOS, la porte logique élémentaire NOT est réalisée à l'aide de deux transistors, un transistor NMOS et un transistor PMOS. En régime statique, cette porte logique a une consommation de courant I_{stat}. Le fait de positionner un 1 logique en entrée d'une telle porte met le transistor PMOS à l'état bloqué et le NMOS à l'état passant, reliant donc la sortie à l'état logique bas correspondant à la valeur 0. Inversement, dans le cas d'un 0 logique en entrée, l'état logique haut correspondant à un 1 logique se retrouve relié à la sortie. Toutefois, la transition entre un état logique et son complémentaire peut entraîner un léger pic sur la consommation de courant, comme expliqué dans (GUILLEY, HOOGVORST et al., 2004). Ce pic peut être la conséquence de deux courants :

— Un courant de court-circuit I_{cc}

— Un courant de charge de ligne I_{cl}

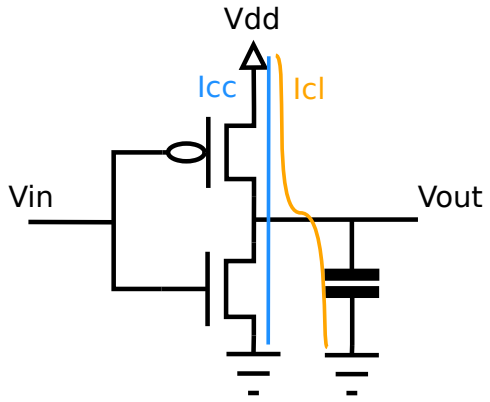


Figure 1.1. : Courants de charge de ligne et de court-circuit sur un inverseur CMOS

Ces deux courants à l'origine de pics sur la consommation de courant sont présentés sur la figure 1.1.

Au cours d'une transition de la sortie d'une porte logique, il existe un très bref instant pendant lequel les deux transistors laissent passer du courant, entraînant donc un léger court-circuit dû au passage d'un courant entre V_{dd} et la masse. Ce courant est appelé I_{cc}. Il apparaît indifféremment pour les deux types de transitions possibles.

La connexion entre la sortie de la porte NOT et une éventuelle porte suivante, ainsi que la proximité des lignes de connexion entraîne l'apparition de phénomènes capacitifs généralement modélisés par un condensateur parasite en sortie de l'inverseur.

Lors d'une transition entre un 0 et un 1 logique sur la sortie V_{out}, ce condensateur se retrouve connecté au V_{dd} et se charge, entraînant une légère surconsommation de courant.

Ainsi, au niveau d'une porte CMOS élémentaire, un attaquant capable de mesurer la consommation de courant globale de la porte est également capable :

— de distinguer une transition d'une absence de changement sur la sortie

— de distinguer une transition de 0 vers 1 d'une transition de 1 vers 0

Un circuit intégré étant constitué d'un ensemble de portes logiques, la consommation instantanée de courant du circuit est la somme des consommations de courants de chacune de ses portes logiques.

1.3.1.3. Rayonnements électromagnétiques

Les rayonnements électromagnétiques émis par un circuit peuvent être mesurés en champ proche. (GANDOLFI et al., 2001) et (QUISQUATER et SAMYDE, 2001) ont été les premiers articles à obtenir des résultats expérimentaux qui utilisent ce type de canal auxiliaire. La mesure des rayonnements électromagnétiques permet d'obtenir des mesures physiques plus localisées par rapport à la mesure de courant, mais cet effet localisé de l'analyse ajoute une contrainte liée au positionnement de la sonde électromagnétique. Pour définir quelles parties du circuit émettent des rayonnements électromagnétiques pouvant être utilisés pour une attaque, l'attaquant doit réaliser des cartographies spatiales en faisant varier la position de la sonde (DEHBAOUI, ORDAS et al., 2010), comme illustré sur la figure 1.2. Cette figure représente une cartographie des rayonnements électromagnétiques pour différents circuits intégrés. Elle montre également que les rayonnements mesurés sont liés à la position des blocs fonctionnels du circuit visé.

L'émission de rayonnement électromagnétique est une conséquence de plusieurs phénomènes, notamment la commutation des portes logiques. Celle-ci suit la loi de Lenz-Faraday. La rapide variation du courant s'accompagne d'une variation du champ magnétique (qui entraîne elle-même une variation du champ électrique) mesurable par l'attaquant en champ proche. Pour réaliser ces mesures, la littérature scientifique propose l'utilisation de sondes électromagnétiques qui peuvent être des boucles (PEETERS et al., 2007) ou des solénoïdes (MOUNIER et al., 2012).

1.3.1.4. Temps d'exécution

Le concept d'attaque temporelle a été introduit par (KOCHER, 1996). Il consiste à mesurer le temps d'exécution de l'algorithme voire, selon les implémentations et le degré de connaissance de l'attaquant, le temps d'exécution de certaines sous-fonctions. Par exemple, lors d'un branchement conditionnel, deux branches différentes de code

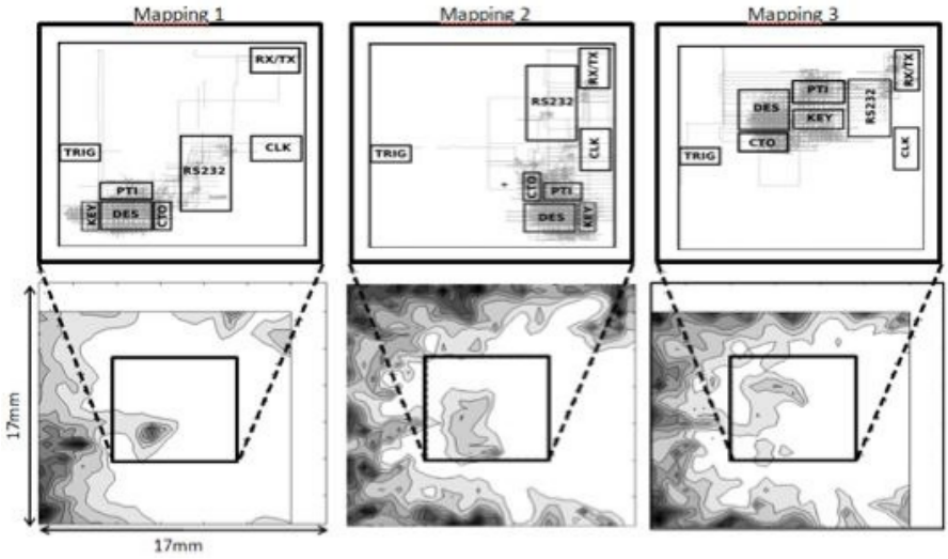


Figure 1.2. : Exemple de cartographie des rayonnements électromagnétiques pour plusieurs circuits intégrés (DEHBAOUI, ORDAS et al., 2010)

peuvent être appelées : une branche de code qui correspond au cas où la condition est vérifiée et une autre branche de code qui correspond au cas où la condition n'est pas vérifiée. Ces deux branches de code peuvent néanmoins présenter des différences de temps d'exécution. Un tel cas est illustré dans l'algorithme 1, où deux additions sont réalisées si la condition $n = 0$ est vérifiée et une seule addition sinon. Dans un tel cas de figure et pour un attaquant qui connaîtrait l'algorithme visé, la mesure du temps d'exécution de la fonction permet de savoir quelle branche de code a été exécutée et donc de connaître la valeur de la condition $n = 0$. Le temps d'exécution d'un algorithme peut également être lié à l'utilisation de mémoires cache par un processeur qui en est équipé. Ces mémoires cache peuvent introduire des différences de temps d'accès qui dépendent des données lues depuis la mémoire (BERNSTEIN, 2005 ; BERTONI et al., 2005).

Algorithme 1 : Exemple d'algorithme qui présente un déséquilibre du graphe de flot de contrôle

```
Data : x, y, z, n
1 if n == 0 then
2     x=x+1
3     y=y+1
4 else
5     z=z+1
```

La mesure du temps d'exécution peut être réalisée à l'aide d'un chronomètre déclenché automatiquement lors d'une requête au circuit et arrêté lors de la réponse de celui-ci. Elle peut également être réalisée indirectement en observant les émissions

du circuit sur un autre canal auxiliaire. Par exemple, l'observation de la consommation de courant peut permettre de repérer des motifs correspondant à certaines étapes importantes de l'algorithme. Un exemple d'utilisation de la consommation de courant pour repérer l'intervalle d'exécution d'un algorithme cryptographique est présenté en 1.3.4.1.

1.3.1.5. Émissions de photons

Très récemment, des travaux ont montré qu'il était possible d'utiliser les émissions de photons du circuit comme canal auxiliaire (SCHLÖSSER et al., 2012). Les auteurs de cette méthode proposent un modèle dans lequel le nombre de photons émis au niveau d'un transistor en technologie CMOS dépend de la tension appliquée à la grille du transistor. L'effet est davantage marqué sur les transistors NMOS. Plus précisément, un mouvement des porteurs de charge entre le drain et la source, qui intervient lorsque le transistor fonctionne en mode passant ou saturé, est accompagné d'une émission de photons. Ainsi, la quantité de photons émise dépend des données en entrée des portes logiques CMOS. Les auteurs mesurent les photons émis en face arrière du circuit à l'aide d'un capteur de type Charge-Coupled Device (CCD) et d'une photodiode.

1.3.1.6. Température

L'utilisation de la température comme grandeur mesurable a également été proposée à plusieurs reprises dans la littérature scientifique (BROUCHIER et al., 2009). Néanmoins, la première caractérisation de ce canal auxiliaire a été proposée récemment dans (SCHMIDT et HUTTER, 2013). Dans cet article, les auteurs utilisent une mesure de température sur des microcontrôleurs AVR et PIC et montrent qu'il existe une dépendance linéaire entre l'activité du circuit et les rayonnements thermiques émis par celui-ci.

1.3.1.7. Ondes acoustiques

Très récemment, des attaques acoustiques visant l'implémentation Rivest Shamir Adleman (RSA) du programme GnuPG ont été proposées dans (GENKIN et al., 2013). En raison de l'augmentation de la température à l'intérieur du circuit lors de calculs, une contrainte mécanique est appliquée à certains éléments du circuit et celle-ci entraîne l'émission d'ondes acoustiques qui peuvent être mesurées.

1.3.2 Modèles de consommation

La réalisation d'attaques par observation peut nécessiter le choix d'un modèle de consommation par l'attaquant. Ce modèle de consommation a pour but d'estimer

à l'aide d'une relation mathématique le lien entre une grandeur observable et une grandeur interne. La représentativité de chacun de ces modèles dépend du type de canal auxiliaire observé et de l'architecture du circuit ciblé. Le terme de *consommation* renvoie aux premières attaques pour lequel ces modèles ont été utilisés, qui se basaient sur une observation de la consommation de courant d'un circuit. Toutefois, ces modèles dits de *consommation* peuvent être utilisés pour d'autres grandeurs observables comme par exemple les rayonnements électromagnétiques (PEETERS et al., 2007), la température (SCHMIDT et HUTTER, 2013) voire le temps de calcul (LI et al., 2010). Dans la littérature scientifique, trois modèles de consommation sont couramment utilisés (PEETERS et al., 2007). Ces trois modèles sont présentés dans les paragraphes qui suivent.

1.3.2.1. Poids de Hamming

Le poids de Hamming d'un octet correspond au nombre de bits à 1 de l'octet.

$$hw(X) = \sum_{i=1}^{n}(X_i) \tag{1.1}$$

Ce modèle suppose que la consommation de courant associée à la manipulation d'un octet par le circuit est proportionnelle au nombre de bits à 1 de l'octet manipulé. Il suppose donc que la manipulation d'un 1 logique induit une consommation supérieure à la manipulation d'un 0 logique. Le modèle néglige également la consommation de courant associée aux transitions. Le poids de Hamming est généralement un modèle pertinent pour les circuits qui possèdent un bus préchargé.

1.3.2.2. Distance de Hamming

La distance de Hamming entre deux octets correspond au nombre de bits distincts entre ces deux octets.

$$hd(X, Y) = \sum_{i=1}^{n}(X_i \oplus Y_i) \tag{1.2}$$

Ce modèle suppose que le courant associé à l'inversion de la valeur d'un bit est supérieur à celui associé au maintien d'un bit à sa valeur précédente. Pour ce modèle, la consommation de courant liée à la manipulation d'un 1 logique ou un 0 logique est donc négligée au profit du nombre de transitions. A titre d'exemple, la distance de Hamming est souvent utilisée dans le cas d'une analyse de la consommation ou des rayonnements électromagnétiques émis par le composant.

1.3.2.3. Valeur de l'octet

Dans (GIERLICHS et al., 2008), les auteurs proposent également un autre modèle pour lequel la consommation du circuit pour effectuer un traitement sur un octet est modélisée directement par la valeur numérique de l'octet.

1.3.2.4. Bilan

Ces modèles de consommation permettent à l'attaquant de définir, pour un texte clair et une clé donnés, une estimation de la consommation ou de l'émission de rayonnements électromagnétiques. Lors d'une attaque sur un circuit pour lequel l'attaquant ne possède pas d'informations détaillées sur l'implémentation, le choix d'un modèle pour réaliser une attaque par canaux auxiliaires peut se faire de façon très empirique. Il existe plusieurs modèles communément admis dans la littérature, et l'attaquant peut tout simplement essayer successivement avec chacun de ceux-ci jusqu'à trouver un modèle qui lui permettra de mener à bien son attaque.

1.3.3 Exploitation des données obtenues

Une fois la phase d'acquisition d'une grandeur observable réalisée, une étape d'exploitation des données obtenues permet de finaliser l'attaque. Les différentes techniques d'exploitation permettant de retrouver une grandeur secrète sont présentées dans les paragraphes qui suivent.

1.3.3.1. Analyse simple

Les analyses simples permettent d'obtenir des informations sur une grandeur secrète d'un algorithme en cours d'exécution par une observation des émissions sur certains canaux auxiliaires. Dans le meilleur des cas, ces analyses simples peuvent permettre de réaliser une attaque à l'aide d'une seule acquisition d'une grandeur observable.

Consommation de courant L'analyse simple de la consommation, ou Simple Power Analysis (SPA), a été introduite par Kocher en 1999 (KOCHER et al., 1999). Celle-ci vise à déterminer directement, à partir d'une observation de la consommation de courant lors d'une exécution normale de l'algorithme, des informations sur le calcul effectué ou les données manipulées. A titre d'illustration, la figure 1.3 présente six courbes de consommation correspondant à la manipulation d'une variable prenant six valeurs différentes. Les six courbes de consommation peuvent aisément être distinguées sur l'un des pics de consommation. Leur observation peut donc permettre d'obtenir des informations sur les données manipulées par le circuit.

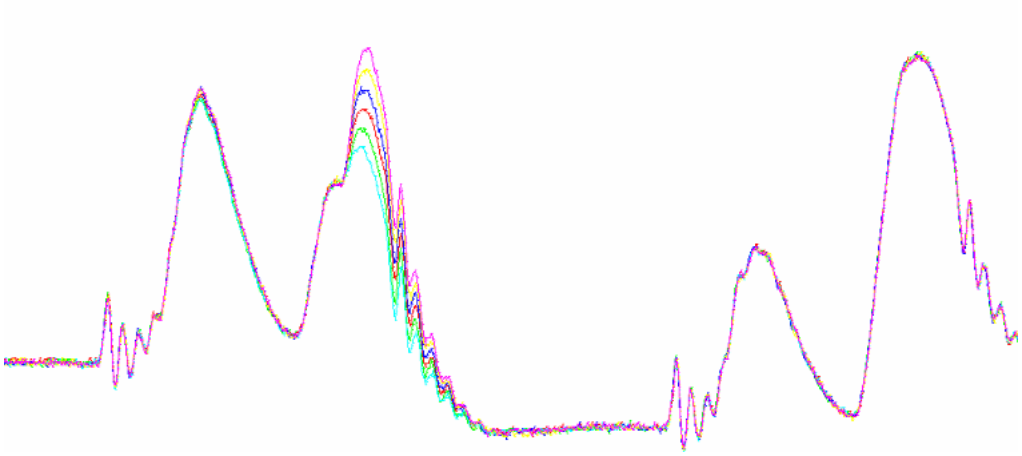


Figure 1.3. : Consommations de courant lors de la manipulation d'une variable pour 6 valeurs différentes (CLAVIER, 2007a)

Temps d'exécution Les premiers cas pratiques d'utilisation du temps d'exécution pour réaliser une attaque ont été présentés dans (KOCHER, 1996). Dans (DHEM et al., 1998), l'attaque vise une implémentation de l'algorithme RSA et a été réalisée sur la fonction d'exponentiation rapide (ou *Square and Multiply*). Cette fonction d'exponentiation rapide est présentée dans l'algorithme 2. L'attaque se base sur le fait que dans le cas d'une implémentation logicielle de cet algorithme, la condition n *est pair* revient à considérer la valeur du bit de poids faible de n. Si le bit de poids faible de n vaut 0, l'algorithme effectue une exponentiation. En revanche, si le bit de poids faible de n vaut 1, l'algorithme effectue une exponentiation et une multiplication. L'algorithme s'exécute donc plus rapidement si le bit de poids faible vaut 0.

Algorithme 2 : Algorithme d'exponentiation rapide

Data : x, n
Result : $y = x^n$
1 **if** $n = 1$ **then**
2 **return** x
3 **if** n *est pair* **then**
  `/* Une exponentiation */`
4 **return** $exponentiation(x^2, n/2)$
5 **else**
  `/* Une exponentiation et une multiplication */`
6 **return** $x \times exponentiation(x^2, n/2)$

1.3.3.2. Analyse différentielle

L'analyse différentielle de consommation, ou Differential Power Analysis (DPA) a été introduite par Kocher *et al.* (KOCHER et al., 1999) puis appliquée à d'autres canaux

(QUISQUATER et SAMYDE, 2001 ; AGRAWAL et al., 2003 ; GIERLICHS et al., 2008). Une analyse différentielle peut être réalisée en 5 étapes, qui sont présentées dans les paragraphes qui suivent.

1ère étape : Acquisition de courbes La première étape consiste à réaliser un ensemble d'acquisitions sur le circuit ciblé en faisant varier le message à chiffrer sur un ensemble T. Pour chaque chiffrement, la courbe correspondant à la grandeur observée est enregistrée. L'attaquant obtient donc une matrice de mesures M constituée de $|T|$ lignes, chaque ligne correspondant à une courbe d'acquisition de la grandeur observée lors d'un chiffrement. Le nombre de colonnes de M correspond au nombres d'instants échantillonnés lors de la mesure de la grandeur observée.

2ème étape : Prédiction de valeurs intermédiaires La deuxième étape consiste à choisir une valeur intermédiaire v du chiffrement qui dépend du texte clair [1] et d'un nombre limité de bits de la clé. Le nombre limité de valeurs que peuvent prendre ces bits de la clé définit un ensemble K de *clés partielles*. La valeur v est alors calculée pour l'ensemble T des textes clairs utilisés lors de l'étape d'acquisition et l'ensemble K des clés partielles. Une matrice V dont les coefficients sont les valeurs intermédiaires v est alors constituée.

3ème étape : Choix d'un modèle de consommation Cette étape consiste à choisir un modèle de consommation noté mc, qui est généralement choisi parmi les modèles présentés en 1.3.2. Ce modèle de consommation est utilisé pour lier les valeurs prédites lors de la deuxième étape de l'attaque à une estimation de la grandeur observable utilisée pour l'attaque. L'attaquant obtient donc une matrice de prédictions P avec $P_{i,j} = mc(V_{i,j})$. Dans l'attaque DPA de Kocher *et al.*, le modèle de consommation utilisé correspond simplement à la valeur du bit choisi lors de l'étape de prédiction des valeurs intermédiaires ; ce bit est appelé *bit de sélection*.

4ème étape : Comparaison entre mesures et prédictions La quatrième étape consiste à appliquer un opérateur statistique de similarité pour comparer les valeurs de la matrice de prédictions P à celles de la matrice de mesures M. Pour les courbes correspondant à une mauvaise hypothèse de clé partielle, les prédictions et les mesures ont un degré de similarité faible. Inversement, pour la courbe correspondant à la bonne hypothèse de clé partielle, les prédictions et les mesures ont une bonne similarité. L'attaquant obtient alors une matrice de résultats R où, pour chaque instant d'échantillonnage un scalaire mesurant la similarité est associé à chaque hypothèse de clé. L'attaque DPA de Kocher *et al.* utilise comme opérateur statistique une différence entre la moyenne des courbes pour lesquelles $P_{i,j} = 1$ et celles pour lesquelles $P_{i,j} = 0$ pour chaque hypothèse de clé partielle de l'ensemble K. Des variantes de l'analyse DPA ont ensuite été proposées. Elles mettent en œuvre d'autres

1. Le texte chiffré peut également être utilisé à la place du texte clair s'il est connu de l'attaquant, la valeur intermédiaire sera alors choisie dans les dernières étapes du chiffrement.

opérateurs et d'autres modèles. Parmi les opérateurs statistiques couramment utilisés, on trouve notamment la corrélation de Pearson (BRIER et al., 2004) ou l'information mutuelle (GIERLICHS et al., 2008).

5ème étape : Détection de la valeur maximale de la matrice de résultats La matrice R obtenue précédemment définit un ensemble de courbes de corrélation, avec une courbe de corrélation pour chaque hypothèse de clé. La figure 1.4 montre des exemples de courbes de corrélation pour trois hypothèses de clé (qui correspondent aux trois courbes du bas sur la figure). Sur la figure, la courbe associée à la première de ces hypothèses de clé montre un « pic de similarité » pour le résultat de l'opérateur statistique utilisé. L'attaquant peut en déduire que cette hypothèse de clé correspond à la clé partielle utilisée sur le circuit attaqué.

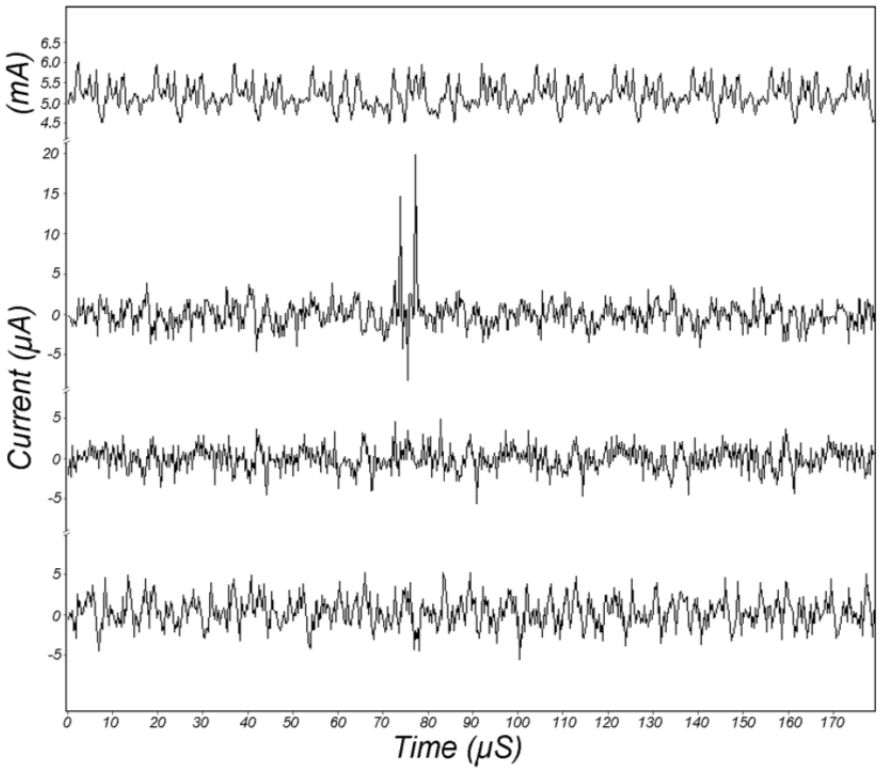


Figure 1.4. : Résultats d'analyse DPA pour trois hypothèses de clé (KOCHER et al., 1999)

1.3.3.3. Analyse par template

Les attaques par *template*, introduites par (CHARI et al., 2003), exploitent également des données obtenues via des canaux auxiliaires. Dans ce type d'attaque, l'attaquant réalise préalablement un ensemble de mesures sur un composant identique au composant ciblé tout en faisant varier un nombre de bits restreint de la clé de chiffrement. A partir de cet ensemble de mesures, il construit pour chaque hypothèse de clé un dictionnaire, ou *template*. Celui-ci est constitué de deux éléments : la

moyenne des courbes de consommation associées à la clé, et une information sur le bruit des mesures (sous la forme d'une matrice de covariance). Il réalise ensuite des mesures sur le circuit attaqué et recherche dans les *templates* constitués les échantillons les plus proches.

1.3.4 Rétro-ingéniérie

Des attaques par observation peuvent également être réalisées dans un but de rétro-ingéniérie pour retrouver des informations sur un circuit. Les paragraphes qui suivent en présentent quelques exemples.

1.3.4.1. Analyses simples

L'analyse SPA peut être utilisée pour permettre à un attaquant d'isoler une fenêtre temporelle pendant laquelle a lieu la partie du calcul visée. L'analyse SPA s'applique ainsi très bien au cas des algorithmes cryptographiques symétriques : ceux-ci ont des traces de consommation facilement reconnaissables de par leur structure qui consiste à répéter un certain nombre de fois une ronde de chiffrement. Un exemple de consommation de courant d'un circuit exécutant un chiffrement Data Encryption Standard (DES) à 16 rondes est présenté sur la figure 1.5. A partir de cette trace de consommation, il est possible d'en déduire la position des différentes rondes de chiffrement. Une application du principe de l'analyse SPA a également été proposée dans (NOVAK, 2003). L'auteur utilise des mesures effectuées lors de chiffrements pour retrouver les valeurs de tables de substitution d'un algorithme dont une partie de la sécurité repose sur sa non-connaissance. Cette attaque a ensuite été améliorée dans (CLAVIER, 2004), où l'auteur propose une construction sous forme de graphe permettant également de retrouver la clé de chiffrement en plus des valeurs des tables de substitution.

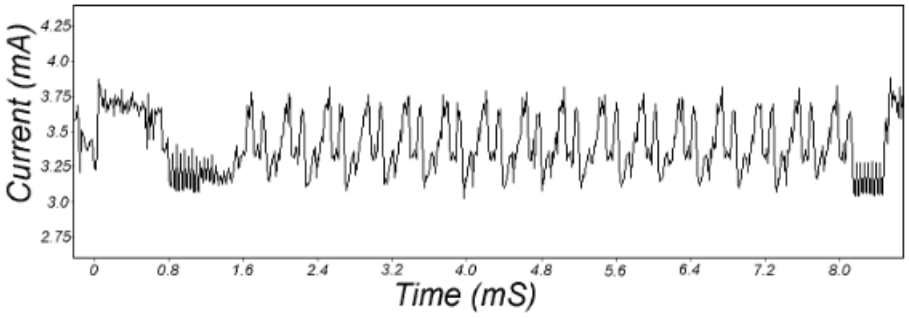


Figure 1.5. : Consommation de courant d'un circuit exécutant un chiffrement DES à 16 rondes (KOCHER et al., 1999)

1.3.4.2. Analyses par template

En utilisant le principe des attaques par *template*, un attaquant peut identifier une partie des instructions exécutées par un microcontrôleur. La figure 1.6 présente un exemple de construction de *template* sur des traces de courant pour deux suites d'instructions assembleur. Une telle approche a été proposée par (GOLDACK, 2008) puis (EISENBARTH et al., 2010). Dans ces deux articles, le programme exécuté par le microcontrôleur est décrit sous la forme d'un modèle de Markov caché, et les auteurs utilisent plusieurs algorithmes pour retrouver le flot d'exécution le plus probable à partir des *templates* constitués sur un modèle de référence du microcontrôleur ciblé. A partir de leurs mesures, ils sont ainsi capables de retrouver jusqu'à 70% des instructions réellement effectuées par le microcontrôleur.

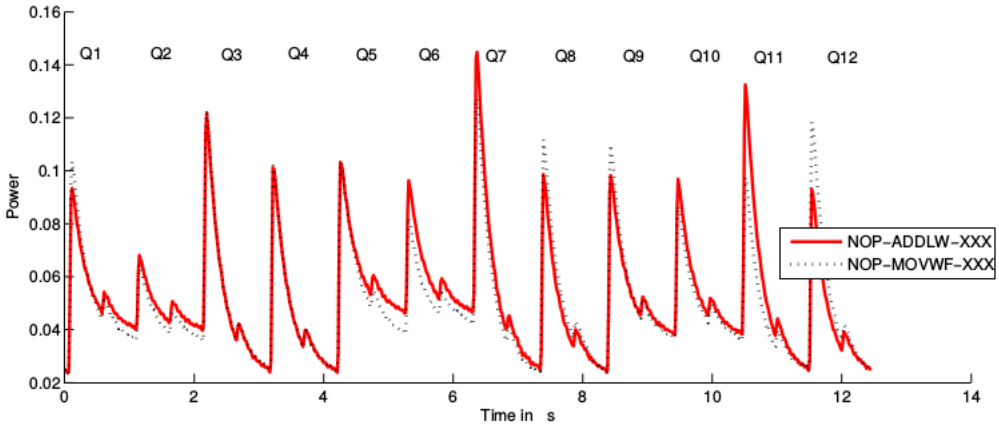


Figure 1.6. : Exemple de traces de courant utilisées pour la rétro-ingéniérie d'un programme assembleur (EISENBARTH et al., 2010)

1.3.5 Contre-mesures

Les contre-mesures face aux attaques par observation visent à réduire le rapport entre le signal transmis et le bruit. Pour cela, deux types d'approches peuvent être utilisées : celles consistant à réduire le signal transmis sur un canal auxiliaire (présentées en 1.3.5.1) et celles consistant à ajouter du bruit (présentées en 1.3.5.2).

1.3.5.1. Réduction du signal

De nombreuses techniques permettent de réduire le signal transmis sur un canal auxiliaire. Les paragraphes qui suivent présentent les principales techniques utilisées dans la littérature scientifique.

Équilibrage au niveau du graphe de flot de contrôle Pour contrer les attaques par analyse du temps d'exécution, les contre-mesures visent à équilibrer le graphe de

flot de contrôle des portions sensibles de l'algorithme en cours d'exécution. Pour atteindre ce but, une approche consiste à élaborer un code source présentant un graphe de flot de contrôle équilibré et pour lequel la compilation n'apporte pas de modification à cet équilibre (MOLNAR et al., 2006). Une autre approche consiste à modifier directement le compilateur de façon à permettre au programmeur d'annoter les portions de code sensible et ainsi de signaler les portions de code pour lesquelles devra être généré un code machine présentant un graphe de flot de contrôle équilibré (COPPENS et al., 2009). Dans le cas des analyses du temps de consommation qui exploitent les optimisations apportées par les mémoires caches, une solution possible consiste à désactiver toute utilisation de mémoire cache.

Équilibrage au niveau des données Les codages *1 parmi n* sont une approche possible pour équilibrer le poids de Hamming des données manipulées. Dans un codage *1 parmi n*, un seul fil est actif pour représenter une donnée, le poids de Hamming est donc constant. Un codage *1 parmi 2* (également appelé *double-rail*) est couramment utilisé dans la littérature scientifique (TIRI et VERBAUWHEDE, 2004). Afin d'équilibrer également les distances de Hamming au niveau des transitions, une phase dite de *précharge* peut être ajoutée avant chaque cycle de calcul. Cette phase de précharge consiste à placer les deux fils correspondant à un signal à une valeur définie (0 ou 1). Ensuite, lors de la phase de calcul, une seule transition est alors réalisée sur un des fils pour chaque signal. Plusieurs exemples de logiques double-rail à précharge sont présentés dans (GUILLEY, SAUVAGE et al., 2008) ou (DANGER et al., 2009). Au niveau des programmes embarqués, il est possible d'utiliser une variante du principe des logiques double-rail. Une telle approche a été proposée récemment dans (RAUZY, GUILLEY et NAJM, 2014). Pour cela, les auteurs décomposent chaque opération sous la forme d'une suite d'opérations élémentaires bit à bit. Chacune de ces opérations élémentaires charge ensuite ses résultats depuis une table précalculée en mémoire au lieu de faire appel à l'unité arithmétique et logique du processeur.

1.3.5.2. Ajout de bruit

Les paragraphes qui suivent présentent les principales techniques connues dans la littérature scientifique pour augmenter le bruit des signaux transmis via les canaux auxiliaires.

Ajout d'instructions factices L'ajout d'instructions factices permet de modifier le temps d'exécution ou la consommation de courant d'une portion de code embarqué. Une approche possible est de repérer les instructions les plus vulnérables aux attaques par consommation, puis de rajouter du bruit sous la forme d'instructions factices autour de ces instructions sensibles. Une telle approche est par exemple présentée dans (BAYRAK et al., 2011).

Code auto-modifiant et polymorphisme Des approches à base de code auto-modifiant peuvent également être utilisées pour ajouter du bruit sur un code embarqué. Un code auto-modifiant est un programme qui peut modifier à la volée des parties de son code. Pour cela, une approche possible est de définir un schéma à base de recompilation dynamique directement sur le processeur en utilisant des classes d'équivalence pour différentes instructions. Ce type de protection permet au processeur de compiler différentes séquences d'instructions pour réaliser une opération (AGOSTA et al., 2012). Une autre approche possible est d'éviter une recompilation réelle sur le processeur et lui faire choisir une version du code parmi celles dont il dispose en mémoire pour réaliser une opération bien définie. Cette approche a par exemple été appliquée à une implémentation de l'algorithme Advanced Encryption Standard (AES) dans (AMARILLI et al., 2011).

Modification de la tension ou du signal d'horloge Pour un circuit CMOS synchrone, la tension d'alimentation ou le signal d'horloge peuvent subir de petites variations sans que des fautes apparaissent dans les calculs. Une légère jigue sur la tension ou une légère désynchronisation du signal d'horloge peuvent par exemple être utilisées pour rendre plus difficile la réalisation pratique d'attaques par observation (GIRAUD et THIEBEAULD, 2004).

Masquage Le masquage consiste à appliquer un masque aléatoire aux données en entrée d'un algorithme. Le circuit applique alors une variante de l'algorithme en question à ces données masquées. Enfin, le masque est à nouveau appliqué au résultat de l'algorithme pour obtenir le résultat non-masqué. Cette contre-mesure rend caduques les prédictions faites par un attaquant lors d'une analyse DPA. Elle peut s'appliquer directement à tout algorithme linéaire mais doit nécessiter certaines modifications de fonctions dans le cas d'algorithmes non-linéaires. Cette technique de masquage a été utilisée pour renforcer de nombreuses implémentations cryptographiques (MESSERGES, 2001 ; JOYE et al., 2005 ; FUMAROLI et al., 2010).

1.4 Attaques par injection de faute et contre-mesures

Les attaques par injection de faute ont été introduites par Boneh *et al.* en 1997 (BONEH et al., 1997). Cette section commence par présenter les différents moyens d'injection de fautes connus en 1.4.1. Ensuite, la notion de *modèle de fautes*, qui fait le lien entre l'injection de fautes à l'aide de moyens physiques et la définition d'attaques et contre-mesures à un niveau d'abstraction supérieur, est présentée en 1.4.2. Les techniques d'exploitation des données obtenues à la suite d'une injection de fautes sont présentées en 1.4.3. Enfin, les différentes approches pour la définition de contre-mesures sont présentées en 1.4.4.

1.4.1 Moyens d'injection de faute

Divers phénomènes physiques permettent d'insérer une faute lors d'un calcul. Les moyens d'injection de faute connus et référencés par la communauté scientifique sont le rayonnement lumineux, la température, la perturbation du signal d'horloge, la perturbation de la tension d'alimentation ou encore l'injection électromagnétique (BARENGHI, BREVEGLIERI, KOREN et NACCACHE, 2012). Ces différents moyens d'injection sont présentés plus en détail dans les paragraphes qui suivent.

1.4.1.1. Rayonnement lumineux

Un rayonnement lumineux émis par un laser ou une source lumineuse focalisée peut être utilisé pour injecter des fautes dans un circuit. Un tel dispositif d'injection de fautes nécessite une ouverture du boîtier, comme illustré sur la figure 1.7.

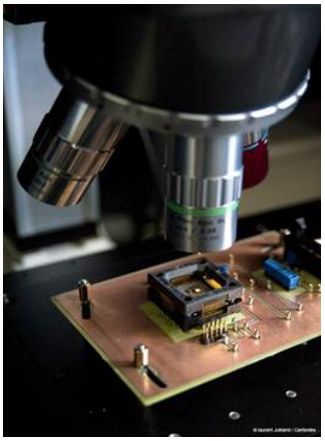


Figure 1.7. : Injection de fautes par laser sur un circuit intégré ouvert

L'énergie du rayonnement lumineux utilisé est absorbée par le silicium du circuit. Lorsque l'énergie transmise est supérieure au seuil permettant à des électrons de passer dans la bande de conduction du silicium, des paires électrons-trous sont alors créées le long du faisceau lumineux (ROSCIAN et al., 2013 ; J.-M. DUTERTRE, FOURNIER et al., 2011). Ce phénomène est illustré sur la figure 1.8. Ces paires électrons-trous peuvent aboutir à l'apparition d'un courant photoélectrique au niveau d'un transistor. Ce courant entraîne alors l'apparition d'un pic de tension qui peut se propager dans des blocs de logique combinatoire. Ce phénomène de propagation d'un pic de tension à travers la logique combinatoire est appelé Single Event Transient (SET). Le pic de tension induit peut entraîner l'apparition d'une faute s'il est échantillonné par un élément mémoire comme un registre.

Dans (S. P. SKOROBOGATOV et ANDERSON, 2003), les auteurs décrivent une attaque basée sur l'utilisation de l'effet photoélectrique pour insérer des fautes dans un circuit : ils utilisent ainsi un flash d'appareil photo contre une cellule de mémoire Static Random Access Memory (SRAM). Plus récemment, dans (SCHMIDT et HUTTER,

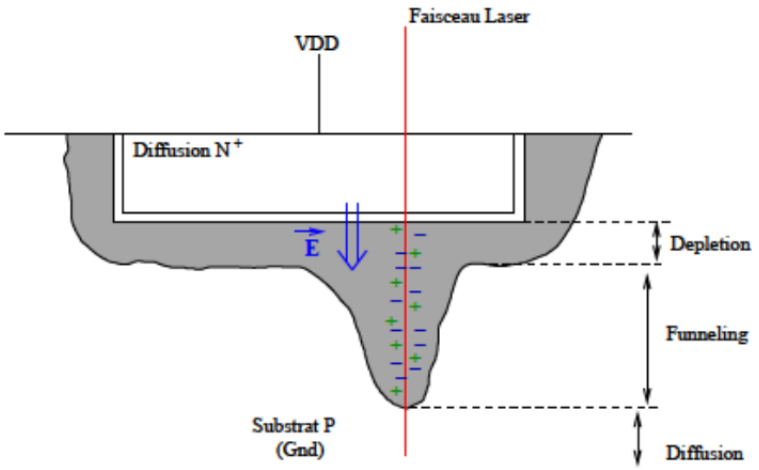


Figure 1.8. : Apparition de paires électrons-trous lors d'un tir laser sur une jonction PN (CYRIL ROSCIAN, 2013)

2007), les auteurs ont utilisé une fibre optique afin de mieux focaliser la source de lumière blanche sur la cellule de SRAM visée. Dans (ROSCIAN et al., 2013), les auteurs ont utilisé une source laser pour injecter une faute dans une cellule SRAM. Par rapport à l'utilisation de lumière concentrée, le laser permet une bien plus grande précision lors d'une injection de faute (CANIVET et al., 2010). Des effets sur certains blocs de logique d'un microcontrôleur ont également été obtenus dans (TRICHINA et KORKIKYAN, 2010), ceux-ci ont permis de fauter l'exécution d'instructions assembleur sur le microcontrôleur ciblé.

1.4.1.2. Température

Les circuits intégrés sont conçus pour fonctionner dans une certaine plage de température. Un attaquant peut faire chauffer certains éléments du circuit en dehors de leurs conditions normales de fonctionnement et ainsi créer des fautes à l'intérieur du circuit. Dans (S. SKOROBOGATOV, 2009), l'auteur a utilisé une modification de la température du circuit afin d'injecter des fautes dans des mémoires EEPROM et Flash. Plus récemment, dans (SCHMIDT et HUTTER, 2013), les auteurs ont attaqué une implémentation de l'algorithme RSA sur un microcontrôleur ATmega162 en faisant chauffer le circuit sur une plaque chauffante.

1.4.1.3. Perturbation du signal d'horloge

Les circuits synchrones cadencés par un signal d'horloge ont une fréquence de fonctionnement maximale. Celle-ci est définie en considérant la durée maximale d'un transfert de données entre deux éléments de mémorisation. Si l'attaquant réussit à forcer une fréquence d'horloge supérieure à cette fréquence maximale (on parle alors d'*overclocking*), la période d'horloge devient alors plus courte que la durée de certains

transferts dans la logique combinatoire. Dans ce cas, des valeurs incorrectes sont échantillonnées par certains éléments de mémorisation. Ce procédé d'*overclocking* peut être appliqué de façon à induire des fautes sur un seul cycle d'horloge. On parle alors de perturbation transitoire du signal d'horloge (couramment appelée *clock glitch*) (AGOYAN et al., 2010). La figure 1.9 présente un exemple de signal d'horloge modifié pouvant être utilisé pour une telle attaque.

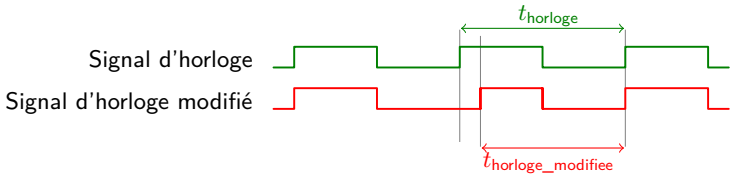


Figure 1.9. : Signal d'horloge modifié pouvant être utilisé pour une attaque

1.4.1.4. Perturbation de la tension d'alimentation

Les modifications de la tension d'alimentation ont des effets sur la vitesse de cheminement de données dans la logique combinatoire entre des éléments de mémorisation (ZUSSA, J.-m. DUTERTRE et al., 2012). En modifiant la valeur de tension utilisée pour alimenter un circuit, un attaquant peut donc induire des fautes dans les calculs qui sont effectués. Afin d'injecter une faute à un instant précis, un attaquant peut réaliser une perturbation transitoire de la tension d'alimentation (couramment appelée *power glitch*) afin de temporairement modifier la vitesse de circulation des données entre deux éléments de mémorisation (CARPI et al., 2013).

1.4.1.5. Injection électromagnétique

Le couplage électromagnétique entre une antenne et certains éléments du circuit visé peut provoquer des courants sur certains fils du circuit. Plusieurs types d'antennes peuvent être utilisées pour cela. Dans (SCHMIDT et HUTTER, 2007), les auteurs ont utilisé un solénoïde. Dans (DEHBAOUI, J.-M. DUTERTRE, ROBISSON, ORSATELLI et al., 2012), les auteurs ont ajouté un matériau ferromagnétique comme cœur du solénoïde. La figure 1.10 présente quelques antennes qu'il est possible d'utiliser pour réaliser une injection électromagnétique de fautes. Dans la littérature scientifique, l'injection électromagnétique a été utilisée principalement sous forme harmonique ou bien sous forme d'impulsions. La première, notamment présentée dans (POUCHERET et al., 2011) ou (MARKETTOS, 2011), a principalement été utilisée pour perturber des générateurs de nombres aléatoires et y introduire un biais statistique qui peut alors être utilisé par un attaquant. La seconde, introduite dans (QUISQUATER et SAMYDE, 2001) et présentée dans (SCHMIDT et HUTTER, 2007), (DEHBAOUI, J.-M. DUTERTRE, ROBISSON et TRIA, 2012) et (DEHBAOUI, J.-M. DUTERTRE, ROBISSON, ORSATELLI et al., 2012) a été utilisée pour réaliser des injections de fautes sur des implémentations

matérielles et logicielles d'algorithmes cryptographiques. Par rapport à l'injection harmonique, l'injection de fautes par impulsion permet de viser un instant particulier de l'exécution de l'algorithme ciblé.

Figure 1.10. : Exemples de sondes d'injection électromagnétique (DEHBAOUI, 2011)

1.4.2 Modèles de fautes

Différents modèles de fautes peuvent être utilisés pour représenter les types de fautes obtenus à l'aide des moyens d'injection présentés précédemment. Plusieurs attaques visant des implémentations cryptographiques supposent qu'un attaquant est capable de réaliser des fautes selon un de ces modèles. L'article (VERBAUWHEDE et al., 2011) présente une revue bibliographique des différents modèles de fautes existants et l'article (BARENGHI, BREVEGLIERI, KOREN et NACCACHE, 2012) résume les différents types d'attaques physiques possibles et présente les modèles de fautes couramment associés.

1.4.2.1. Modèles de fautes au niveau logique

Une injection de fautes peut avoir plusieurs types d'effets sur un bit. Ceux-ci sont :

- *Bit flip* : la valeur du bit est inversée
- *Bit set* : le bit est forcé à 1
- *Bit reset* : le bit est forcé à 0
- *Collage* : le bit est maintenu à sa valeur précédente

Aspect déterministe La plupart des moyens d'injection de fautes génèrent en réalité plusieurs types de fautes à l'intérieur du circuit. Chacun des type de fautes présentés précédemment peut donc être associé à une probabilité de réalisation.

Localisation temporelle Dans certains cas, des fautes peuvent être injectées de façon permanente sur une donnée. Néanmoins, la plupart des scénarios d'attaque se basent sur la possibilité de réaliser des fautes transitoires. Dans le cas de fautes

transitoires, la précision temporelle peut également varier. Par exemple, une attaque très précise peut permettre d'injecter des fautes sur un seul cycle d'horloge alors que des attaques moins précises ne peuvent pas le permettre.

Localisation spatiale La précision spatiale des fautes peut varier et certaines fautes peuvent modifier plusieurs parties d'un circuit. Lorsqu'une faute est injectée sur un élément du circuit, celle-ci peut également atteindre un ou plusieurs bits (et ainsi modifier la valeur d'un octet ou d'un mot mémoire). Par exemple, un modèle de fautes sur un octet est parfois considéré dans la littérature, pour lequel l'attaquant peut forcer un octet à 0x00 ou 0xFF (A. A. SERE et al., 2009).

1.4.2.2. Modèles de fautes au niveau algorithmique

Des modèles de fautes au niveau algorithmique peuvent également être considérés pour certaines attaques. Les modèles principaux utilisés dans la littérature scientifique sont le remplacement d'instruction (avec le cas particulier du modèle de saut d'instruction) et la corruption de tests conditionnels.

Remplacement d'instructions Sur un processeur embarqué, certains types d'attaque peuvent entraîner l'exécution d'une autre instruction suite à une modification du code de l'instruction au moment de son transfert sur le bus d'instructions (TRICHINA et KORKIKYAN, 2010 ; BALASCH et al., 2011 ; HUMMEL, 2014). Ce cas du remplacement d'une instruction par une autre a été rarement étudié dans la littérature scientifique, notamment en raison du manque de connaissances sur les conséquences d'une injection de fautes au niveau des instructions. Néanmoins, plusieurs articles ont étudié le cas de la corruption d'une instruction de branchement inconditionnel (BOUFFARD et al., 2011), ou encore le cas du remplacement d'une instruction par un branchement (BERTHOME et al., 2012). Un modèle simplifié dans lequel un attaquant est capable d'empêcher l'exécution d'une instruction est également régulièrement utilisé (NACCACHE, 2005 ; BARENGHI, BREVEGLIERI, KOREN et NACCACHE, 2012).

Corruption de tests conditionnels Les tests conditionnels représentent une partie sensible du graphe de flot de contrôle d'un algorithme. La corruption d'un de ces tests peut donc aboutir à une modification dans l'exécution de l'algorithme. Un attaquant peut corrompre un test conditionnel en injectant une faute sur la donnée testée dans la condition. Au niveau d'un programme embarqué, un attaquant peut également corrompre l'instruction qui réalise le test conditionnel, ce qui constitue un cas particulier de remplacement d'instruction. Par exemple, dans (BARBU et al., 2011) les auteurs utilisent une attaque laser pour corrompre un branchement conditionnel sur une machine virtuelle Javacard.

1.4.3 Exploitation des données obtenues

Les paragraphes qui suivent présentent les principales techniques d'exploitation des données obtenues à la suite d'une injection de fautes.

1.4.3.1. Analyse différentielle

L'exploitation de couples C, C' (un texte chiffré et un texte chiffré fauté, pour les mêmes valeurs de texte clair et de clé) est à la base de l'analyse différentielle de fautes. Celle-ci a été introduite par Boneh *et al.* (BONEH et al., 1997) et l'acronyme Differential Fault Analysis (DFA) a ensuite été proposé dans (BIHAM et SHAMIR, 1997). Chacune de ces attaques DFA est généralement basée sur un modèle de fautes et nécessite la réalisation d'un type de faute à un instant de l'exécution de l'algorithme. Au niveau des attaques par DFA sur l'algorithme AES, on peut notamment citer (PIRET et QUISQUATER, 2003), (GIRAUD, 2005), (MORADI et al., 2006) et (LASHERMES et al., 2012).

1.4.3.2. Safe error

Dans une attaque en *safe-error*, l'attaquant va exploiter le fait qu'une injection de fautes à un moment de l'algorithme produise ou non un résultat fauté en sortie (JOYE, 2000). Une variante de la technique de *safe-error*, nommée Ineffective Fault Analysis (IFA) a également été proposée dans (CLAVIER, 2007b). Leur principe est quasiment similaire, mais l'analyse IFA se place au niveau d'une instruction assembleur, alors que la technique de *safe-error* étudie les sorties fautées davantage au niveau de l'algorithme. Une autre variante d'attaque en *safe-error* est proposée dans (ROBISSON et MANET, 2007). L'attaque proposée exploite à la fois un principe de *safe-error* et des moyens statistiques similaires à ceux de l'analyse DPA.

1.4.3.3. Modification dans l'exécution de l'algorithme

Les modifications dans l'exécution de l'algorithme peuvent notamment permettre de contourner un contrôle d'accès ou d'empêcher l'exécution de certaines fonctions (WOUDENBERG et al., 2011). De telles modifications peuvent notamment être la conséquence de la corruption d'un branchement ou d'un test conditionnel, ou encore être la conséquence du remplacement ou du saut d'une instruction. Plusieurs attaques ont ainsi exploité le cas d'un saut d'instruction ou le saut d'une sous-fonction (SCHMIDT et HERBST, 2008 ; BLÖMER et al., 2014). De telles modifications peuvent par exemple être utilisées pour permettre l'exécution de code arbitraire par dépassement de tampon (FOUQUE et al., 2012) ou modifier le nombre de rondes d'un algorithme de chiffrement (J.-M. DUTERTRE, MIRBAHA et al., 2012 ; DEHBAOUI, MIRBAHA et al., 2013).

1.4.3.4. Rétro-ingéniérie

Les attaques par injection de fautes ont également été utilisées dans un but de rétro-ingéniérie, notamment pour identifier certains éléments d'algorithmes partiellement secrets. (San Pedro et al., 2011) et (Le Bouder et al., 2013) ont ainsi conçu un procédé de rétro-conception pour retrouver les boîtes de substitution (S-Box) d'un algorithme DES modifié. Dans (Clavier et Wurcker, 2013), les auteurs présentent une méthodologie de rétro-conception pour caractériser l'ensemble des opérations d'un algorithme AES modifié. Pour cela, les auteurs utilisent une technique d'IFA et un modèle de fautes pour lequel un attaquant peut mettre à zéro la valeur d'un octet.

1.4.4 Contre-mesures

Les paragraphes qui suivent présentent les principales approches proposées dans la littérature scientifique pour renforcer des circuits intégrés face aux injections de faute.

1.4.4.1. Renforcements au niveau de la technologie CMOS

Des modifications au niveau de la technologie de fabrication des circuit peuvent permettre de diminuer leur sensibilité à certains types de moyens d'injection de fautes. Par exemple, dans (Sarafianos et al., 2013) les auteurs proposent un changement dans la structure des cellules mémoire SRAM pour renforcer leur résistance face aux attaques par laser.

1.4.4.2. Capteurs physiques

Plusieurs capteurs physiques peuvent permettre de détecter des tentatives d'attaque. Parmi ceux-ci, on peut notamment mentionner les capteurs de lumière, de variations anormales de la tension d'alimentation ou du signal d'horloge. Un autre mécanisme de détection au niveau de la logique a également été proposé pour détecter des fautes injectées par violation des contraintes temporelles des circuits synchrones (J.-M. Dutertre, Fournier et al., 2011 ; Zussa, Dehbaoui et al., 2014). Ce mécanisme consiste à ajouter un signal de délai entre deux éléments de mémorisation synchrones pour assurer que le bloc de logique combinatoire situé entre ces éléments de mémorisation a eu le temps de terminer son calcul.

1.4.4.3. Mécanismes de détection de fautes

Cette classe de contre-mesures vise à détecter des fautes injectées dans un circuit par un attaquant. Plusieurs approches possibles sont présentées dans les paragraphes qui suivent.

Redondance temporelle et spatiale Dans le cas des attaques par injection de faute, les contre-mesures existantes visent principalement à ajouter de la redondance dans les calculs effectués par le circuit de manière à pouvoir détecter une éventuelle faute. Cette redondance peut être spatiale (où plusieurs éléments distincts du circuit réalisent la même opération en parallèle) ou bien temporelle (une même opération est répétée plusieurs fois) (BAR-EL et al., 2006). Les contre-mesures par redondance spatiale s'appliquent aux architectures matérielles, où le concepteur de circuit peut régler l'emplacement des différents blocs qui constituent son algorithme. Un schéma de principe d'une contre-mesure par redondance spatiale est présenté sur la figure 1.11. Les logiques double-rail à précharge (présentées en 1.3.5.1) peuvent également être utilisées comme mécanisme de redondance spatiale (SELMANE et al., 2009).

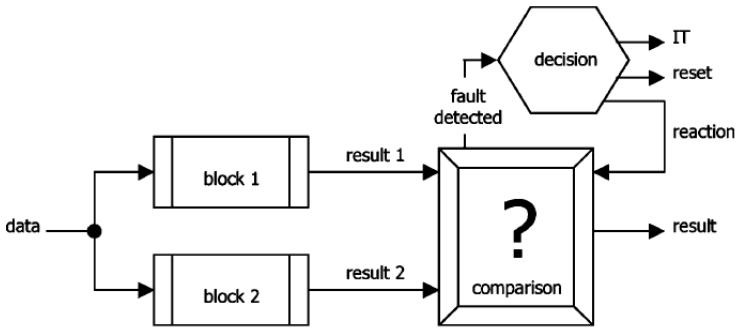


Figure 1.11. : Exemple de contre-mesure par redondance spatiale (BAR-EL et al., 2006)

Sur un processeur embarqué, la redondance temporelle peut être effectuée à plusieurs niveaux : à l'échelle de l'algorithme ou de ses sous-fonctions (en répétant la même fonction ou appliquant la fonction inverse) (BAR-EL et al., 2006), au niveau du code source en vérifiant chacune des opérations (WITTEMAN, 2008) ou enfin au niveau du code compilé à l'échelle de l'instruction assembleur (BARENGHI, BREVEGLIERI, KOREN, PELOSI et al., 2010).

Bits de parité et codes détecteurs d'erreur Plusieurs autres méthodes permettant de détecter des fautes intentionnellement injectées ou des erreurs dans un calcul ont également été proposées. Parmi celles-ci, on peut notamment citer l'utilisation de bits de parité (BARENGHI, BREVEGLIERI, KOREN, PELOSI et al., 2010) ou d'autres codes détecteurs d'erreur.

Utilisations des propriétés mathématiques des algorithmes Le cas des attaques par injection de fautes a été particulièrement traité au niveau des algorithmes cryptographiques. De nombreuses contre-mesures au niveau algorithmique ont alors été proposées. En particulier, pour l'algorithme RSA (et pour son implémentation CRT-RSA qui utilise le théorème des restes Chinois), on peut notamment citer (AUMÜLLER et al., 2003), (VIGILANT, 2008), (CORON et al., 2010), ou (JOYE, 2012). Ces contre-mesures évaluent la validité des relations mathématiques entre variables

en certains points de l'algorithme. La corruption de ces relations traduit l'apparition d'erreurs. Un ensemble de contre-mesures pour plusieurs algorithmes dont AES et RSA est également présenté dans (BARENGHI, BREVEGLIERI, KOREN et NACCACHE, 2012).

Contrôle du flot d'exécution Le contrôle du flot d'exécution consiste à vérifier que l'algorithme en cours d'exécution sur un circuit suit bien le flot d'exécution prévu par le concepteur du circuit (A. A. K. SERE et al., 2011 ; BOUFFARD et al., 2011).

1.5 Objectifs de la thèse et approche choisie

1.5.1 Objectifs

L'état de l'art décrit précédemment montre que de nouvelles attaques apparaissent régulièrement. Celles-ci peuvent être basées sur de nouveaux canaux auxiliaires, sur de nouvelles techniques d'exploitation ou encore sur une utilisation conjointe d'attaques existantes. Ces nouvelles attaques peuvent permettre de contourner des contre-mesures existantes, et de nouvelles contre-mesures doivent alors être proposées. L'anticipation des menaces par les concepteurs de circuits est donc rendue particulièrement difficile. De plus, la définition de contre-mesures nécessite une excellente connaissance de l'ensemble des attaques qui peuvent atteindre un circuit intégré. Certaines contre-mesures conçues spécifiquement pour un type d'attaque ont ainsi pu introduire des vulnérabilités face à d'autres attaques. Enfin, plusieurs années peuvent être nécessaires pour que la communauté scientifique, les centres d'évaluation ou les concepteurs de circuit puissent évaluer le degré d'efficacité pratique d'une contre-mesure. D'une manière générale, les protections doivent constamment être adaptées en fonction de l'évolution des attaques.

L'ajout de contre-mesures matérielles peut nécessiter de très lourds changements dans l'architecture d'un circuit et se révéler très coûteux en temps de développement, pour un résultat qui n'est parfois pas garanti si les contre-mesures en question n'ont pas fait l'objet de suffisamment de tests en pratique. A l'opposé, les contre-mesures au niveau du logiciel présentent l'avantage de pouvoir être rapidement déployées sur des architectures existantes à moindre coût. Elles présentent donc des avantages intéressants pour renforcer des systèmes embarqués face à des attaques contre lesquelles la définition de contre-mesures matérielles efficaces et à coût abordable prendra plusieurs années.

L'injection de fautes par ondes électromagnétiques est une technique récente contre laquelle un très large ensemble de systèmes embarqués actuels ne sont pas ou sont mal protégés. Les contre-mesures logicielles constituent par conséquent une solution de choix à court et moyen terme contre cette menace. L'objectif de cette thèse est donc de contribuer à la sécurisation de code embarqué face aux attaques par injection de fautes à l'aide d'impulsions électromagnétiques.

1.5.2 Approche choisie

Pour contribuer à renforcer un code embarqué face aux attaques par impulsion électromagnétique, nous avons choisi d'utiliser une approche en trois étapes : l'élaboration d'un modèle d'attaquant, la définition de contre-mesures adaptées à ce modèle, et enfin la vérification expérimentale de l'efficacité des contre-mesures. Ces trois étapes sont détaillées dans les paragraphes qui suivent.

1.5.2.1. Définition d'un modèle de fautes

Dans cette thèse, nous avons choisi d'élaborer un modèle de fautes au niveau « assembleur » par l'expérimentation en nous focalisant sur les effets d'une d'injection de fautes sur l'exécution d'un programme. Le niveau d'abstraction « assembleur » a l'avantage d'être à la fois proche du matériel, tout en permettant une bonne représentation des conséquences d'une attaque sur le programme embarqué. Ce travail d'élaboration du modèle est l'objet des chapitres 2 et 3.

1.5.2.2. Définition et validation théorique d'une contre-mesure

Une contre-mesure permettant de renforcer un code face aux fautes définies dans ce modèle peut alors être proposée. Nous proposons ensuite de garantir à l'aide d'outils de vérification formelle son efficacité théorique. Nous choisissons également de définir la contre-mesure de façon à ce qu'elle puisse être appliquée de façon automatique à un code générique pour en renforcer la résistance. Ce travail est l'objet du chapitre 4.

1.5.2.3. Vérification expérimentale de l'efficacité de la contre-mesure

L'évaluation de la contre-mesure à l'aide de méthodes formelles permet d'obtenir une première garantie quant à son efficacité théorique. Cette contre-mesure doit finalement être testée expérimentalement et ce travail est l'objet du chapitre 5.

1.6 Conclusion

Dans ce chapitre, une présentation concernant les différentes attaques et contre-mesures qui existent pour des circuits intégrés a été présentée. D'une part, les attaques par observation permettent de récupérer certaines informations sensibles d'un circuit en analysant une ou des grandeurs observables comme la consommation de courant ou les émissions de rayonnements électromagnétiques. Plusieurs contre-mesures visant à limiter les fuites d'informations via ces grandeurs observables ont été proposées dans la littérature scientifique. D'autre part, les attaques par

injection de fautes visent à modifier le comportement du circuit et différents moyens physiques comme le laser ou les perturbations transitoires du signal d'horloge peuvent être utilisés pour injecter des fautes dans un circuit intégré. Des contre-mesures peuvent être définies à plusieurs niveaux d'abstraction, du niveau physique au niveau algorithmique, et visent à renforcer le circuit face à ces attaques. A la fin du chapitre, l'objectif de la thèse et l'approche choisie ont été présentés. Le chapitre suivant traite de la mise en place d'un banc d'injection de fautes par impulsion électromagnétique pour un processeur embarqué.

2 Conception d'un banc d'injection de fautes pour processeur ARM Cortex-M3

*Certains des travaux présentés dans ce chapitre ont fait l'objet d'un article présenté lors de la conférence COSADE 2013 (*DEHBAOUI, MIRBAHA *et al., 2013).*

Sommaire

2.1 Introduction

Ce chapitre s'intéresse à la mise en place d'un banc d'injection de fautes pour réaliser des attaques sur un processeur ARM Cortex-M3. Ce banc est basé sur celui proposé dans (DEHBAOUI, J.-M. DUTERTRE, ROBISSON et TRIA, 2012) et en réutilise plusieurs éléments. Les premiers résultats obtenus dans cet article pour un processeur ATmega128 ont montré la possibilité de générer en pratique des fautes par injection électromagnétique sur un microcontrôleur lors de l'exécution d'un chiffrement AES. Les premiers résultats expérimentaux associés sont détaillés en annexe A. Toutefois,

dans le cadre de cette série d'expérimentations, aucune analyse détaillant la cause de l'apparition de ces fautes sur un octet lors de la dernière ronde du chiffrement n'avait été proposée. Les travaux présentés dans ce chapitre, et plus généralement dans cette thèse s'inscrivent donc dans la continuité de ces résultats préliminaires et réutilisent la même technique d'injection de fautes par impulsion électromagnétique.

Ce chapitre commence par présenter brièvement le processeur ARM Cortex-M3 utilisé dans cette thèse en 2.2. Cette présentation n'a pas pour but de présenter l'ensemble de l'architecture mais vise surtout à présenter les éléments principaux qui seront utilisés dans les chapitres suivants de cette thèse. Pour obtenir davantage de détails sur le processeur ARM Cortex-M3 et son architecture, le lecteur pourra consulter l'ouvrage (YIU, 2009). Ensuite, le banc d'injection et ses différents éléments constitutifs sont présentés en 2.3. Enfin, la fin du chapitre permet de vérifier le bon fonctionnement du banc mis en place et étudie la réalisation pratique d'attaques sur une implémentation de l'algorithme AES en 2.4.

2.2 Processeur ARM Cortex-M3

Pour les travaux présentés dans cette thèse, il nous a été nécessaire de choisir un type de microcontrôleur suffisamment représentatif des technologies actuellement utilisées. Les microcontrôleurs à base de processeur ARM Cortex sont utilisés à la fois pour des usages génériques ou pour la conception de circuits de sécurité par plusieurs fondeurs. Néanmoins, l'utilisation d'un microcontrôleur du commerce pour lequel nous n'avons pas accès à tous les détails sur l'architecture interne impose un certain nombre de contraintes au niveau de notre démarche visant à élaborer un modèle de fautes. Il est donc impératif de choisir un modèle pour lequel de bons outils de *debug* sont disponibles. Les processeurs ARM Cortex répondent donc bien au double besoin exprimé précédemment : à la fois largement répandus et possédant de très bons outils de *debug*. Notre choix s'est donc porté sur un microcontrôleur à base de processeur ARM Cortex-M3 pour la suite de cette thèse.

Le microcontrôleur ciblé possède un coeur ARM Cortex-M3 à architecture Harvard ARMv7-M et est fabriqué en technologie CMOS $180\ \mathrm{nm}$. Bien que nous n'ayons pas choisi de travailler sur une version pour carte à puce de ce microcontrôleur, la cible que nous utilisons embarque un certain nombre de mécanismes de sûreté permettant de filtrer la majeure partie des perturbations transitoires d'horloge ou de tension. De plus, plusieurs vecteurs d'interruption ont été définis de manière à pouvoir traiter certaines exceptions matérielles. Ces interruptions peuvent être utilisées dans le cadre d'une détection de fautes basique et seront détaillées ultérieurement. Enfin, la fréquence du microcontrôleur peut être réglée par programme : celui-ci peut être configuré de façon à pouvoir utiliser un oscillateur externe et une boucle à verrouillage de phase. La fréquence maximale acceptable par le microcontrôleur est de $72\ \mathrm{MHz}$. Pour la majorité des expérimentations de cette thèse, la fréquence du

microcontrôleur a été réglée à 56 MHz et pour quelques expérimentations celui-ci a fonctionné à 24 MHz [1]. A titre d'illustration, la figure 2.1 présente une image d'un modèle endommagé du microcontrôleur utilisé, celle-ci a été obtenue à l'aide d'un microscope à rayons X.

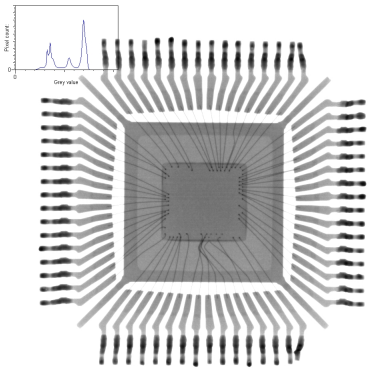


Figure 2.1. : Cartographie aux rayons X sur un modèle endommagé du microcontrôleur étudié

2.2.1 Jeu d'instructions

Le processeur ARM Cortex-M3 utilise le jeu d'instructions Thumb-2. Celui-ci est issu du rapprochement entre les jeux d'instructions Thumb (16 bits) et ARM (32 bits) : les instructions Thumb-2 peuvent donc être codées sur 16 ou 32 bits. Le jeu d'instructions Thumb-2 contient 151 instructions, qui elles-mêmes peuvent avoir jusqu'à 4 encodages binaires (selon les registres utilisés, la taille des opérandes, ...).

Convention d'écriture des instructions D'une manière générale, pour les mnémoniques des instructions exécutant des opérations arithmétiques et logiques, le registre de destination est placé en premier. Par exemple, l'instruction `add r1,r2,r3` additionne `r2` et `r3` et place le résultat dans `r1`. Lorsqu'un registre est à la fois source et destination, il est parfois possible de trouver une syntaxe simplifiée à deux registres. Par exemple, l'instruction `add r1,#1` ajoute `1` à la valeur contenue dans `r1` et stocke le résultat dans `r1`.

Architecture load/store Le processeur ARM Cortex-M3 est basé sur une architecture de type *load/store* : les opérations arithmétiques et logiques sont effectuées sur des registres, et tous les accès mémoire doivent passer par des instruction de type *load* (pour lire une donnée en mémoire et la placer dans un registre) ou *store* (pour écrire une donnée placée dans un registre en mémoire). Ces instructions de *load* ou *store*

1. Les différentes parties présentées dans cette thèse ne suivent pas un ordre chronologique. Cependant, une partie des expérimentations réalisées pendant la première année de ce travail de thèse ont été réalisées avec une fréquence de 24 MHz et sont présentées dans les chapitres 2 et 3.

sont utilisées à de nombreuses reprises dans cette thèse. Dans le jeu d'instructions Thumb-2, l'instruction `ldr` réalise une opération de *load* et l'instruction `str` une opération de *store*. Néanmoins, de nombreuses variantes de ces deux instructions existent dans le jeu Thumb-2.

Exécution conditionnelle et blocs `it` Le jeu d'instructions Thumb-2 fournit également un mécanisme d'exécution conditionnelle de code via les blocs `it` (*If-Then*). Une instruction `it` spécifie une condition et les 1 à 4 instructions qui la suivent peuvent être exécutées conditionnellement selon cette condition ou son inverse. Les blocs `it` permettent ainsi d'implémenter des structures de plus haut niveau de type *if-then* ou *if-then-else* et sont particulièrement utiles quand les blocs de code du *then* ou du *else* contiennent peu d'instructions. Un exemple de bloc `it` est présenté dans le listing 2.1. Dans cet exemple de code, l'instruction `cmp r0, #10` positionne les *flags* après avoir comparé `r0` par rapport à 10. L'instruction `itte` (*If-Then-Then-Else*) définit ensuite le mode d'exécution conditionnelle pour les trois instructions qui la suivent. Ensuite, si la condition NE est vérifiée (*i.e.* si le *flag* Z contient la valeur `1`), alors les deux instructions qui suivent (`addne` et `eorne`) sont exécutées. Sinon, l'instruction `moveq` est exécutée.

Listing 2.1 : Exemple de bloc `it`

```
1 cmp   r0 , #10        ; comparaison de r0 et 10
2 itte  NE              ; itte -> if then then else
3 addne r1 , r2 , #10   ; executee si NE est vraie (r0 != 10)
4 eorne r3 , r5 , r1    ; executee si NE est vraie (r0 != 10)
5 moveq r3 , #10        ; executee sinon (r0 == 10)
```

2.2.2 Registres

Le processeur ARM Cortex-M3 dispose d'un ensemble de registres sur 32 bits accessibles par le programmeur. Ces registres sont :

- 13 registres généraux (`r0`-`r12`)
- deux registres de pointeur de pile (accessibles via `r13`)
- un registre de pointeur de retour (`r14`)
- un registre de compteur de programme (`r15`)
- un registre d'état du programme (`xPSR`)
- un registre de niveau de privilège (`CONTROL`)
- des registres de masquage des interruptions (`PRIMASK`, `FAULTMASK`, `BASEPRI`)

Registres généraux Les 13 registres généraux sont numérotés de `r0` à `r12`. Les registres `r0` à `r7` sont utilisables avec l'ensemble des instructions (16 ou 32 bits) qui prennent pour argument un ou des registres généraux. En revanche, les registres `r8` à `r12` ne sont utilisables que dans des instructions 32 bits [2]. Toute instruction utilisant

2. Bien que quelques très rares exceptions existent pour lesquelles des instructions 16 bits peuvent manipuler des registres de `r8` à `r12`

un de ces registres sera donc automatiquement encodée sur 32 bits par le compilateur. Par ailleurs, au niveau de la convention qu'est l'Application Binary Interface (ABI) ARM, les registres `r0` à `r3` sont utilisés pour faire passer des arguments à des sous-fonctions. Le registre `r0` est également utilisé pour transférer le résultat d'une sous-fonction à la fonction appelante. Les registres `r4` à `r11` servent à manipuler les variables locales à l'intérieur d'une fonction. Enfin, le registre `r12` est utilisé pour réaliser de petits traitements sur quelques instructions à l'intérieur d'une procédure. Pour un code respectant la convention, ce registre n'a jamais besoin d'être sauvegardé sur la pile. Ce registre particulier est également nommé Intra-Procedure call scratch register (IP).

Pointeurs de pile (SP) Le processeur Cortex-M3 dispose de deux pointeurs de pile. Il est possible d'accéder à ces deux pointeurs via le registre `r13`. Néanmoins, un seul de ces deux registres est accessible à un instant donné. Ces deux registres sont le Main Stack Pointer (MSP) et le Process Stack Pointer (PSP). Le MSP est le pointeur de pile utilisé par défaut. Le PSP peut être utilisé si un système d'exploitation est installé, comme pointeur de pile pour du code en mode utilisateur (en dehors du noyau).

Pointeur de retour (LR) Le registre `r14` de pointeur de retour sert à stocker l'adresse de retour lors de l'appel d'une sous-fonction. Ainsi, l'instruction `bl` d'appel de sous-fonction met à jour le pointeur de retour et réalise un branchement vers le code de la sous-fonction. A la fin d'une sous-fonction, l'instruction `bx lr` est utilisée pour réaliser un branchement vers ce pointeur de retour.

Compteur de programme (PC) Le registre `r15` est le compteur de programme et contient l'adresse de l'instruction en cours d'exécution (décalée de 4 octets à cause de l'exécution pipelinée des instructions). Il est possible d'écrire directement dans ce registre, ce qui a pour effet de réaliser un branchement.

Registre d'état du programme (xPSR) Le registre `xPSR` contient des informations sur l'état interne du processeur et les exceptions qui ont été déclenchées. Il est constitué des 3 registres `APSR` (Application Program Status Register), `IPSR` (Interrupt Program Status Register) et `EPSR` (Execution Program Status Register). Ces 3 registres sont présentés dans le tableau 2.1.

Par ailleurs, les 5 *flags* du processeur peuvent être mis à jour par plusieurs instructions. Parmi celles-ci, on peut notamment citer les instructions de comparaison comme `cmp` ou `cmn`. Un grand nombre d'instructions arithmétiques et logiques disposent également d'un encodage pour lequel elles mettent à jour les *flags* selon la valeur du résultat. Au niveau du mnémonique de l'instruction, un suffixe `s` est utilisé pour utiliser cet encodage particulier. Ainsi, l'instruction `adds r1, r2, r3` exécute une addition entre `r2` et `r3`, stocke le résultat dans `r1` et met à jour les *flags* selon la valeur obtenue pour le résultat du calcul.

Table 2.1. : Registres PSR

APSR	5 *flags* (N, Z, C, V, Q) N - Négatif Z - Zéro C - Retenue V - Débordement Q - Saturation
IPSR	Si une interruption est en cours, ce registre contient un nombre servant à identifier l'interruption
EPSR	Informations sur l'utilisation des blocs d'instructions `IF-THEN`

Registres de masquage des interruptions Les trois registres de masquage des interruptions (`PRIMASK`, `FAULTMASK` et `BASEPRI`) permettent de désactiver certaines interruptions. Ces registres ne sont pas utilisés dans les travaux présentés dans cette thèse.

Registre de contrôle Le registre `CONTROL` est un registre sur 2 bits. Le bit de poids fort est utilisé pour définir lequel des deux pointeurs de pile est utilisé. Le bit de poids faible est utilisé pour définir si le processeur fonctionne en mode privilégié ou non. Ce dernier bit peut seulement être écrit si le processeur fonctionne en mode privilégié.

2.2.3 Modes d'exécution

Le Cortex-M3 supporte deux modes d'exécution (*Thread* et *Handler*) et deux niveaux de privilège (*Priviledged* et *User*). Le mode *Handler* est utilisé lors des déclenchements d'exceptions, autrement le processeur fonctionne en mode *Thread*. Le mode d'exécution *Handler* fonctionne uniquement avec un niveau *Priviledged*. Lors d'un redémarrage du microcontrôleur, et plus généralement si aucune instruction ne fait changer son mode d'exécution, celui-ci fonctionne en mode *Thread* et avec le niveau *Priviledged*. Comme précisé précédemment, le bit de poids faible du registre `CONTROL` sert à définir le niveau de privilège. Le fonctionnement au niveau *User* empêche l'exécution de certaines instructions spéciales, l'accès à certains registres de configuration du processeur et l'accès à certaines zones réservées de la mémoire. Pour des applications basiques, ces différents niveaux de privilège ne sont pas réellement nécessaires. Leur utilité principale est généralement liée aux systèmes d'exploitation qui ont besoin de séparer le code du noyau du code des applications utilisateur.

2.2.4 Exceptions matérielles

Plusieurs exceptions matérielles peuvent être déclenchées dans les cas d'accès mémoire interdits ou d'erreurs dans le traitement des instructions. Ces exceptions sont nommées `Hard Fault`, `Bus Fault`, `Usage Fault` et `Memory Management`

`Fault`. Chacune de ces exceptions peut être déclenchée pour différentes classes de fautes matérielles, et un descriptif plus détaillé de leurs conditions de déclenchement est proposé dans le tableau 2.2.

Table 2.2. : Conditions de déclenchement des différentes exceptions matérielles

Bus Fault	Se déclenche en cas d'erreur au niveau du bus, notamment lors de l'accès à une donnée ou à une instruction
Usage Fault	Se déclenche principalement en cas d'instruction invalide, d'opération interdite sur certains registres ou encore de division par zéro
Memory Management Fault	Se déclenche en cas d'accès à une zone mémoire interdite
Hard Fault	Toutes les classes d'erreur, se déclenche quand l'exception spécifique à la classe d'erreur ne peut pas être déclenchée

2.2.5 Pipeline et exécution des instructions

Le processeur Cortex-M3 est basé sur un pipeline à 3 niveaux. Ces trois niveaux sont *fetch*, *decode* et *execute*. Le tableau 2.3 présente les principales opérations réalisées pendant chacune de ces trois étapes.

Table 2.3. : Détail des différentes étapes du pipeline

Fetch	- Chargement de l'instruction dans le registre d'instruction
Decode	- Décodage de l'instruction - Chargement des opérandes de l'instruction - Détection des branchements
Execute	- Exécution de l'instruction - Écriture des résultats

L'étape de *fetch* charge 32 bits de données à chaque cycle d'horloge. Ces 32 bits provenant de la mémoire d'instructions peuvent correspondre à une seule instruction 32 bits, à deux instructions 16 bits ou peuvent également inclure 16 bits provenant du code binaire d'une instruction 32 bits. L'étape de *fetch* a également pour tâche de reconstituer les instructions 32 bits non-alignées (qui sont donc chargées en deux appels à la mémoire d'instructions). Néanmoins, si l'étape de *fetch* charge toujours ses données 32 bits par 32 bits, les étapes de *decode* et *execute* traitent toujours une seule instruction par cycle d'horloge, sur 16 ou 32 bits. De façon informelle, l'unité de *fetch* fonctionne donc plus rapidement en termes d'instructions traitées que les deux unités qui la suivent. Ainsi, l'unité de *fetch* est associée à une mémoire tampon permettant de stocker temporairement jusqu'à 3 mots de 32 bits. Dans le cas d'une instruction qui ne nécessite pas de récupérer la valeur d'une opérande, les étapes de *decode* et *execute* du pipeline ont chacune besoin d'au maximum un demi-cycle d'horloge pour s'exécuter. Sinon, pour les instructions de type `ldr` qui nécessitent la lecture d'une donnée depuis la mémoire, l'étape de *decode* requiert plus d'un cycle

d'horloge. La figure 2.2 montre les différentes étapes du pipeline lors de l'exécution d'une séquence d'instructions `nop` sur 16 bits.

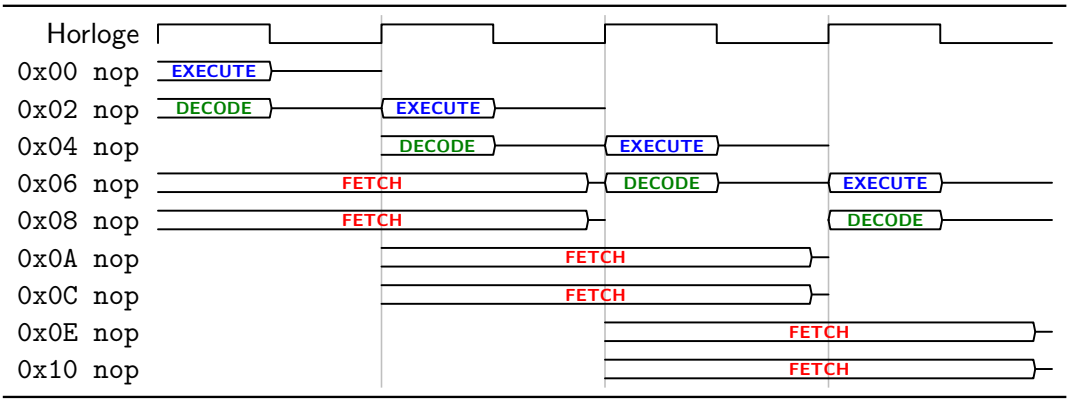


Figure 2.2. : Fonctionnement du pipeline pour une séquence d'instructions de type `nop` sur 16 bits

2.2.6 Mémoires d'instructions et de données

Le processeur Cortex-M3 est basé sur une architecture de type Harvard modifiée, pour laquelle la mémoire de données et la mémoire d'instructions sont physiquement séparées. La mémoire de données est placée en mémoire SRAM, et la mémoire d'instructions est placée par défaut en mémoire Flash (bien qu'il soit possible de placer des zones de mémoire d'instructions en mémoire SRAM).

2.2.7 Bus de données et d'instructions

En raison de son architecture Harvard modifiée, le processeur Cortex-M3 possède des bus séparés pour les mémoires d'instructions et de données. Ces bus utilisent la structure AMBA AHB-Lite (ARM, 2006). Chacun de ces deux bus est en réalité composé d'un bus de 32 bits pour l'envoi des adresses et d'un autre bus de 32 bits pour la réception des données ou instructions. Par ailleurs, selon les différents fabricants de microcontrôleurs basés sur le processeur ARM Cortex-M3, il est possible que ces bus soient préchargés à une certaine valeur au début de chaque cycle d'horloge. L'accès à une donnée ou une instruction sur l'un de ces bus requiert au minimum deux cycles. Pendant le premier cycle d'horloge, l'adresse de la donnée ou l'instruction à charger est envoyée sur le bus d'adresse associé (HADDR pour les données, HADDRI pour les instructions). Pendant le second cycle d'horloge, la donnée ou l'instruction à charger est alors envoyée depuis la mémoire sur le bus (HRDATA pour les données, HRDATAI pour les instructions). Ce fonctionnement des bus lors du chargement d'une donnée depuis la mémoire Flash est illustré sur la figure 2.3.

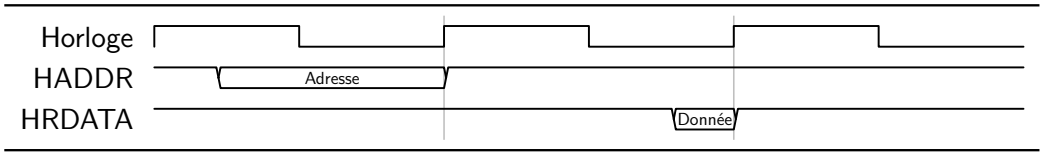


Figure 2.3. : Transferts sur les bus lors du chargement d'une donnée depuis la mémoire Flash

2.2.8 Préchargement d'instructions depuis la mémoire

Comme précisé précédemment, le microcontrôleur utilisé dans cette thèse peut fonctionner avec une fréquence allant jusqu'à 72 MHz. La mémoire Flash du microcontrôleur, qui est utilisée pour stocker les instructions, possède un temps de réponse de 35 ns. Ainsi, si la période d'horloge du microcontrôleur est inférieure à 35 ns (ce qui correspond à une fréquence d'environ 28 MHz), il devient impossible d'envoyer une donnée depuis la mémoire Flash sur le bus en un seul cycle d'horloge et un ou deux cycles supplémentaires deviennent nécessaires selon la fréquence du processeur. Pour éviter ainsi de sous-exploiter le processeur et pour maximiser sa vitesse de traitement des instructions, le fabricant du microcontrôleur utilisé dans cette thèse a mis en place une mémoire tampon (différente de celle présentée en 2.2.5) dont le but est de précharger des instructions depuis la mémoire Flash. Cette mémoire tampon se place donc avant le pipeline, et une fois activée celle-ci est transparente pour le programmeur. Il est en réalité composé de deux mémoires de 64 bits qui peuvent chacun lire directement 64 bits de mémoire Flash. Ces 64 bits peuvent donc correspondre au maximum à 4 instructions, qui seront chargées dans cette mémoire tampon. Pour garder un certain contrôle sur le chargement des instructions dans le pipeline et pouvoir clairement définir l'intervalle de temps pendant lequel une instruction précise est chargée sur le bus, cette mémoire tampon a été désactivée pour l'ensemble des expérimentations de cette thèse.

2.2.9 Chaîne de compilation

A l'exception de certains programmes présentés dans le chapitre 5 (qui ont été développés à l'aide de la suite IAR Embedded WorkBench), l'ensemble des programmes présentés dans cette thèse ont été développés à l'aide de la suite MDK-ARM de Keil. Cette suite regroupe les différents outils de compilation ARM (compilateur C `armcc`, bibliothèque C MicroLib, linker `armLink`, assembleur `armasm`) et l'environnement de développement µVision. Les programmes de cette thèse ont été compilés sans optimisations (`-O0`).

Enfin, deux points particuliers liés à la manière avec laquelle les instructions sont encodées par l'assembleur `armasm` sont présentés dans les paragraphes qui suivent et sont utilisés à plusieurs reprises dans cette thèse.

2.2.9.1. Contrôle de l'encodage des instructions

Afin de forcer l'assembleur à utiliser un encodage sur 32 bits pour des instructions, il est possible d'utiliser un suffixe `.w` après le mnémonique correspondant à l'instruction (par exemple `add.w`) ou bien d'utiliser les registres `r8` à `r14`. De manière similaire, un suffixe `.n` existe pour forcer un encodage sur 16 bits.

2.2.9.2. Pseudo-instruction pour le chargement de constantes sur 32 bits

Les instructions du jeu Thumb-2 sont encodées sur au maximum 32 bits. Cette taille d'instructions pose un problème pour le chargement de constantes sur 32 bits dans un registre, car des instructions comme `mov` (ou `mvn`) peuvent au maximum charger une données sur 16 bits dans un registre. En utilisant des instructions comme `mov` (ou `mvn`), il serait donc impossible de charger une constante sur 32 bits en une seule instruction.

Pour éviter au programmeur (lors de l'écriture d'un code en assembleur) d'avoir à gérer manuellement ce cas des chargements de constantes sur 32 bits en les décomposant en plusieurs instructions, une pseudo-instruction `ldr Rd,=constante` de chargement a été mise en place au niveau de l'assembleur `armasm`. Si la constante peut être chargée depuis une instruction `mov` (ou `mvn`), alors la pseudo-instruction `ldr Rd,=constante` est transformée en instruction `mov` (ou `mvn`). Sinon, cette constante est placée en mémoire d'instructions, dans une plage mémoire qui ne sera pas exécutée [3]. Ensuite, une instruction `ldr` dont l'adresse de chargement est définie par rapport à la valeur du compteur de programme est utilisée.

Par exemple, la pseudo-instruction `ldr r0,=0x12345678` vise ainsi à charger la valeur `0x12345678` dans le registre `r0`. Pour cela, la valeur `0x12345678` est placée en mémoire d'instructions, avec un décalage qu'on suppose de 10 octets pour cet exemple entre l'adresse à laquelle est placée la constante et l'adresse de l'instruction qui doit la charger. Ensuite, la pseudo-instruction `ldr r0,=0x12345678` est donc convertie en une instruction `ldr r0,[pc,#10]` qui charge une donnée avec un décalage de 10 octets par rapport à l'adresse du compteur de programme.

Ce type de pseudo-instructions est utilisé à de nombreuses reprises dans cette thèse pour générer des instructions de `ldr` depuis la mémoire Flash, notamment dans les expérimentations qui sont menées dans les chapitres 3 et 5.

3. Généralement, cette plage mémoire est située juste après l'instruction de retour de la sous-fonction dans laquelle cette pseudo-instruction est utilisée

2.3 Dispositif expérimental d'injection de fautes

Cette section commence par présenter le montage expérimental utilisé dans cette thèse en 2.3.1, puis détaille les différentes étapes du processus expérimental utilisé dans cette thèse pour réaliser des attaques par injection de fautes en 2.3.2.

2.3.1 Montage expérimental d'injection

Le montage expérimental d'injection choisi permet de réaliser des injections de fautes à l'aide d'impulsions électromagnétiques. Il s'inscrit dans la continuité de celui proposé dans (Dehbaoui, J.-M. Dutertre, Robisson et Tria, 2012) pour un processeur ATmega128 en utilisant le même générateur d'impulsions ainsi que la même antenne d'injection. En revanche, le dispositif expérimental présenté dans cette thèse a été dès le départ conçu de façon à pouvoir être utilisé pour un microcontrôleur à coeur ARM Cortex-M3 et surtout de façon à pouvoir observer davantage de données sur l'état interne du circuit.

Les éléments qui constituent le montage expérimental sont :

- un générateur d'impulsions
- une antenne d'injection sous forme de solénoïde
- un microcontrôleur lié à une sonde Joint Test Action Group (JTAG)
- une table X-Y-Z motorisée
- un ordinateur pour le pilotage du banc
- une alimentation stabilisée

Un schéma du montage expérimental est présenté sur la figure 2.4.

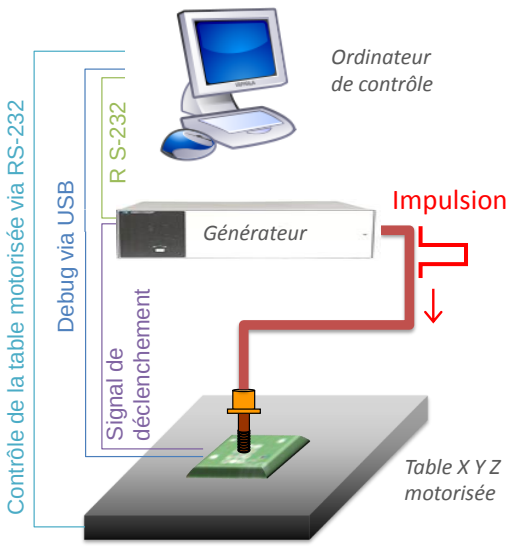


Figure 2.4. : Schéma du banc d'injection

2.3.1.1. Générateur d'impulsions

Le générateur est utilisé pour envoyer des impulsions de tension à travers l'antenne d'injection. Il dispose d'un temps de montée et de descente constant de 2 ns. Il génère des impulsions dont la tension peut aller de -210 V à 210 V et peut délivrer de forts courants allant jusqu'à 4 A. La durée des impulsions peut varier de 10 ns à 200 ns. Par ailleurs, le générateur peut envoyer des séries d'impulsions à une fréquence maximale de 20 kHz et requiert donc un temps de 50 µs entre deux générations d'impulsions (ce qui correspond approximativement à 2800 cycles d'horloge pour une fréquence de 56 MHz). L'antenne magnétique utilisée est un solénoïde de 1 mm de diamètre et qui possède une impédance d'entrée de 50 Ω. La figure 2.5 présente l'allure générale des impulsions reçues par le circuit. Cette mesure a été réalisée à l'aide d'une antenne sous forme de spire placée sous l'antenne d'injection. Le graphe de gauche présente l'allure obtenue pour une impulsion positive, et celui de droite l'allure obtenue pour une impulsion négative.

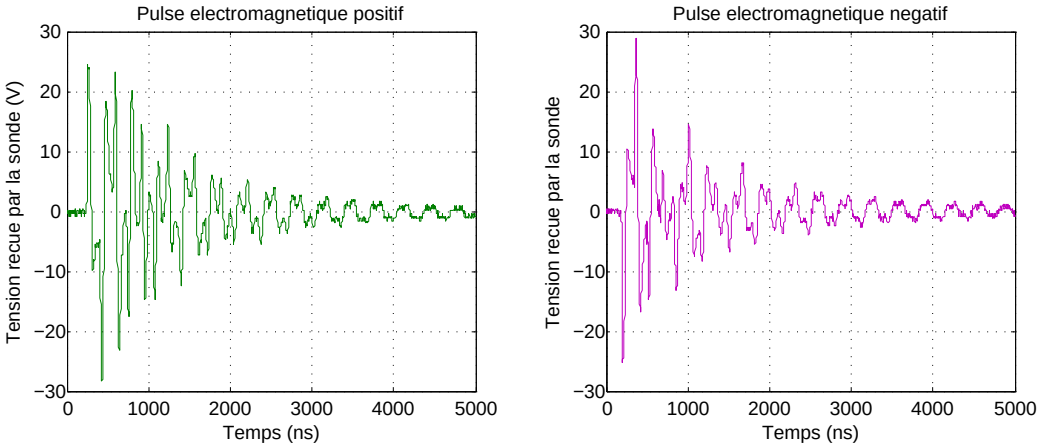


Figure 2.5. : Résultats de mesure depuis une antenne sous forme de spire une impulsion électromagnétique positive (gauche) et négative (droite)

2.3.1.2. Antenne d'injection

L'antenne utilisée pour l'injection de fautes est un solénoïde en cuivre, de 1 mm de diamètre, d'environ 15 spires et possédant un noyau de ferrite. La figure 2.6 montre l'antenne d'injection électromagnétique utilisée pour cette thèse au dessus d'un microcontrôleur.

2.3.1.3. Microcontrôleur et sonde JTAG

Comme précisé au début de ce chapitre, le microcontrôleur utilisé est basé sur le processeur ARM Cortex-M3. Celui-ci est relié à une sonde JTAG qui permet d'obtenir des informations sur l'état interne du microcontrôleur. Pour les travaux présentés

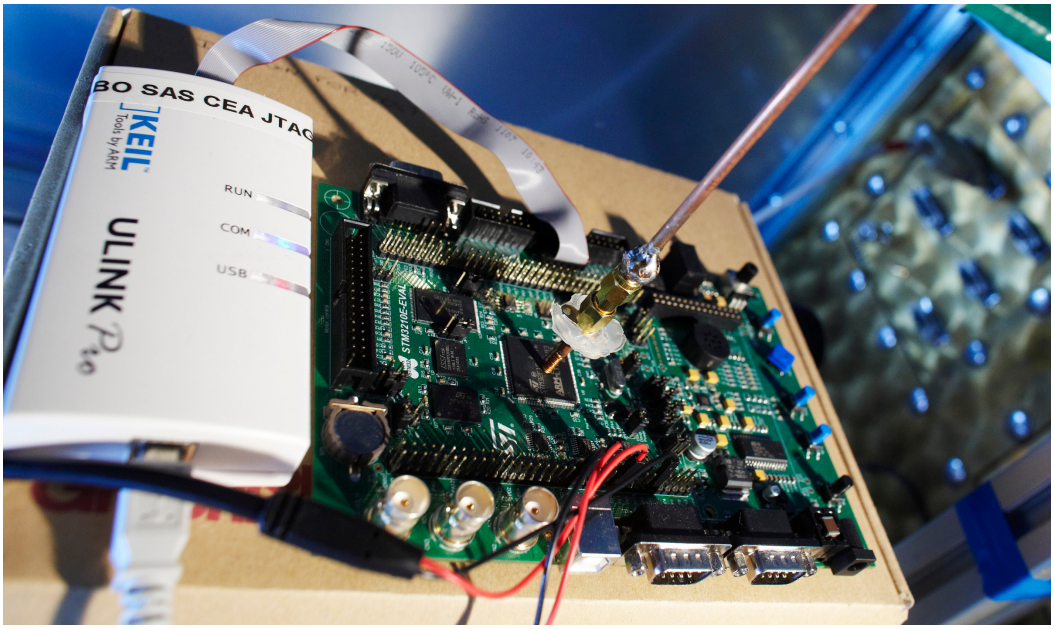


Figure 2.6. : Antenne d'injection électromagnétique positionnée au-dessus d'un microcontrôleur

dans cette thèse, une sonde JTAG Keil ULINKpro a été utilisée. Celle-ci peut être observée sur la figure 2.6.

2.3.1.4. Table motorisée

Le microcontrôleur est posé sur une table X Y Z motorisée de haute précision. Celle-ci sert à faire varier la position relative de l'antenne d'injection par rapport au circuit. Cette table est donc utilisée afin de positionner l'antenne sur différentes portions du circuit.

2.3.1.5. Ordinateur pour le pilotage du banc

Un ordinateur sert au pilotage des différents éléments du banc. Celui-ci est relié à la table motorisée et au générateur d'impulsions par des liaisons RS-232, ainsi qu'à la sonde JTAG par une liaison USB. Cet ordinateur permet de gérer le déroulement du processus expérimental d'injection, qui est présenté dans la section suivante.

2.3.2 Processus expérimental

Un processus expérimental spécifique a été conçu de façon à pouvoir récupérer un maximum d'informations sur l'état interne du circuit à la suite d'une impulsion électromagnétique. Pour cela, un programme de pilotage des différents éléments constitutifs du banc d'injection a été réalisé. Celui-ci a été écrit en C et utilise la

bibliothèque UVSOCK[4]. Cette bibliothèque s'interface avec le debugger de Keil µVision et permet de le piloter. Ce programme a été utilisé pour l'ensemble des expérimentations de cette thèse, y compris celles pour lesquelles le code n'a pas été développé avec Keil µVision.

Le processus expérimental utilisé est le suivant :

— Redémarrer le microcontrôleur

— Exécuter le code visé

— Envoyer une impulsion de tension à travers l'antenne d'injection

— Interrompre l'exécution du programme

— Récupérer les données internes du microcontrôleur

Les paragraphes qui suivent détaillent les éléments importants qui constituent ce procédé expérimental.

2.3.2.1. Mécanisme de déclenchement d'une impulsion

Pour contrôler le déclenchement d'une impulsion de tension par le générateur, nous avons choisi de faire émettre un signal de déclenchement par le microcontrôleur. Lors de la réception de ce signal de déclenchement par le générateur d'impulsions, un délai de $100\ \mathrm{ns}$ s'écoule avant l'envoi de l'impulsion par le générateur. Même si ce signal n'est pas représentatif d'une situation réelle, il est utilisé dans cette thèse car il permet de faciliter grandement la démarche expérimentale permettant de comprendre les effets de l'injection par impulsion électromagnétique.

2.3.2.2. Point d'observation et arrêt du programme

Pour l'ensemble des expérimentations présentées dans cette thèse, il est nécessaire de définir un point d'arrêt pour le programme afin de récupérer l'état interne du processeur. Comme le code visé est encapsulé dans une fonction, ce point d'observation est défini avant la restauration du contexte à la fin de la fonction en question. Toutefois, à la suite d'injections de fautes, le programme visé n'atteint pas toujours ce point d'observation. Par exemple, certaines attaques peuvent aboutir à des déclenchements d'exceptions matérielles ou encore à l'exécution de branchements inconditionnels vers une autre partie du code. Pour ces cas particuliers, les routines appelées lors du déclenchement d'exceptions matérielles exécutent une boucle infinie et un délai fixe de $100\ \mathrm{ms}$ a été mis en place. L'ordinateur de contrôle arrête le microcontrôleur si le programme n'a pas atteint le point d'observation au bout de ce délai.

4. http://www.keil.com/appnotes/docs/apnt_198.asp

2.3.2.3. Intervalle de temps à balayer

Pour pouvoir réellement viser un cycle d'horloge précis, les exceptions matérielles du Cortex-M3 sont utilisées. Celles-ci permettent d'obtenir des informations sur l'instruction qui a déclenché l'exception. Cette information est utilisée dans une étape de calibration préliminaire pour définir l'intervalle de temps à balayer afin de perturber une instruction ou une portion de code. Par exemple, lors du déclenchement d'une exception matérielle de type `UsageFault` pour un code d'instruction invalide, l'adresse de l'instruction pour laquelle l'exception a été déclenchée est placée sur la pile. En récupérant cette information, il est donc possible d'associer une instruction atteinte à une valeur de l'instant d'injection.

2.3.2.4. Données internes

Grâce à la sonde JTAG utilisée pour le *debug*, les données qu'il est possible d'obtenir depuis le microcontrôleur sont :

- les 13 registres généraux `r0` à `r12`
- le *Stack Pointer* `sp` (`r13`, pointeur de pile)
- le *Link Register* `lr` (`r14`, pointeur de retour de fonction)
- le *Program Counter* `pc` (`r15`, compteur de programme)
- les registres de *Program Status Register* PSR (`xPSR`)
- certaines variables choisies en mémoire
- le nombre de cycles d'horloge qui ont été nécessaires pour exécuter le bout de code cible

2.3.3 Bilan sur le dispositif expérimental utilisé

Les paragraphes de cette section ont présenté le dispositif expérimental utilisé dans cette thèse. Celui-ci s'appuie sur le montage proposé dans (Dehbaoui, J.-M. Dutertre, Robisson et Tria, 2012) pour ATmega128 mais a été conçu de façon différente, en utilisant un programme spécifique de contrôle pour une récupération de l'état interne du processeur et une analyse des effets de la technique d'injection. La section suivante présente deux attaques sur des implémentations de l'algorithme AES en utilisant le banc d'injection présenté dans ce chapitre. Toutefois, ces attaques n'exploiteront pas l'ensemble des capacités du dispositif expérimental présenté et visent surtout à en montrer le bon fonctionnement.

2.4 Expérimentations sur une implémentation de l'algorithme AES

Pour illustrer les capacités du montage expérimental à réaliser des attaques par injection de fautes, la suite de ce chapitre présente deux exemples d'attaques sur une implémentation de l'algorithme AES. Cette section commence donc par présenter plus en détail l'algorithme AES en 2.4.1. Ensuite, la partie 2.4.2 présente un exemple d'attaque pour laquelle le compteur du nombre de rondes de l'algorithme a été attaqué et une ronde supplémentaire a ainsi été ajoutée. Enfin, la partie 2.4.3 présente un exemple d'attaque utilisant une double faute sur la fonction d'addition de clé de ronde au cours de la dernière ronde d'une implémentation de l'algorithme AES.

2.4.1 Advanced Encryption Standard (AES)

L'algorithme AES est le standard actuel de chiffrement symétrique sélectionné à la suite du concours lancé par le National Institute of Standards and Technology (NIST) en 2001. Ce concours visait à sélectionner un successeur pour l'algorithme standard de chiffrement symétrique DES, dont la conception remontait à 1976 (NIST, 1977). Le standard emploie l'algorithme Rijndael, conçu par Vincent Rijmen et Joan Daemen (NIST, 2001). Rijndael est un algorithme de chiffrement symétrique et traite des blocs de 128 bits. Dans sa version standardisée, il peut travailler avec trois tailles de clés : 128, 192 et 256 bits, exécutant ainsi respectivement 10, 12 ou 14 rondes. Il se distingue notamment du DES par le fait qu'il soit basé sur un réseau de substitution-permutation (contre une construction basée sur un schéma de Feistel pour le DES) et par la taille des clés utilisées [5].

L'AES regroupe deux processus séparés :

— `KeyExpansion`, qui calcule les clés pour chacune des rondes à partir de la clé initiale

— `DataEncryption`, qui chiffre le texte clair à l'aide de chacune des clés de ronde

La figure 2.7 présente un schéma du processus de `DataEncryption` de l'algorithme. La partie gauche de la figure présente l'ordre des différentes étapes du processus, et la partie droite présente le schéma d'une implémentation matérielle possible pour ce processus. Lors d'une opération de chiffrement sur un système embarqué, le calcul des clés de ronde à l'aide de l'algorithme de `KeyExpansion` peut être réalisé avant l'opération de `DataEncryption` en précalculant les clés pour chacune des rondes ou être réparti dans chacune des rondes, la clé de ronde étant alors calculée au début de la ronde.

La taille des clés de l'algorithme AES le met à l'abri de toute attaque par recherche exhaustive sur l'ensemble des clés (Lenstra et Verheul, 2001). L'algorithme a

5. DES utilisait des clés de 56 bits

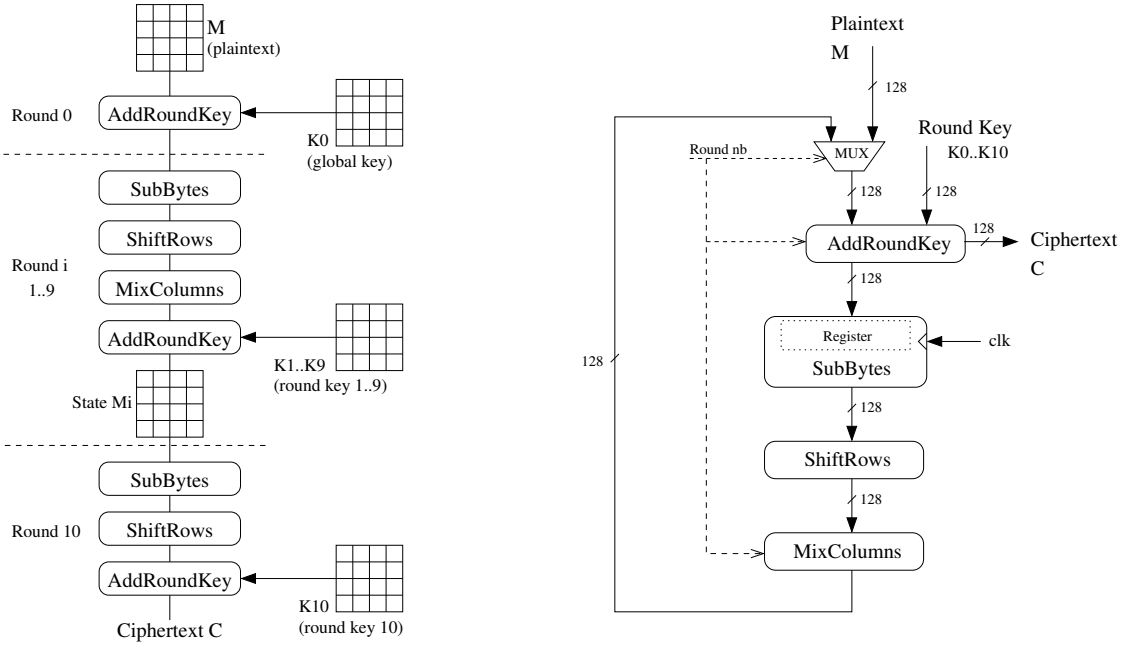


Figure 2.7. : Processus `DataEncryption` de l'algorithme AES-128

également été conçu de façon à résister aux attaques algébriques connues au moment de sa conception et s'est pour l'instant montré résistant face à celles qui sont apparues ces dix dernières années. De plus, la possibilité de l'implémenter sur des circuits embarqués faisait partie des spécifications de base de l'algorithme. Ainsi, il est utilisé dans de nombreux systèmes embarqués, que ce soit sous forme de code exécuté par un microcontrôleur, sous forme de coprocesseur cryptographique ou bien en utilisant les jeux d'instructions spéciaux proposés sur certains processeurs.

2.4.2 Attaque sur l'incrémentation du compteur de ronde

Cette attaque se base sur une modification du compteur de rondes de l'algorithme de façon à rajouter une ronde au chiffrement. Ce principe de modification du compteur de rondes a été introduit par Choukri et Tunstall qui ont réduit un chiffrement AES à une seule ronde (CHOUKRI et TUNSTALL, 2005). Plus récemment, Dutertre *et al.* (J.-M. DUTERTRE, MIRBAHA et al., 2012) ont également étudié les conséquences de l'ajout d'une ronde de chiffrement.

2.4.2.1. Présentation de l'attaque

Dans l'implémentation considérée, les clés de rondes ne sont pas pré-calculées (étape de `KeyExpansion`) avant l'exécution du chiffrement (étape de `DataEncryption`). Le calcul de chaque clé de ronde par la fonction `computeKey` a lieu pendant l'étape de chiffrement, juste avant l'appel à la fonction `addRoundKey`. Une telle structure est illustrée dans le listing 2.2.

Listing 2.2 : Structure générale de l'implémentation AES considérée

```
1  int i;
2
3  addRoundKey();                    // OU EXCLUSIF avec la clé de chiffrement
4
5  for(i = 0; i < 9; i++)            // 9 rondes
6  {
7      subBytes();
8      shiftRows();
9      mixColumns();
10     computeKey(rcon[i]);          // calcul de la clé de ronde
11     addRoundKey();
12 }
13
14 // dernière ronde
15 subBytes();
16 shiftRows();
17 computeKey(rcon[i]);
18 addRoundKey();
```

Le listing 2.3 présente une partie du code assembleur issu de la compilation du code précédent. Dans cette suite d'instructions assembleur, on constate qu'à la fin de chaque ronde le compteur de boucle est comparé avec la valeur 9 et qu'en fonction du résultat de cette comparaison le programme boucle sur le début d'une ronde supplémentaire ou bien continue pour exécuter la dixième ronde. Il est à noter que le compteur de boucle est décalé d'un indice par rapport au numéro de ronde en cours d'exécution : il varie de 0 à 9 pour compter les rondes 1 à 10. L'incrémentation du compteur de boucle, ici stocké dans le registre `r4`, est réalisée à la fin de chaque ronde via l'instruction `adds r4, r4, #1` (à la ligne 14).

Listing 2.3 : Code compilé pour l'implémentation d'AES considérée

```
1  premiere_ronde
2  bl      addRoundKey       ; branchement vers la fonction addRoundKey
3  movs    r4, #0            ; réinitialisation du registre r4
4  b       comp_compteur     ; branchement vers la comparaison du compteur
5
6  debut_ronde
7  bl      subBytes          ; branchement vers la fonction subBytes
8  bl      shiftRows         ; branchement vers la fonction shiftRows
9  bl      mixColumns        ; branchement vers la fonction mixColumns
10 ldr     r1, =ad_rcon      ; chargement de l'adresse de rcon dans r1
11 ldrb    r0, [r1, r4]      ; chargement d'un octet de rcon dans r0
12 bl      computeKey        ; calcul de la clé de ronde en utilisant rcon
13 bl      addRoundKey       ; branchement vers la fonction addRoundKey
14 adds    r4, r4, #1        ; incrémentation du compteur de ronde
15
16 comp_compteur
17 cmp     r4, #9            ; si compteur de ronde inférieur ou egal a 9
18 blt     debut_ronde       ; alors branchement au début de la ronde
```

```
19
20 derniere_ronde
21 bl     subBytes          ; branchement vers la fonction subBytes
22 bl     shiftRows         ; branchement vers la fonction shiftRows
23 ldr    r1, =ad_rcon      ; chargement de l'adresse de rcon dans r1
24 ldrb   r0, [r1, r4]      ; chargement d'un octet de rcon dans r0
25 bl     computeKey        ; calcul de la clé de ronde en utilisant rcon
26 bl     addRoundKey       ; branchement vers la fonction addRoundKey
```

L'attaque présentée dans cette partie vise l'instruction `adds r4,r4,#1` pour perturber l'incrémentation du compteur de boucle en fin de neuvième itération. Si cette instruction n'est pas exécutée, alors le registre `r4` contient toujours la valeur 8 au moment de l'instruction `cmp r4,#9`. Cette instruction compare la valeur dans `r4` à la valeur 9 et positionne les *flags* en fonction du résultat. Quand `r4` vaut `8`, elle met donc le *flag* `N` à 1 et laisse les autres *flags* à 0. Ensuite, l'instruction `blt debut_ronde` réalise un branchement vers le début d'une ronde comme le flag `N` est à 1. Une nouvelle ronde est donc exécutée, mais avec la particularité d'avoir la même valeur pour le compteur de ronde que la ronde précédente. Par ailleurs, cette valeur est utilisée pour le calcul de la clé de ronde par la fonction `computeKey`. Ainsi, l'algorithme exécute finalement 11 rondes dont 2 rondes avec la même valeur de compteur. Une cryptanalyse associée a ensuite été élaborée. Celle-ci permet de retrouver la clé de chiffrement à l'aide de deux couples associant chacun un chiffré et son chiffré fauté. Elle utilise ces deux paires de chiffrés pour remonter les étapes de chiffrement et retrouver la clé de la dixième ronde, et nécessite une recherche exhaustive sur 2^{16} hypothèses pour cette clé de la dixième ronde. Comme l'objet de cette section est avant tout la réalisation d'attaques permettant cette cryptanalyse, la cryptanalyse en elle-même ne sera pas davantage détaillée et le lecteur pourra consulter (DEHBAOUI, MIRBAHA et al., 2013) pour obtenir une description plus précise de celle-ci.

2.4.2.2. Résultats expérimentaux

Pour cette expérimentation, la fréquence d'horloge du processeur a été fixée à 24 MHz. Pour perturber l'instruction qui incrémente le compteur de ronde, l'instant d'injection de l'impulsion électromagnétique a varié sur un intervalle de 500 ns par pas de 100 ps. Pour chacun de ces instants d'injection, 100 chiffrements ont été réalisés. De plus, pour chaque chiffrement, les valeurs correspondant au chiffré non-fauté et au chiffré pour lequel une ronde a été ajoutée ont été calculées par l'ordinateur de contrôle en parallèle du processus d'injection. Ces valeurs permettent de détecter les sorties fautées pour lesquelles une itération supplémentaire de la boucle de chiffrement a été réalisée. Enfin, le microcontrôleur a été automatiquement redémarré entre chaque tentative d'injection de faute.

Sur l'intervalle de temps parcouru, de nombreuses exceptions matérielles de type `Bus Fault` ont été déclenchées à la suite d'une faute. Pour certains autres instants d'injection, le microcontrôleur a été victime d'un *crash*. Néanmoins, sur un intervalle de temps d'environ 1 ns, près de 100% des tentatives d'injection de faute ont amené à l'exécution d'une ronde supplémentaire sans détection par le microcontrôleur.

Ainsi, il a été possible à l'aide du montage expérimental décrit au début de ce chapitre de fauter une instruction d'incrémentation du compteur de ronde, et de pouvoir retrouver une clé de chiffrement AES. La technique d'injection a montré son efficacité dans un cas concret même si des détails sur les effets précis induits par l'injection manquent encore (ils seront étudiés en détail dans le chapitre 3).

2.4.3 Attaque sur la fonction d'addition de clé de ronde

Cette attaque cible une implémentation de l'algorithme AES qui a été renforcée à l'aide d'une contre-mesure basique de redondance temporelle face aux attaques par injection de fautes. La contre-mesure en question suit le principe de duplication présenté dans (BAR-EL et al., 2006) et consiste à réaliser deux chiffrements AES et à en comparer les résultats de sortie. Pour cette attaque, l'implémentation de l'algorithme AES est la même que celle présentée en 2.4.2.

2.4.3.1. Présentation de l'attaque

L'attaque vise à perturber l'opération `AddRoundKey` de la dernière ronde de chiffrement. Dans l'implémentation considérée, qui n'est pas optimisée pour un processeur 32 bits et qui se contente de réaliser l'opération octet par octet, les 16 octets sont traités les uns à la suite des autres. Une faute qui empêche l'opération de OU EXCLUSIF (XOR) entre un octet de clé et un octet de texte permet théoriquement d'obtenir l'octet de clé via une cryptanalyse quasi-immédiate. Cette cryptanalyse consiste à réaliser le XOR entre l'octet fauté obtenu et l'octet chiffré sans faute. On note C le chiffré, C' le chiffré fauté obtenu (pour lequel la dernière étape d'`AddRoundKey` n'a pas été effectuée sur l'octet i), M le texte à chiffrer avant l'opération d'`AddRoundKey` et K^{10} la sous-clé utilisée pour la 10ème ronde.

On a donc :

$$\begin{aligned} C_i &= M_i \oplus K_i^{10} \\ C'_i &= M_i \end{aligned} \tag{2.1}$$

D'où :

$$K_i^{10} = C_i \oplus C'_i \tag{2.2}$$

Le listing 2.4 présente le code source C de la fonction `AddRoundKey` pour l'implémentation utilisée.

Listing 2.4 : Code source C pour la fonction `AddRoundKey`

```
1  void addRoundKey(byte* state, byte* key)
2  {
3      int i;
4
5      for(i=0; i < 16; i++)
6      {
7          state[i] ^= key[i];
8      }
9  }
```

Ce code a ensuite été compilé et le code assembleur obtenu est présenté dans le listing 2.5. La séquence composée des deux instructions `ldrb`[6], de l'instruction `eors`[7] et de l'instruction `strb`[8] réalise l'opération d'addition de clé de ronde sur un octet. Par ailleurs, au niveau du lien entre les variables définies au niveau du code source et les registres du code compilé, on a `r3` = state[i] et `r4` = key[i].

Listing 2.5 : Code compilé correspondant à la fonction `AddRoundKey`

```
1  push    {r4, lr}        ; sauvegarde de registres sur la pile
2  movs    r2, #0          ; initialisation du compteur de boucle
3
4  debut_boucle
5  ldrb    r3, [r0, r2]    ; r3 = state[i]
6  ldrb    r4, [r1, r2]    ; r4 = key[i]
7  eors    r3, r3, r4      ; r3 = r3 xor r4
8  strb    r3, [r0, r2]    ; stockage du résultat en mémoire
9  adds    r2, r2, #1      ; r2 = r2 + 1
10 cmp     r2, #16         ; comparaison entre le compteur de boucle et 16
11 blt     debut_boucle    ; si r2 < 16, branche vers le début de la boucle
12 pop     {r4, pc}        ; retour vers la fonction appelante
```

Instructions `ldrb` Si la première instruction `ldrb` n'est pas exécutée, le registre `r3` garde sa valeur précédente, qui peut soit correspondre à celle utilisée pour l'octet précédent, soit s'il s'agit du premier octet à la valeur qu'avait le registre avant l'appel à la fonction `AddRoundKey`. Les sorties fautées générées permettent potentiellement d'obtenir des informations sur la clé mais pas de réaliser la cryptanalyse présentée précédemment. De même, si la deuxième instruction `ldrb` n'est pas exécutée, la situation est la même pour le registre `r4` : il est donc possible d'obtenir deux octets chiffrés avec le même octet de clé, ce qui constitue là encore une fuite potentielle d'information sur la clé mais ne permet pas la cryptanalyse présentée précédemment.

6. L'instruction `ldrb` charge un octet depuis une adresse mémoire dans un registre (son fonctionnement est similaire à celui de l'instruction `ldr` mais `ldrb` charge 8 bits au lieu d'un mot de 32 bits pour `ldr`)
7. L'instruction `eors` réalise une opération de OU EXCLUSIF entre deux registres, stocke le résultat dans un troisième registre et met à jour les *flags* du processeur selon la valeur du résultat
8. L'instruction `strb` stocke un octet situé dans un registre à une adresse mémoire

Instruction `eors` ou instruction `strb` Si l'instruction `eors` n'est pas exécutée, l'opération de XOR n'est pas effectuée. De même, si l'instruction `strb` n'est pas effectuée, l'opération de XOR est réalisée mais son résultat n'est pas stocké en mémoire. La perturbation de l'une de ces deux instructions peut donc mener à un résultat fauté qui permet la cryptanalyse présentée précédemment. L'attaque vise donc à corrompre l'exécution d'au moins une de ces deux instructions.

2.4.3.2. Résultats expérimentaux

Comme précisé en 2.3.1.1, le générateur utilisé requiert un temps de $50\ \mu s$ entre deux envois d'impulsions. Or, l'exécution complète de l'implémentation AES sur ce microcontrôleur est plus longue que $50\ \mu s$. Pour réaliser une attaque sur cette implémentation où la fonction de chiffrement est exécutée deux fois, il est possible d'injecter une faute au même instant pour les deux exécutions du chiffrement. Dans le cas considéré, l'intervalle correspondant au traitement du premier octet a été choisi comme cible. Cet intervalle correspond à une plage de temps de $70\ ns$. Cet intervalle a été parcouru par pas de $200\ ps$, le même décalage de l'instant d'injection étant appliqué pour la deuxième exécution de l'algorithme. Le processus complet d'injection sur les deux chiffrements successifs a été répété deux fois pour chaque instant d'injection. Au total, 700 tentatives d'attaques ont été réalisées (pour 1400 impulsions envoyées) pour chaque valeur de tension considérée. En l'occurrence, des valeurs de tension pour l'impulsion allant de $-210\ V$ à $-170\ V$ (par pas de $5\ V$) ont été utilisées. Pour cette expérimentation, le texte clair, la clé, le texte chiffré attendu et le texte chiffré fauté attendu en corrompant l'exécution de l'instruction `eors` ou `strb` sont présentés dans le tableau 2.4.

Table 2.4. : Clé, texte clair et chiffrés attendus pour cette expérimentation

Clé de chiffrement	2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Texte clair	32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Texte chiffré	39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Texte chiffré fauté	e9 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32

La figure 2.8 présente le nombre de fautes détectées (ce qui signifie que les deux chiffrements ont donné des résultats différents pour le premier octet) et non-détectées (ce qui signifie que la même faute a été injectée sur les deux chiffrements). Il est donc possible de constater que, dans les conditions expérimentales considérées, des doubles fautes peuvent parfaitement être réalisées avec un important taux de succès. Par exemple, 117 attaques sur 700 ont été couronnées de succès pour une tension d'impulsion de $-210\ V$.

Néanmoins, toutes ces sorties fautées non-détectées ne correspondent pas nécessairement à des valeurs exploitables pour une cryptanalyse. Pour une tension d'impulsion

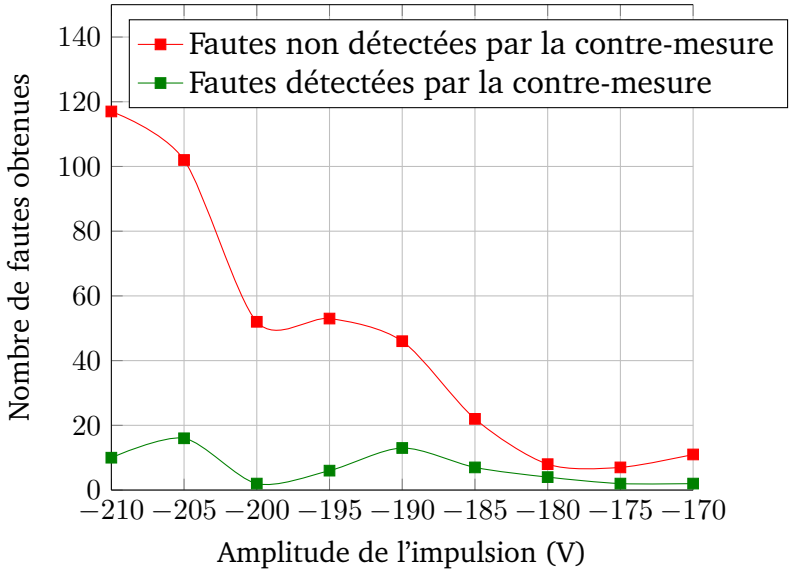


Figure 2.8. : Fautes détectées et non-détectées par la contre-mesure de duplication

de -210 V, 10 valeurs différentes pour le premier octet du chiffré ont été obtenues en sortie de l'algorithme. Ces 10 valeurs sont :

— `0x39` (la valeur du chiffré non-fauté)

— `0xe9` (la valeur permettant la cryptanalyse présentée)

— `0x00`, `0x08`, `0x18`, `0x19`, `0x3C`, `0x90`, `0xB8`, `0xD0`

La répartition des valeurs obtenues pour le premier octet du chiffrement pour une tension d'impulsion de -210 V et en fonction de l'instant d'injection est présentée sur la figure 2.9. Certains instants d'injection ne sont associés à aucune valeur de sortie car une exception matérielle y a été déclenchée. Les fautes correspondant à la valeur `e9` (dont l'ordonnée est identifiée par la ligne rouge sur le graphique) ont été obtenues sur deux intervalles de temps, le plus long d'entre-eux faisant environ 5 ns.

Les expérimentations précédentes ont également montré qu'avec la plate-forme expérimentale utilisée l'injection de double fautes espacées de plus de 50 µs était parfaitement réalisable et qu'il était possible d'obtenir un taux de succès assez élevé pour celle-ci. Cette observation sur la possibilité de réaliser des double fautes sera notamment utilisée par la suite dans le chapitre 4. Elles montrent également qu'une contre-mesure basique de redondance temporelle au niveau de la fonction n'est pas robuste.

2.5 Conclusion

Dans ce chapitre, le montage utilisé pour la grande majorité des expérimentations de cette thèse a été présenté et ses éléments constitutifs ont été détaillés. Ce montage utilise une technique d'injection de fautes par impulsion électromagnétique et un

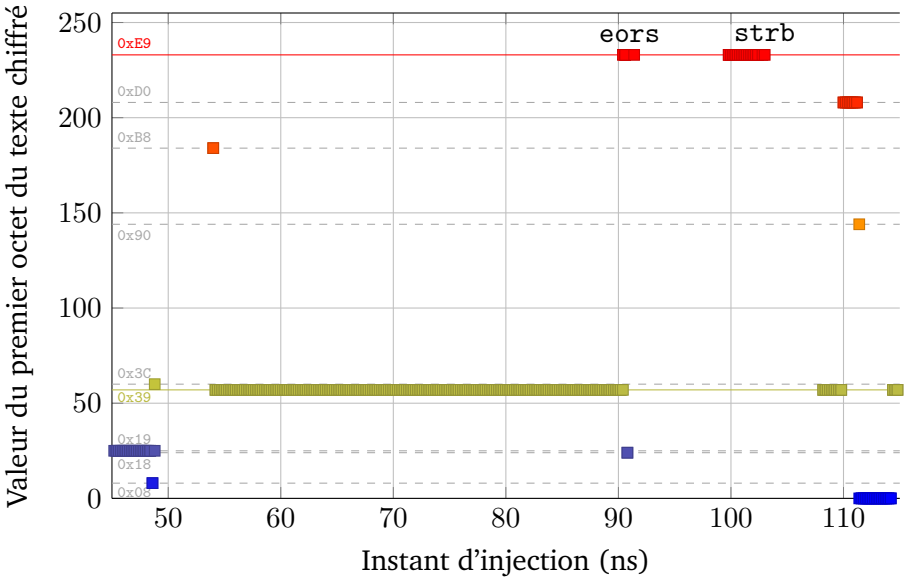


Figure 2.9. : Valeur du premier octet du texte chiffré en sortie de l'algorithme

microcontrôleur basé sur le processeur ARM Cortex-M3. Il se base sur celui proposé dans (DEHBAOUI, J.-M. DUTERTRE, ROBISSON et TRIA, 2012) mais a été conçu de façon à permettre des analyses plus précises en utilisant des grandeurs liées à l'état interne du processeur.

Deux premières applications de la plate-forme expérimentale ont également été proposées, en réalisant des attaques sur une implémentation de l'algorithme AES. Ces expérimentations ont permis de montrer l'efficacité pratique de la plate-forme pour réaliser des scénarios d'attaque, y compris pour réaliser des attaques basées sur une double injection de faute. Néanmoins, si l'exécution de certaines instructions a pu être perturbée pour les attaques présentées dans ce chapitre, celles-ci ont été menées en boîte noire et les effets précis de la technique d'injection n'ont pas réellement été caractérisés. C'est l'objet du prochain chapitre, qui a pour but d'étudier en détail les effets d'une injection par impulsion électromagnétique avec le banc d'attaque présenté dans ce chapitre en fonction des différents paramètres et en vue d'élaborer un modèle de fautes au niveau assembleur.

3 Validation d'un modèle de fautes au niveau assembleur

*Certains des travaux présentés dans ce chapitre ont fait l'objet d'articles présentés lors des conférences FDTC 2013 (*Moro, Dehbaoui *et al., 2013) et IEEE HOST 2014 (*Moro, Heydemann, Dehbaoui *et al., 2014).*

Sommaire

3.1 Introduction

Dans le chapitre précédent, le montage expérimental d'injection électromagnétique utilisé dans cette thèse a été présenté. De premières expérimentations ont permis de vérifier son bon fonctionnement et sa possible utilisation pour réaliser des attaques. Néanmoins, ce montage fait intervenir un grand nombre de paramètres expérimentaux dont l'influence doit être étudiée pour d'une part avoir une meilleure maîtrise du processus d'injection de fautes et d'autre part avoir une meilleure compréhension des fautes observées.

Une étude de l'influence de différents paramètres expérimentaux d'injection électromagnétique est présentée en 3.2. De celle-ci, il ressort que le cas particulier des

corruptions d'instructions et de données doit être analysé plus précisément, ce qui est présenté en 3.3. Cette étude permet ensuite d'élaborer un modèle des fautes réalisées au niveau Register-Transfer Level (RTL), celui-ci est présenté en 3.4. De ce modèle RTL, une abstraction vers un modèle de plus haut niveau, au niveau des instructions assembleur, est finalement présentée en 3.5.

3.2 Étude expérimentale des paramètres d'injection de fautes

Le but des paragraphes qui suivent est d'isoler l'influence de différents paramètres expérimentaux sur la génération de fautes. En raison de notre manque d'informations sur certains détails de l'architecture interne du microcontrôleur ciblé, notre approche part de l'expérimentation pour proposer un modèle exploitable à un niveau d'abstraction supérieur. Les paramètres principaux que nous pouvons faire varier sont listés dans le tableau 3.1.

Table 3.1. : Paramètres expérimentaux

Paramètres d'injection électromagnétique	- Position x-y-z de la sonde d'injection - Instant d'injection - Tension de l'impulsion
Paramètres du microcontrôleur	- Type des instructions exécutées - Localisation des instructions et données (RAM ou Flash) - Tension d'alimentation

Il est très important de préciser que les valeurs (de tension, de position d'antenne, ...) présentées dans cette thèse pour ces différents paramètres expérimentaux ne doivent pas être considérées de manière absolue. Comme de nombreuses études et expérimentations ont été réalisées à l'aide de ce dispositif expérimental, l'antenne d'injection a été replacée à de nombreuses reprises entre deux expérimentations différentes. Ainsi, de petites différences dans le positionnement de l'antenne entraînent par exemple le fait que des fautes aient pu être obtenues à 190 V pour une expérimentation et à 150 V pour une autre. Néanmoins, au niveau de la position de l'antenne d'injection, ces variations ne dépassent pas quelques millimètres et une même zone bien délimitée et d'une taille approximative de quelques dizaines de millimètres carrés du circuit a été utilisée pour positionner l'antenne pour l'ensemble des expérimentations de cette thèse.

Pour les expérimentations qui suivent, la fréquence du microcontrôleur a été fixée à 56 MHz. De plus, la longueur de l'impulsion a été fixée à 10 ns, ce qui représente un temps plus court que la période d'horloge (de 17.8 ns). L'influence séparée de ces différents paramètres expérimentaux est donc étudiée dans les paragraphes qui suivent, en commençant par étudier la répétabilité des fautes injectées.

3.2.1 Répétabilité des fautes injectées

Cette expérimentation vise a étudier la répétabilité du processus d'injection de fautes. On s'intéresse ici à l'instruction sur 16 bits `ldr r0,[pc,#40]` qui charge une donnée depuis la mémoire Flash, l'adresse de la donnée à charger étant relative à la valeur du compteur de programme. Cette instruction a été générée depuis la pseudo-instruction `ldr r0,=0x12345678`. Pour cette expérimentation, la tension de l'impulsion a été fixée à 190 V, la position d'antenne fixée et l'instant d'injection fixé à une valeur constante permettant d'obtenir des fautes en sortie. 10000 exécutions de cette instruction ont été réalisées dans ces conditions expérimentales, et les résultats sont présentés dans le tableau 3.2. Dans ce tableau, chaque valeur de sortie observée est associée à sa fréquence d'apparition. On constate donc que les valeurs obtenues dans le registre suivent une distribution de probabilités, avec certaines fautes en sortie plus probables que d'autres.

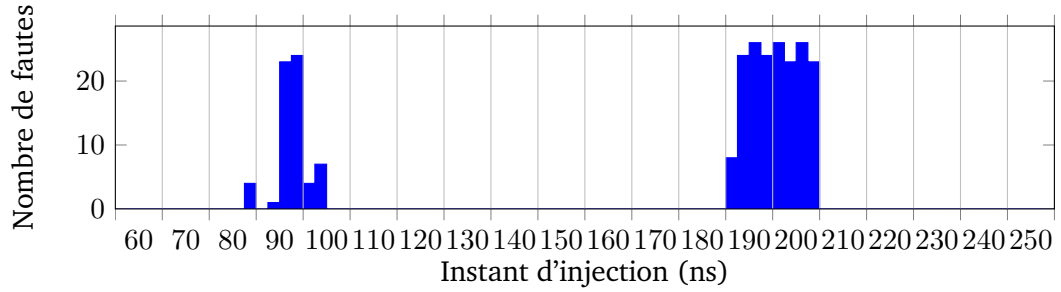
Table 3.2. : Valeurs fautées associées à leur taux d'occurrence et obtenues à la suite d'une injection de fautes sur une instruction `ldr`

Valeur chargée dans le registre	Taux d'occurrence
1234 5678 (pas de faute)	60.1%
FFF4 5679	27.4%
FFFC 5679	12.3%
FFFC 567b	0.1%
FFFC 7679	0.1%

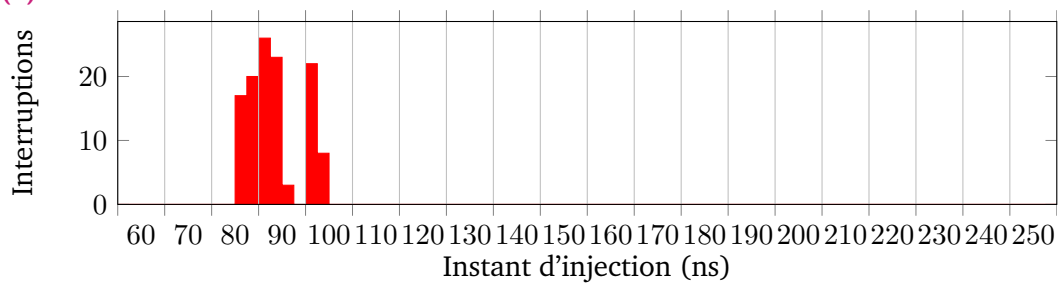
3.2.2 Instant d'injection

Pour étudier l'impact de l'instant d'injection, on s'intéresse ici à l'instruction sur 16 bits `ldr r0,[pc,#40]` qui charge une donnée depuis la mémoire Flash, l'adresse de la donnée à charger étant relative à la valeur du compteur de programme. Cette instruction a été générée depuis la pseudo-instruction `ldr r0,=0xCAFECAFE`. Pour cette étude, une tension de -210 V a été utilisée, et un intervalle de temps de 200 ns (correspondant à environ 12 cycles d'horloge) a été balayé par pas de 200 ps. Pour chaque instant d'injection, le processus d'injection de fautes a été réalisé deux fois. Les résultats de cette expérimentation sont présentés sur la figure 3.1a, qui montre le nombre de fautes obtenues dans le registre de sortie en fonction de l'instant d'injection de l'impulsion. On y distingue clairement deux intervalles temporels distincts pour lesquels des fautes sont obtenues dans `r0`. La figure 3.1b représente les déclenchements d'interruptions obtenus en fonction de l'instant d'injection. On constate qu'une grande partie de ces déclenchements d'interruptions a lieu légèrement avant le premier intervalle pour lequel des fautes sont générées. Par ailleurs, l'ensemble des interruptions déclenchées ont été du type `Usage Fault`[1]. Ainsi, deux intervalles de temps distincts permettent d'obtenir des fautes sur l'instruction visée.

1. Pour rappel, les exceptions de type `Usage Fault` se déclenchent principalement en cas d'instruction invalide, d'opération interdite sur certains registres ou encore de division par zéro



(a) Sorties fautées



(b) Déclenchements d'interruptions

Figure 3.1. : Influence de l'instant d'injection lors de la perturbation d'une instruction `ldr`

3.2.3 Position de l'antenne d'injection

La table X Y Z utilisée permet de faire varier la position de l'antenne d'injection. Dans cette expérimentation, une position pour l'axe Z a été fixée et seule l'influence de la position selon les axes X et Y est étudiée. Cette expérimentation est réalisée à l'échelle d'une seule instruction sur 32 bits de type `ldr` qui charge la valeur `0x12345678` depuis la mémoire Flash dans le registre `r8`. Cette injection de fautes a été réalisée sur un intervalle de temps de 30 ns, par pas de 1 ns. L'intervalle de 30 ns balayé dans cette expérimentation correspond au second intervalle trouvé en 3.2.2, pour lequel aucune interruption n'a été déclenchée. Par ailleurs, une tension d'injection de -210 V a été utilisée. Enfin, la sonde a parcouru un carré de 3 mm de côté au dessus du circuit, par pas de 200 µm. La figure 3.2 montre les résultats de cette expérimentation. Elle représente, pour différentes valeurs de l'instant d'injection, le poids de Hamming de la valeur chargée dans le registre `r8`. Ce poids de Hamming est représenté par une échelle de couleurs, et le poids de Hamming de la valeur `0x12345678` attendue est de 13.

Les résultats montrent que le poids de Hamming obtenu dans le registre de sortie varie avec l'instant d'injection et la position d'antenne. Pour les premières valeurs d'instant d'injection, des poids de Hamming nuls ont été obtenus (correspondant à la valeur `0x0`) et des poids de Hamming plus élevés ont été obtenus pour des instants d'injection plus tardifs. Cette expérimentation confirme donc le fait que la zone permettant d'injecter des fautes est très réduite et montre également que pour

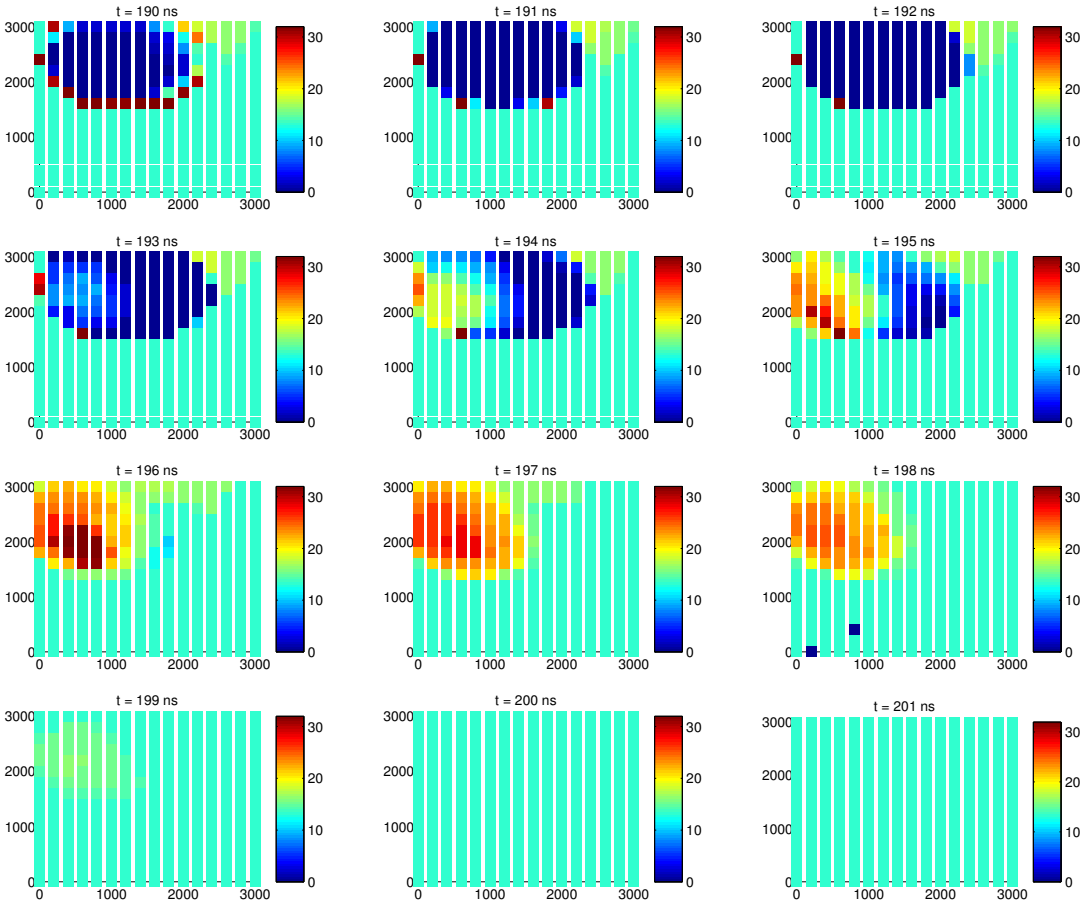


Figure 3.2. : Influence de la position X-Y de l'antenne d'injection lors de la perturbation d'une instruction `ldr`

un même instant d'injection la valeur obtenue dans le registre de sortie peut varier avec la position d'antenne.

3.2.4 Tension d'injection

Pour évaluer l'influence de la tension d'injection, le programme de test est constitué d'une seule instruction `ldr r4,[pc,#44]` (sur 16 bits) qui charge dans le registre `r4` la valeur `0x12345678` située à l'adresse `pc+44` (en mémoire Flash, adresse relative à la valeur du compteur de programme). Cette instruction a été générée depuis la pseudo-instruction `ldr r4,=0x12345678`. Pour cette expérimentation, l'intervalle temporel parcouru correspond au second intervalle présenté en 3.2.2, pour lequel aucune interruption n'a été déclenchée. Le tableau 3.3 montre les valeurs obtenues dans `r4` pour différentes valeurs de tension d'injection. Nous avons reporté pour chaque tension la valeur fautée avec le plus grand taux d'apparition. Selon ces résultats, le fait d'augmenter la tension d'impulsion augmente le poids de Hamming de la valeur chargée dans le registre et aboutit à un effet de *mise à 1* des bits. Toutefois, il semble que seules les instructions de type `ldr` depuis la mémoire Flash permettent d'obtenir ce type d'effet de *mise à 1* des bits. Plusieurs expérimentations

visant à perturber des instructions réalisant un transfert depuis la mémoire SRAM ont été menées sans succès. Enfin, la figure 3.3 montre la distance de Hamming par rapport à la valeur attendue (`0x12345678`) en fonction de la tension d'injection. Ainsi, pour cette expérimentation, la tension d'impulsion a eu une influence sur les fautes obtenues lors d'un chargement de données depuis la mémoire Flash.

Table 3.3. : Influence de la tension d'injection sur la valeur fautée

Tension de l'impulsion	Valeur chargée	Taux d'occurrence
170 V	1234 5678 (pas de faute)	100%
172 V	1234 5678 (pas de faute)	100%
174 V	9234 5678	73%
176 V	FE34 5678	30%
178 V	FFF4 5678	53%
180 V	FFFD 5678	50%
182 V	FFFF 7F78	46%
184 V	FFFF FFFB	40%
186 V	FFFF FFFF	100%
188 V	FFFF FFFF	100%
190 V	FFFF FFFF	100%

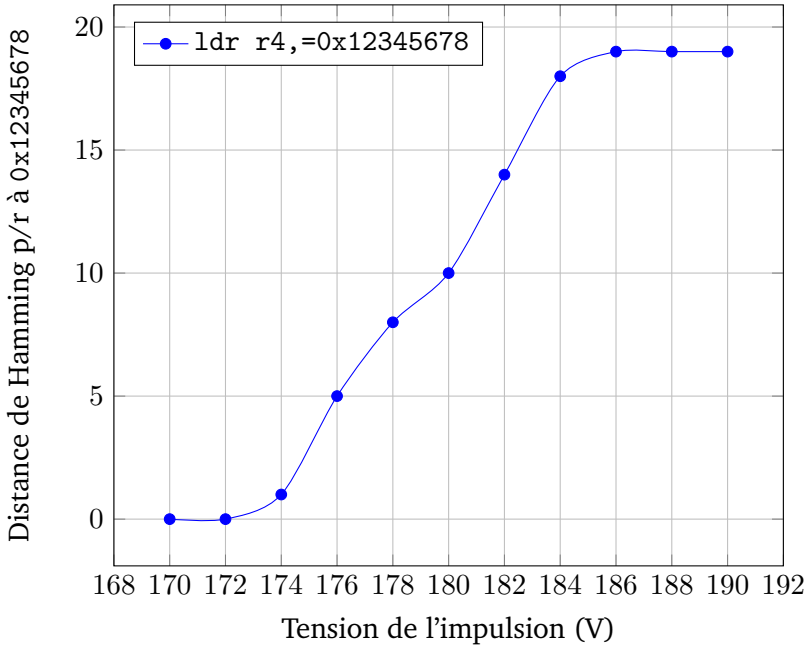


Figure 3.3. : Distance de Hamming de la faute la plus fréquente par rapport à `0x12345678` en fonction de la tension d'injection

3.3 Corruptions de données et d'instructions

Pour une instruction qui manipule des données, une faute observée en sortie peut être due à une corruption de la donnée manipulée, mais également à la corruption du code binaire de l'instruction elle-même. Le but de cette section est donc de

distinguer ces deux cas en proposant des méthodes et des expérimentations pour arriver à mieux caractériser l'origine des fautes générées. Pour cela, un raffinement du protocole expérimental présenté dans le chapitre 2 a été mis en place. Celui-ci a pour but de déterminer par simulation si une faute observée peut être la conséquence d'une corruption du code binaire d'une instruction. Dans le cas où aucune corruption d'instruction ne peut expliquer une faute observée, il est donc fortement probable que cette faute soit due à la corruption de la donnée manipulée. Ce raffinement du protocole expérimental est présenté en 3.3.1. Enfin, des expérimentations qui visent à isoler l'influence des corruptions d'instructions et de données sont présentées en 3.3.2.

3.3.1 Simulation de corruption d'instructions

Avec la plate-forme expérimentale utilisée dans cette thèse, il est impossible d'accéder à certaines données internes du microcontrôleur comme le registre d'instruction. Ce registre contient l'encodage binaire de l'instruction en cours de traitement dans le pipeline. Or, pour certains phénomènes observés dans ce chapitre, une connaissance de ce registre d'instruction serait une information précieuse pour une meilleure compréhension des effets du mécanisme d'injection de fautes. En l'absence d'accès à ce registre et pour expliquer certains effets d'une injection de fautes, un outil de simulation de corruption d'instructions a été développé.

3.3.1.1. Objectifs de l'outil de simulation

La simulation de corruption d'instructions a pour but de trouver des modèles qui expliquent des états de sortie observés expérimentalement. Un état du circuit se caractérise par la valeur des registres, des *flags* du processeur et de la mémoire. Une instruction permet normalement de passer d'un état initial `A` à un état attendu `B`. Toutefois, en présence d'une injection de fautes, l'instruction a pu être corrompue, et une instruction modifiée fait passer le processeur de l'état `A` à l'état `B'`. Le but de ce simulateur de corruption d'instructions est de retrouver l'ensemble des instructions qui permettent d'atteindre l'état `B'` à partir de l'état `A`. Cette approche est résumée par la figure 3.4.

Ce simulateur permet donc de retrouver une liste d'instructions admissibles qui peuvent mener à un état observé expérimentalement. Ce terme d'*admissible* est ici très important : plusieurs corruptions d'instructions peuvent très bien expliquer une sortie fautée obtenue expérimentalement. Quand aucune instruction ne peut expliquer une sortie fautée, il y a donc de grandes chances que cette sortie fautée soit due à une faute sur une donnée directement.

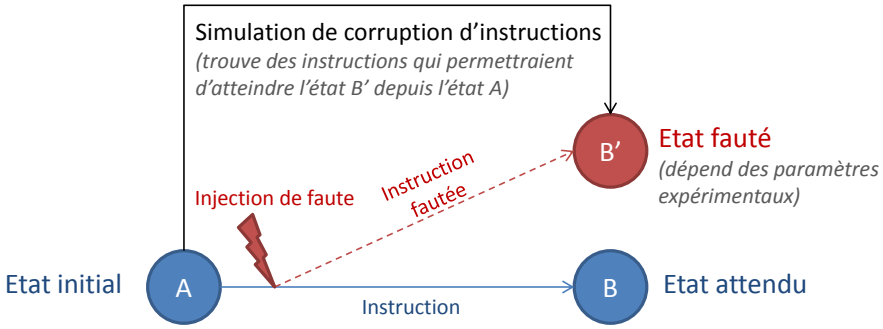


Figure 3.4. : Approche visant à modéliser les fautes injectées

Pour réaliser cette simulation, un programme spécifique a été développé, basé sur la bibliothèque Keil UVSOCK[2]. Ce programme de simulation est capable de contrôler le debugger lors d'une exécution du simulateur de Keil µVision. Il émule une faute sur le flot de contrôle en remplaçant à la volée l'instruction ciblée. Un remplacement d'instruction peut expliquer une sortie expérimentale si les données internes considérées au point d'observation à la fin du programme testé sont les mêmes pour la simulation et les mesures. Par ailleurs, cette simulation est réalisée sur le même binaire que celui qui est utilisé pour les expérimentations d'injection de fautes.

3.3.1.2. Espace de recherche et limitations de la simulation

Le jeu d'instructions Thumb-2 regroupe à la fois des instructions 16 et 32 bits. Il représente donc un espace de recherche sur 32 bits. 32 bits chargés depuis la mémoire d'instructions lors d'une étape de *fetch* peuvent correspondre à :

- une instruction sur 32 bits
- deux instructions sur 16 bits
- les 16 bits de poids faible d'une instruction sur 32 bits et une instruction sur 16 bits
- une instruction sur 16 bits et les 16 bits de poids fort d'une instruction sur 32 bits
- les 16 bits de poids faible d'une instruction sur 32 bits et les 16 bits de poids fort d'une instruction sur 32 bits

Une recherche exhaustive des remplacements d'instructions sur 32 bits n'est pas réalisable en pratique car trop longue. La part du jeu d'instructions pour laquelle les instructions sont codées sur 32 bits est majoritairement creuse, il est donc possible de retirer un grand nombre de branches dans l'espace de recherche en faisant en

2. Le programme de pilotage du banc d'injection présenté dans le chapitre 2 est également basé sur la bibliothèque Keil UVSOCK

sorte que l'outil de simulation ne teste que les codes d'instructions valides. Une simulation exhaustive des instructions 32 bits du jeu d'instructions Thumb-2 est donc possible. Toutefois, l'outil de simulation ne peut pas tester exhaustivement le cas de deux instructions sur 16 bits qui se suivent et peut seulement modéliser le remplacement de 16 des 32 bits chargés. Pour 32 bits de mémoire d'instructions, l'outil de simulation parcourt donc l'ensemble des instructions sur 16 bits pour les bits de poids fort en laissant inchangés les 16 bits suivants, puis procède de même avec les bits de poids faible. Les trois derniers cas présentés précédemment, où au moins une instruction sur 32 bits est coupée en deux ne sont pas modélisés par l'outil [3].

3.3.2 Résultats expérimentaux

Les paragraphes qui suivent détaillent des expérimentations visant à mettre en évidence des fautes par corruption d'instructions et corruption de données. Au niveau de la terminologie, on désignera par l'expression *fautes sur le flot de contrôle* l'ensemble des fautes qui corrompent une instruction et par *fautes sur le flot de données* l'ensemble des fautes qui corrompent seulement une donnée manipulée. Les expérimentations qui suivent ont été réalisées en utilisant deux classes simples de programmes de test. Les valeurs contenues dans les registres au début des programmes de test des expérimentations qui suivent sont détaillées dans le tableau 3.4.

Table 3.4. : Valeurs initiales des registres au début de l'exécution de la fonction visée

Donnée/registre	Valeur
`r0`	Une adresse mémoire en RAM
`r1` à `r4`	0x1 à 0x4
`r5` et `r6`	Non significatif
`r7`	0x100
`r8` à `r12`	0x00
Contenu de l'adresse pointée par `r0`	0x00

3.3.2.1. Fautes au niveau du flot de contrôle

Pour mettre en évidence les fautes sur le flot de contrôle, une suite d'instructions `nop` est utilisée [4]. Comme les instructions de `nop` n'ont aucun effet, une éventuelle sortie fautée est normalement bien plus facile à remarquer et à expliquer. Les expérimentations qui suivent ont été menées pour une tension d'injection de 190 V avec une position d'antenne fixée et en faisant varier l'instant d'injection. Plusieurs types d'effets ont été obtenus, ceux-ci sont présentés dans les paragraphes qui suivent.

3. L'outil de simulation a été testé sur des codes pour lesquels de tels non-alignements d'instructions n'étaient pas présents

4. Cette méthode est également utilisée dans (Spruyt, 2012)

Exceptions matérielles Les expérimentations réalisées ont parfois mené au déclenchement d'exceptions matérielles de type `Usage Fault`. Plus précisément, les sous-classes d'exception `No coprocessor` et `Undefined instruction` ont été les seules sous-classes d'exceptions de type `Usage Fault` qui ont été observées. Or ces deux types d'interruptions peuvent être déclenchées par un code d'instruction invalide ou pour une instruction pour coprocesseur non exécutable en l'absence de coprocesseur. Il semble alors que la faute ait été injectée dans les étapes de *fetch* ou de *decode* du pipeline.

Remplacements d'instructions Dans l'état initial avant l'exécution de l'instruction de test, le registre `r0` pointe vers une adresse mémoire A en SRAM. Suite à plusieurs des injections de fautes réalisées pour cette expérimentation, la valeur de `r0` a été observée à cette adresse A au lieu de la valeur attendue. Dans ce cas particulier, la simulation de remplacement d'instruction a montré que le seul remplacement possible est l'instruction `str r0,[r0,#0]`, qui stocke le contenu de `r0` à l'adresse A pointée par `r0`. De plus, la valeur `0x100` a également été observée à cette adresse A pour une autre expérimentation. Il s'avère que la valeur `0x100` correspond également au contenu du registre `r7`, et plusieurs dizaines de remplacements d'instructions sont admissibles pour expliquer ce résultat. Aucune faute sur `r1-r6` et `r8-r14` n'a été observée, mais des fautes sur le registre général `r7` et le compteur de programme `r15` ont été obtenues. Ces fautes peuvent également être expliquées par au moins un remplacement d'instruction assembleur. De façon informelle, les fautes sur `r7` et `r15 (pc)` sont apparues bien plus souvent que les fautes sur l'adresse mémoire pointée par `r0`. Les résultats obtenus laissent penser que ce taux d'occurrence plus haut pour des fautes sur ces registres `r7` et `r15` pourraient être dus à des corruptions d'instructions, où l'instruction initiale aurait été remplacée par une instruction dont l'encodage binaire a un poids de Hamming plus élevé. En effet : la plupart des instructions 16 bits peuvent seulement manipuler les registres `r0` à `r7` [5] (le numéro du registre étant encodé sur 3 bits, ce qui permet d'encoder des nombres allant de 0 à 7). Dans une instruction 16 bits, `r7` est codé par `111` dans le code de l'instruction. Le fait que les registres `r0-r6` soient encodés avec un nombre plus faible de `1` peut possiblement expliquer ce taux d'occurrence plus haut sur le registre `r7`.

Bilan Bien entendu, les paragraphes qui précèdent ne prétendent pas établir une liste complète de toutes les fautes possibles. En raison du très grand nombre de configurations possibles pour les paramètres d'injection, calculer des pourcentages d'occurrences de fautes ne serait pas pertinent. Néanmoins, qualitativement parlant, ces paragraphes illustrent le fait que très peu de types de fautes ont été observés. En conclusion pour cet ensemble d'expérimentations, chaque sortie fautée sur cette séquence de `nop` a au moins un remplacement d'instruction qui peut l'expliquer. Les

5. Par exemple, une opération comme `movs r7,#ff` est assemblée sous la forme d'une instruction 16 bits, alors qu'une opération comme `movs r8,#ff` est assemblée sous la forme d'une instruction 32 bits

fautes produites ont dont bien un impact sur le flot d'instructions, et ce constat nous amènera donc à une étude plus détaillée des différentes étapes du pipeline dans la section 3.4.

3.3.2.2. Fautes au niveau du flot de données

Pour mettre en avant les fautes sur le flot de données, une instruction `ldr` isolée qui charge un mot depuis une case mémoire dans un registre est utilisée. Les paragraphes qui suivent présentent les résultats expérimentaux obtenus selon l'encodage utilisé pour cette instruction `ldr`.

Fautes sur une instruction 16 bits Pour cette expérimentation, une instruction `ldr r4,[pc,#44]` sur 16 bits a été visée. La valeur initiale de `r4` est `0x0` et `pc+44` est une adresse en mémoire Flash qui contient `0x12345678`. Plusieurs valeurs de `r4` fautées comme `0xFE345678` ou `0xFFF45678` ont été obtenues. Le jeu d'instructions Thumb-2 peut seulement permettre d'atteindre un nombre limité de constantes en une seule instruction. Certaines des valeurs observées, comme `0xFFF45678`, peuvent théoriquement être uniquement chargées avec une instruction de `ldr` par adressage indirect. Comme la mémoire ne contient aucun motif de type `FFF4`, une simple instruction `ldr` ne peut pas expliquer ce résultat. Une simulation sur les instructions 16 et 32 bits a donc été réalisée et aucune instruction ne permet d'atteindre un résultat de `0xFFF45678` dans `r4`.

Fautes sur une instruction 32 bits Comme une injection de fautes peut avoir eu un impact sur deux instructions 16 bits consécutives [6], l'expérimentation présentée précédemment ne permet pas de conclure avec certitude quant à la réalisation de fautes sur le flot de données. Pour résoudre ce problème, une autre expérimentation a été réalisée, dans laquelle l'instruction ciblée est un `ldr r8,[pc,#44]`, avec `0x12345678` stocké à l'adresse `pc+44`. Le fait d'utiliser `r8` au lieu de `r4` entraîne l'assemblage de cette instruction sous la forme d'une instruction 32 bits. Pour cette nouvelle configuration, différentes valeurs fautées comme `0xFFF45679` ou `0xFFFC5679` ont été obtenues. Ces valeurs de sortie mesurées expérimentalement n'ont pas pu être obtenues à la suite d'une simulation de corruption d'instructions. De plus, comme une partie du résultat fauté est similaire à la valeur normalement attendue, il est raisonnable de supposer que cette injection de faute a eu un impact sur le flot de données.

3.3.3 Besoin d'une analyse à un niveau RTL (Register-Transfer Level)

Bien que de nombreux résultats expérimentaux visant à caractériser les effets de la technique d'injection aient été présentés dans les paragraphes précédents, pouvoir

6. Qui auraient été chargées lors d'une seule étape de *fetch* sur 32 bits

déterminer l'élément précis du circuit qui a été fauté reste un problème difficile. Toutefois, plusieurs résultats présentés dans ce chapitre ont montré la possibilité de d'injecter des fautes sur des instructions de `ldr` à deux endroits distincts (présentée notamment en 3.2.2). De plus, plusieurs expérimentations (notamment celle présentée en 3.2.4) ont permis d'observer le fait que le type de mémoire (SRAM ou Flash) depuis lesquelles étaient chargées les données avaient une influence sur l'injection de fautes. En outre, les expérimentations conduites dans la section 3.3 ont permis de montrer que des fautes étaient injectées sur le flot de contrôle et d'autres fautes sur le flot de données. L'ensemble de ces observations nous a orientés vers une analyse des transferts sur les bus de données et d'instructions. La section qui suit étudie plus en détail les transferts sur le bus pour un processeur Cortex-M3 et fournit une explication à un niveau RTL pour les fautes qui ont été observées.

3.4 Modèle de fautes au niveau RTL

A la suite des travaux présentés dans la section précédente, un modèle de fautes au niveau RTL est proposé pour les chargements d'instructions en 3.4.1 puis les chargements de données en 3.4.2. Enfin, une validation expérimentale des modèles proposés est présentée en 3.4.3.

3.4.1 Chargement d'instructions

L'opération de *fetch* d'une donnée ou d'une instruction depuis la mémoire d'instructions requiert au minimum deux cycles d'horloge (ARM, 2006). La figure 3.5 montre un chronogramme des transferts sur le bus Advanced High-performance Bus (AHB) lors de l'exécution d'une séquence de `nop` sur 16 bits. Comme les *fetch* d'instructions font 32 bits de large, deux instructions `nop` sur 16 bits sont récupérées à chaque exécution de l'étape de *fetch* du pipeline. L'étape de *fetch* a besoin d'un cycle d'horloge pour écrire l'adresse de l'instruction sur le bus HADDRI, ainsi qu'un cycle d'horloge supplémentaire pendant lequel 32 bits de la mémoire d'instructions sont écrits sur le bus HRDATAI. Dans le cas d'un transfert depuis la mémoire SRAM, la valeur à transférer est écrite sur le bus HRDATAI au début de ce cycle d'horloge supplémentaire. Toutefois, comme la mémoire Flash a un temps de réponse bien plus long, cette valeur est écrite à la fin du second cycle d'horloge. Dans cette situation, l'opération la plus longue réalisée sur le second coup d'horloge est ce transfert sur le bus HRDATAI.

On s'intéresse maintenant au résultat présenté en 3.3.2.1, dans lequel un `nop` est remplacé par un `str r0,[r0,#0]`. Les encodages binaires pour le `nop` et le `str r0,[r0,#0]` sont présentés dans le tableau 3.5. Dans ce qui a été présenté pour une attaque qui vise une instruction `ldr` en 3.2.4, il semble que plus la tension d'impulsion est élevée, plus le poids de Hamming du mot récupéré est élevé. Toutefois, pour le cas courant, le poids de Hamming du code de l'instruction chargée a diminué.

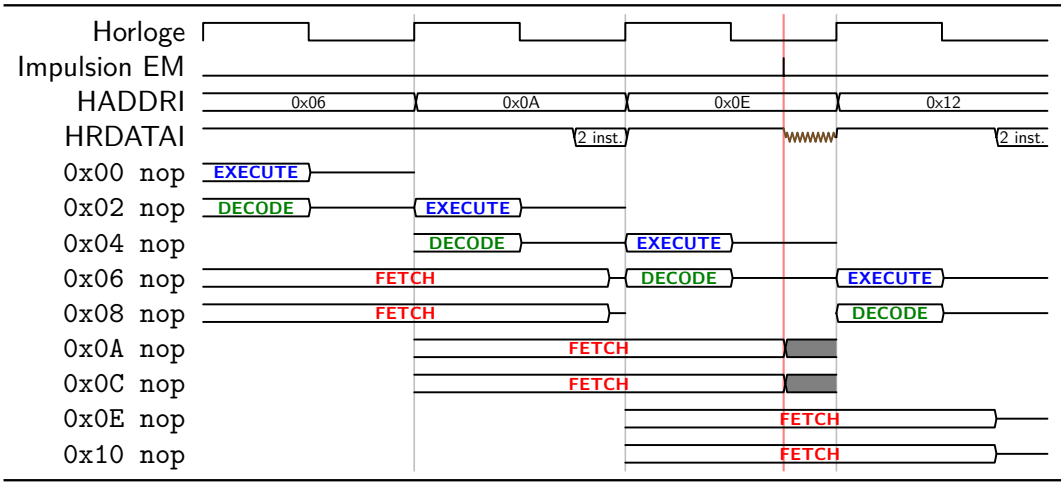


Figure 3.5. : Transferts sur le bus AHB pour la mémoire d'instruction avec une impulsion électromagnétique

Table 3.5. : Encodage binaire pour les instructions `nop` et `str r0,[r0,#0]`

Mnémonique	Instruction	Encodage binaire	Poids de Hamming
`nop`	BF00	10111111 00000000	7
`str r0,[r0,#0]`	6000	01100000 00000000	2

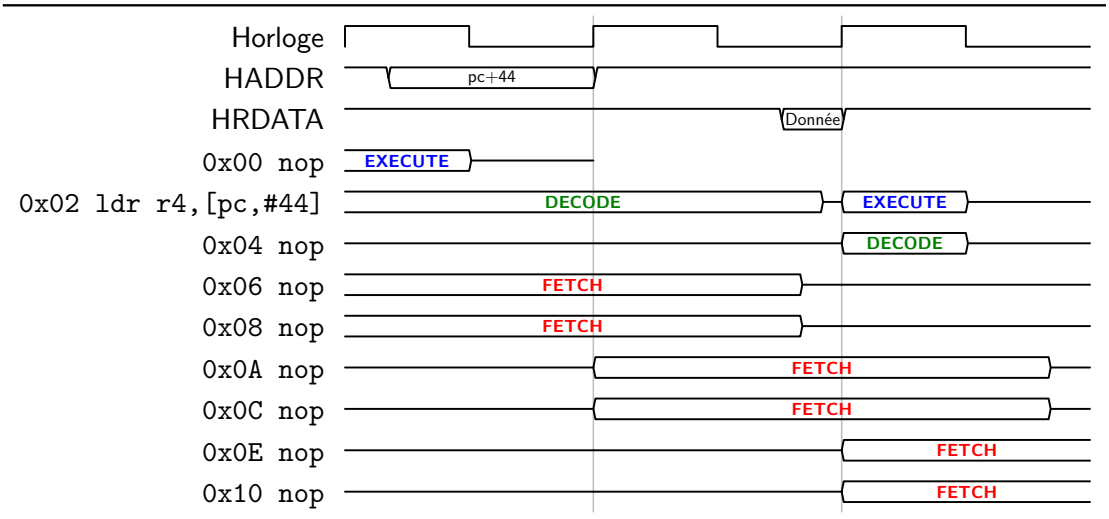
Les valeurs de précharge des bus ne sont pas précisées dans la spécification AHB, elles peuvent être choisies par le fabriquant du circuit. Pour le microcontrôleur utilisé, il semble que le bus HRDATAI ne soit pas préchargé à 1, comme une instruction de `nop` dont le poids de Hamming est 7 a été remplacée par une autre instruction dont le poids de Hamming est 2. En l'état, nous ne sommes pas capables d'inférer davantage de détails sur une possible précharge du bus HRDATAI.

3.4.2 Chargement de données

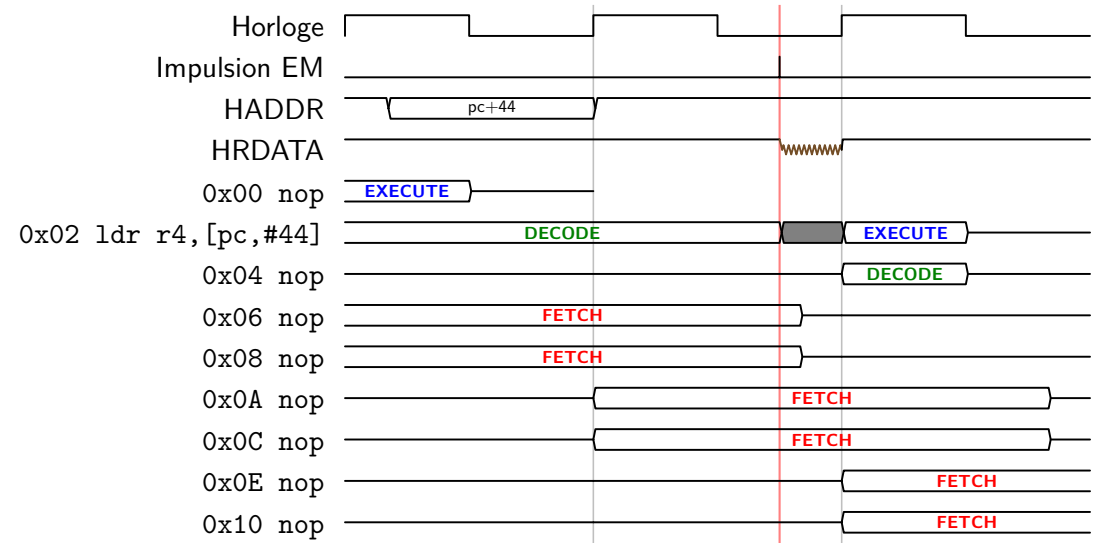
Pour le cas des fautes lors des chargements de données, l'opération la plus longue d'un cycle d'horloge est le transfert sur le bus HRDATA (ARM, 2006). La figure 3.6a montre les transferts sur le bus de données dans le cas d'une seule instruction `ldr` (soit une situation similaire aux expérimentations présentées dans les sections précédentes) sans injection de fautes. L'étape de *decode* de l'instruction `ldr` est plus longue car le processeur doit charger les opérandes de l'instruction et doit pour cela récupérer la valeur située à l'adresse `pc+44` en mémoire Flash. La figure 3.6b montre les mêmes transferts sur le bus en présence d'une impulsion électromagnétique.

3.4.3 Validation expérimentale de ce modèle RTL

Pour valider le modèle de corruptions des transferts depuis la mémoire Flash présenté précédemment, on s'intéresse au cas particulier d'une instruction `ldr` qui charge des données depuis la mémoire Flash. Pour le passage complet de cette instruction dans le pipeline, il y a donc deux transferts sur les bus : un premier qui correspond à



(a) Sans impulsion électromagnétique



(b) Avec une impulsion électromagnétique

Figure 3.6. : Transferts sur le bus AHB pour la mémoire de données

l'étape de *fetch* du pipeline (pour lequel le code binaire de l'instruction est transféré, cas similaire à celui présenté en 3.4.1) et un second qui correspond à l'étape de *decode* (pour lequel la donnée chargée par l'instruction est transférée, cas similaire à celui présenté en 3.4.2). Plus spécifiquement, on s'intéresse ici à l'instruction sur 16 bits `ldr r0,[pc,#40]` qui charge une donnée depuis la mémoire Flash (l'adresse de la donnée à charger est relative à la valeur du compteur de programme). Cette instruction a été générée depuis la pseudo-instruction `ldr r0,=0xCAFECAFE`.

Pour cette étude, une injection de fautes pour 9 valeurs de tension (de -210 V à -170 V) et sur un intervalle de temps de 200 ns a été réalisée. D'autres analyses (non présentées dans cette thèse) avec des tensions positives ont mené aux mêmes types de résultats. Le point d'arrêt du programme a été fixé juste après l'exécution de l'instruction `ldr` ciblée. Cet intervalle de 200 ns a été balayé par pas de 200 ps.

Pour chaque instant d'injection, le processus d'injection de fautes a été réalisé deux fois.

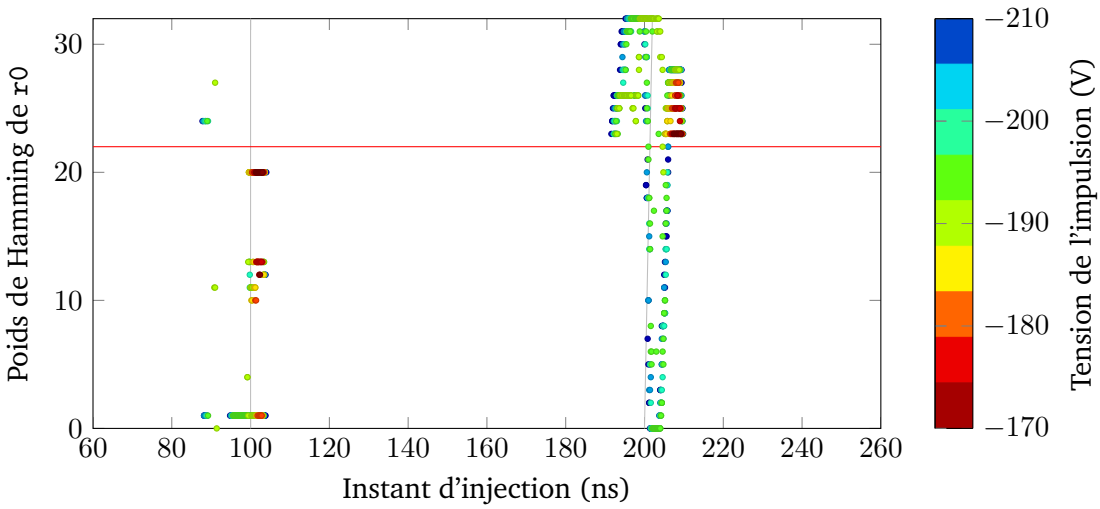


Figure 3.7. : Résultats d'injection de fautes pour l'instruction `ldr r0,[pc,#40]`

Les résultats de cette campagne d'expérimentations sont présentés sur la figure 3.7. Cette figure montre le poids de Hamming de `r0` (quand aucune exception n'a été déclenchée) après l'exécution de l'instruction en fonction de la tension et de l'instant d'injection de l'impulsion. Le poids de Hamming de la valeur `0xCAFECAFE` est 22. Sur cette figure, on distingue très clairement deux groupes de sorties fautées, localisées autour de deux instants d'injection : autour de 100 ns et autour de 200 ns. Ces deux instants sont les mêmes que ceux présentés en 3.2.2. Pour ces deux instants, la distribution des valeurs fautées est très différente. Le premier groupe semble correspondre à l'étape de *fetch*. Comme le code binaire de l'instruction est corrompu, peu de remplacements d'instructions permettent d'obtenir une nouvelle instruction valide. Ainsi, la plupart de ces remplacements génèrent une instruction invalide qui déclenche une exception matérielle. Comme les exceptions matérielles ne sont pas affichées sur la figure, peu de valeurs différentes de poids de Hamming sont représentées pour ce premier groupe. Le second groupe semble correspondre à l'étape de *decode*. On peut observer une bien plus grande diversité dans les sorties fautées pour cet instant d'injection, ce qui tend à valider les hypothèses faites dans ce chapitre sur la corruption de la valeur chargée par l'instruction pendant cette étape de *decode*. Par ailleurs, on observe également que la tendance générale est à une augmentation du poids de Hamming de la donnée en sortie (ce qui fait écho aux résultats obtenus en 3.2.4), bien que des valeurs à poids de Hamming plus faible aient été obtenues. En conclusion, ces résultats expérimentaux sur une instruction `ldr` tendent à valider le modèle RTL défini précédemment, pour lequel les transferts depuis la mémoire Flash peuvent être corrompus.

3.5 Modèle de fautes au niveau assembleur

Dans cette section, nous cherchons à établir un modèle au niveau assembleur pour représenter les fautes obtenues expérimentalement. Un modèle dans lequel un attaquant est capable d'empêcher l'exécution d'une instruction assembleur a été utilisé à de nombreuses reprises dans la littérature scientifique (voir 1.4.2.2) et a également pu être utilisé pour réaliser les attaques présentées en 2.4. Cette section étudie en détail la validité de ce modèle de saut d'instruction sur un exemple en 3.5.1 puis propose un ensemble d'hypothèses pour expliquer les résultats expérimentaux obtenus en 3.5.2.

3.5.1 Validité du modèle de saut d'instruction

Les paragraphes qui suivent cherchent à évaluer dans quelle mesure le modèle de saut d'instruction peut être représentatif des fautes qui sont injectées en pratique.

3.5.1.1. Présentation de l'implémentation visée

Pour évaluer la validité d'une possible modélisation des fautes induites sous la forme d'un saut d'instruction, on s'intéresse ici aux effets sur une portion de code plus large à l'échelle d'une fonction. Le but est d'évaluer, sur ce scénario d'exemple, le nombre d'injections de fautes dont l'effet est exactement équivalent à celui d'un saut d'instruction. La fonction étudiée est présentée dans le listing 3.1 et réalise une addition des éléments d'un tableau à 2 éléments dont les deux premières cases contiennent respectivement les nombres 1 et 2. Le résultat normal attendu en sortie de la fonction (stocké à l'adresse mémoire pointée par `r0`) est donc 3. Par ailleurs, le tableau a également une troisième case qui contient la valeur 4 mais qui n'est pas additionnée. Cette dernière case du tableau qui n'est pas ajoutée aux autres par la fonction a été insérée afin de pouvoir repérer un éventuel ajout de tour de boucle.

Listing 3.1 : Addition des éléments d'un tableau

```
1 boucle_addition :
2     ldr  r4 , [r2 ,r1 , lsl #2]     ; r4 = array[i]
3     ldr  r3 , [r0 ,#0]              ; r3 = result
4     add  r3 , r3 , r4               ; r3 = r3 + r4
5     str  r3 , [r0 ,#0]              ; resultat = r3
6     add  r1 , r1 , #1               ; r1 = r1 + 1
7     cmp  r1 , #2                    ; r1 == 2 ?
8     blt  boucle_addition
```

Cette expérimentation a été réalisée à 24 MHz, pour une position d'antenne fixe et pour un instant d'injection qui a varié sur un intervalle de temps de 1100 ns par pas de 200 ps. Cet intervalle temporel correspond au temps d'exécution de la boucle complète. Pour chaque pas de temps, 5 essais d'injection de faute ont été

réalisés. Le point d'observation auquel le programme s'arrête est situé à la fin de cette fonction, avant l'instruction de manipulation de pile. Au total, cette expérimentation a nécessité 27500 injections de faute.

3.5.1.2. Résultats expérimentaux

Le tableau 3.6 présente plusieurs statistiques sur les résultats expérimentaux obtenus. Parmi les 27500 tentatives d'injection de faute réalisées, 980 ont abouti au déclenchement d'une exception matérielle. 120 tentatives d'injection de fautes ont abouti à une sortie contenant au moins une faute (sur le résultat ou l'un des registres). Ces 120 résultats fautés correspondent à 28 instants d'injection différents. Par ailleurs, 7 sorties différentes ont été obtenues pour la variable de résultat.

Table 3.6. : Statistiques sur les résultats d'injection de fautes

Nombre d'exceptions matérielles	**980**
Nombre de lignes contenant un résultat fauté	**120**
Nombre d'instants où une faute a été obtenue	**28**
Nombre de sorties différentes pour la variable de résultat	**7**

Ces 7 valeurs sont : `0x3` (valeur non-fautée), `0x2`, `0xd1080002`, `0x0`, `0x5710800`, `0x1` et `0x7`. Par une analyse du programme testé en ne considérant que la valeur de sortie, il ressort que :

— la valeur `0x2` peut être obtenue suite à un saut de l'instruction `add r3, r4` en premier tour de boucle

— la valeur `0x1` peut être obtenue suite à un saut de l'instruction `add r3, r4` en second tour de boucle

— la valeur `0x7` peut être obtenue suite à un saut de l'instruction `cmp r1, #2` en second tour de boucle

La valeur `0x0` est plus difficile à expliquer par un saut d'instruction. Les valeurs `0xd1080002` et `0x5710800` ne correspondent ni à une adresse mémoire, ni à la valeur d'un registre. Elles pourraient être issues de la perturbation d'un transfert de données lors de l'une des instructions `ldr` mais cette hypothèse est difficilement vérifiable.

3.5.1.3. Estimation du taux de couverture du modèle

Afin de connaître plus précisément le taux de couverture d'un modèle simplifié de saut d'instruction, l'approche par simulation présentée en 3.3.1 a été utilisée. L'outil de simulation de corruption d'instructions a été modifié de façon à seulement tester le saut d'une instruction. Comme l'espace de recherche a été considérablement réduit, la simulation peut donc être réalisée sur l'ensemble du code de la fonction de

manière très rapide. L'outil de simulation génère ainsi un ensemble de sorties fautées, qui correspondent chacune à une exécution du code pour laquelle une instruction (de 16 ou 32 bits) n'a pas été exécutée. Ensuite, une comparaison est réalisée entre ces sorties fautées et les sorties fautées obtenues expérimentalement.

En pratique, le programme de pilotage du banc et l'outil de simulation de sauts d'instruction permettent de récupérer les mêmes informations, à savoir les registres généraux, le registre xPSR et le résultat de sortie. Une comparaison peut alors être réalisée, sur le modèle de ce qui est présenté sur la figure 3.8. Sur cette figure, on peut donc constater que les deux lignes encadrées ont les mêmes valeurs de sortie pour les premiers registres (les autres données comparées étant également similaires). On peut donc affirmer que la ligne sélectionnée dans le tableau de mesures correspond à la simulation pour laquelle l'instruction `ldr r4,[r2,r1,lsl#2]` n'a pas été exécutée. Il est important de préciser que nous avons choisi un critère très strict pour définir si une sortie fautée peut correspondre à un saut d'instruction. Pour cela, il faut notamment que tous les registres (y compris les registres non-vivants) correspondent. Le taux de couverture du modèle serait donc probablement supérieur si nous avions considéré un critère moins restrictif en nous limitant aux registres vivants.

Simulation

(après saut de l'instruction `ldr r4,[r2, r1, lsl #2]`*)*

Interruption	r0	r1	r2	r3	r4	r5
Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10

Résultats expérimentaux

	Interruption	r0	r1	r2	r3	r4	r5
t= 37.0 ns	Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10
t= 37.2 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.4 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.6 ns	UsageFault	-	-	-	-	-	-

Figure 3.8. : Comparaison entre les résultats de simulation et les résultats expérimentaux

Pour cette expérimentation, le fichier de simulations comporte ainsi 14 lignes correspondant au déroulage des deux tours de boucle avec le saut d'une instruction et le fichier de mesures comporte 27500 lignes. Les résultats de cette comparaison ligne à ligne entre mesures et simulations sont présentés dans le tableau 3.7. Parmi les 120 lignes fautées obtenues expérimentalement, 31 d'entre-elles sont parfaitement expliquées par le saut d'au moins une des instructions de la fonction. Par ailleurs, ces 31 fautes expliquées ont été obtenues pour 8 valeurs d'instant d'injection différentes.

Cette expérimentation n'a pas pour but de se prétendre représentative à elle seule des scénarios d'injection de fautes sur microcontrôleur. Toutefois, elle met néanmoins

Table 3.7. : Correspondances entre les mesures et les simulations

Nombre d'expérimentations expliquées par un saut d'instruction	**31 / 120**
Nombre d'instants expérimentaux où la ligne est expliquée	**8 / 28**

en avant le fait qu'un pourcentage significatif (ici proche de 25%) des injections qui mènent à une faute peuvent exactement se modéliser sous la forme d'un saut d'instruction. En l'absence de moyens de mesure plus avancés (notamment pour pouvoir lire le contenu du registre d'instruction), il paraît pour l'instant difficile de se prononcer quant aux autres 75% de fautes. Néanmoins, le fait qu'une attaque semble avoir un pourcentage aussi élevé de chances de produire un effet exactement similaire à un saut d'instruction constitue un scénario d'attaque réaliste. Dans des travaux similaires sur un autre type de plate-forme expérimentale et pour une architecture différente, un taux de 50% de sauts d'instructions a également été obtenu (BALASCH et al., 2011). Des taux élevés de correspondance du modèle de saut d'instruction ont également été obtenus par injection électromagnétique pour plusieurs instructions spécifiques sur une plate-forme de type Cortex-M3 dans (HUMMEL, 2014).

3.5.2 Hypothèses pour expliquer les effets de sauts d'instructions

Les paragraphes qui suivent proposent plusieurs hypothèses qui permettraient d'expliquer pourquoi un pourcentage significatif des fautes réalisées semblent correspondre à des sauts d'une instruction. Sans connaissance du contenu du registre d'instruction, ces hypothèses sont avant tout des pistes d'études pour d'éventuels travaux futurs sur la question et n'ont pas fait l'objet d'expérimentations associées.

3.5.2.1. Remplacements d'instructions assimilables à des `nop`

Certaines perturbations d'instructions peuvent mener au remplacement de l'une des instructions du programme visé par une instruction n'ayant aucun effet sur les registres, la mémoire ou les *flags* du processeur. Quelques-unes de ces instructions sont présentées dans le listing 3.2.

Listing 3.2 : Quelques instructions sans effet

```
1 mov    r3,r3      ; copie d'un registre dans lui-même
2 pld    r1,#1      ; préchargement d'une donnée
3 nop               ; aucun effet
4 isb               ; flush du pipeline
```

Il y a un nombre très réduit d'instructions qui sont dans ce cas. Toutefois, selon le contexte d'exécution, certaines instructions peuvent être considérées comme sans effet. Le listing 3.3 présente un cas où une instruction de type `ldr` est remplacée par une instruction de `str` qui a déjà eu lieu précédemment. Dans ce cas, l'instruction `str` issue du remplacement a un effet équivalent à celui d'un `nop`.

Listing 3.3 : Quelques instructions sans effet dans ce contexte

```
1  ; Exécution normale
2  str     r0 ,[ r3 ,#0]
3  ldr     r0 ,[ r4 ,#0]
4
5  ; Exécution avec faute, ldr remplacée par str
6  str     r0 ,[ r3 ,#0]
7  str     r0 ,[ r3 ,#0]
```

Les situations présentées dans les paragraphes qui précèdent peuvent théoriquement arriver, mais apparaissent globalement assez improbables. Toutefois, il existe d'autres situations, détaillées dans les paragraphes qui suivent, pour lesquelles un effet macroscopique équivalent au saut d'une instruction peut être observé.

3.5.2.2. Effet macroscopique de saut d'instruction

Définition 1 *Un registre est dit **vivant** en un point du programme s'il existe un chemin de ce point à la sortie du programme le long duquel le registre est lu avant d'être possiblement écrit.*

Plus généralement, l'attaquant voit un effet macroscopique de saut d'instruction lorsque le remplacement d'instruction modifie un registre non vivant ou un *flag* non vivant. De même, un remplacement d'instruction qui modifierait le contenu d'une adresse mémoire non-utilisée ou qui ne sera pas lue avant d'être écrite n'aura aucun effet macroscopique. Dans le listing 3.4, lors de l'instruction `add`, le registre `r9` n'est plus vivant : il sera ensuite écrit par l'instruction de `pop` sans être lu. Le remplacement d'un `add` (ligne 6) par un `sub` qui manipule `r9` a, en sortie de la routine présentée ci-dessous, un effet *macroscopiquement équivalent* à celui d'un saut de l'instruction de `add`.

Listing 3.4 : Effet macroscopique de saut d'instruction

```
1  ; Exécution normale
2  push    {r1 , r2 , r8 , r9}
3
4  ldr     r1 ,[ r8 ,#0]
5  ldr     r2 ,[ r9 ,#0]
6  add     r1 , r1 , r2
7  str     r1 ,[ r8 ,#0]
8
9  pop     {r1 , r2 , r8 , r9}
10 bx      lr
11
12 ; Exécution avec faute, add remplacée par un sub
13 push    {r1 , r2 , r8 , r9}
14
```

```
15  ldr     r1,[r8,#0]
16  ldr     r2,[r9,#0]
17  sub     r9,r6,r11
18  str     r1,[r8,#0]
19
20  pop     {r1,r2,r8,r9}
21  bx      lr
```

En raison de l'utilisation dans un code standard de nombreuses sous-procédures qui font appel aux instructions de manipulation de la pile que sont `push` et `pop`, ce cas de remplacement par une instruction qui manipule des registres non-vivants semble nettement plus probable que les précédentes hypothèses.

3.5.2.3. Absence de précharge du bus

A la fin de la partie précédente sur la définition d'un modèle RTL, une difficulté qui a été rencontrée pour mieux comprendre les effets produits était liée à la stratégie de précharge des bus. Sans informations supplémentaires sur l'architecture interne du microcontrôleur, il est impossible de savoir s'il y a ou non une précharge du bus, ou s'il y en a une de connaître sa valeur.

Toutefois, dans l'hypothèse selon laquelle il n'y aurait pas de précharge de bus, au début d'un nouveau cycle d'horloge le bus reste donc chargé à la valeur qu'il avait lors du cycle d'horloge précédent. Ainsi, lors d'un cycle d'horloge de `fetch` d'instruction la valeur sur le bus d'instructions reste la même jusqu'à l'écriture de la nouvelle valeur. Une injection électromagnétique bien paramétrée pourrait ainsi permettre de maintenir cette valeur sur le bus au moment de l'échantillonnage. Ces phénomènes sont illustrés sur la figure 3.9 (où le code `BF00` correspond à une instruction de `nop` et le code `18D1` à l'instruction `adds r1,r2,r3`), sur laquelle on peut voir une corruption du transfert sur le bus lors du transfert du code de l'instruction `adds r1,r2,r3`.

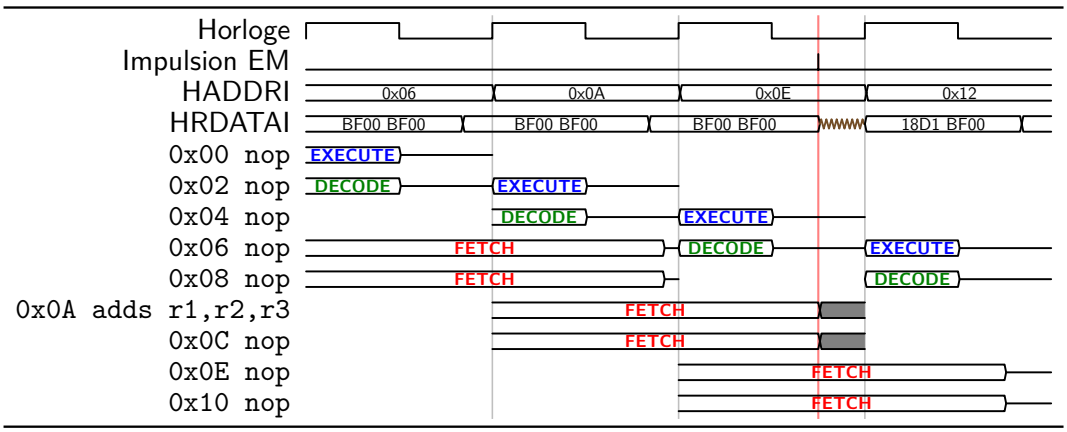


Figure 3.9. : Transferts sur le bus AHB sans précharge du bus

Une telle situation pourrait donc amener à une ré-exécution de l'instruction précédente. Or un certain nombre d'instructions du jeu d'instructions sont idempotentes [7], et la ré-exécution d'une instruction idempotente a un effet équivalent à celui de l'insertion d'un `nop`. Si on considère l'hypothèse selon laquelle aucune précharge du bus d'instructions n'a lieu en début de cycle d'horloge, alors le modèle RTL développé précédemment permet d'expliquer le fait qu'un nombre conséquent d'injections de fautes aboutissent à un effet macroscopiquement équivalent à celui du saut d'une instruction.

3.6 Conclusion et perspectives

Dans ce chapitre, une étude détaillée des effets d'une injection de fautes par impulsion électromagnétique sur un microcontrôleur récent a été présentée. Toutefois, travailler sur un microcontrôleur du commerce dans une approche qui, selon les cas, s'assimile à une approche en boîte noire ou en boîte grise, crée un ensemble de contraintes lorsqu'il s'agit de créer un procédé expérimental en pratique. L'approche proposée dans ce chapitre vise à émettre des hypothèses pour les fautes obtenues au niveau du code assembleur. Des informations supplémentaires sur la stratégie de précharge des bus sur le microcontrôleur utilisé, ou bien la possibilité d'observer le contenu du registre d'instruction permettraient une bien meilleure compréhension des phénomènes étudiés.

Les expérimentations menées confirment le fait qu'un attaquant peut agir à la fois sur le flot de contrôle et le flot de données, en provoquant des remplacements d'instructions ou en modifiant la valeur d'une donnée chargée depuis la mémoire Flash. Sur le microcontrôleur étudié dans cette thèse, les fautes sur le flot de données amènent à une augmentation du poids de Hamming de la valeur chargée. Nous ne prétendons pas que les explications données au niveau RTL sont les seules raisons pour lesquelles des fautes apparaissent au niveau du code assembleur. Néanmoins, ces expérimentations ont permis de nous fournir une première compréhension des fautes qu'une impulsion électromagnétique peut induire sur un programme embarqué.

Ces observations ont ensuite permis d'extraire un modèle de fautes simplifié au niveau du code assembleur et d'étudier les cas où l'injection de fautes amène à des effets assimilables à un saut d'instruction. Les expérimentations qui ont été menées dans ce chapitre ont permis de montrer que s'il semble très difficile de contrôler finement le remplacement d'une instruction par une autre instruction choisie, des effets de sauts d'instruction ont été obtenus pour un nombre significatif d'essais (25% sur une expérimentation de ce chapitre). Néanmoins, l'étude de la validité du modèle de saut d'instruction reste une question difficile et davantage de travaux

7. Une instruction idempotente est une instruction qui peut être appelée plusieurs fois consécutives sans problèmes car l'état atteint est le même à la suite d'une ou plusieurs exécutions de l'instruction

seraient certainement nécessaires à ce sujet. Au niveau d'un programme embarqué, de nombreuses attaques sont possibles grâce au saut d'une instruction et il est relativement aisé pour un attaquant de trouver de tels points faibles. Une contre-mesure qui protégerait l'exécution du programme contre les sauts d'une instruction serait donc très intéressante pour sécuriser un code embarqué. Le chapitre qui suit étudie donc le cas particulier d'un schéma de contre-mesure permettant de rendre un code embarqué résistant face aux sauts d'instruction.

4 Définition et vérification d'une contre-mesure logicielle

*Certains des travaux présentés dans ce chapitre ont fait l'objet d'un article présenté à la conférence PROOFS 2013 (*Heydemann *et al., 2013) et d'une version étendue de cet article pour le Journal of Cryptographic Engineering (*Moro, Heydemann, Encrenaz *et al., 2014).*

Sommaire

4.1 Introduction

Le chapitre précédent nous a amenés à élaborer un modèle bas-niveau de fautes sur microcontrôleur et en abstraire un modèle simplifié au niveau assembleur pour lequel un attaquant peut empêcher l'exécution d'une instruction. À partir de ce modèle, ce chapitre propose un schéma de contre-mesure qui peut s'appliquer de manière automatisée à un code générique.

Les contre-mesures face aux injections de fautes – qu'elles soient matérielles ou logicielles – utilisent généralement un principe de redondance pour permettre soit une détection de fautes soit une tolérance aux fautes injectées. Cette redondance peut être réalisée de façon parallèle pour des contre-mesures matérielles, mais doit être réalisée séquentiellement pour toute contre-mesure logicielle. Pour des implémentations logicielles, un tel principe s'applique généralement à l'échelle de l'algorithme voire de la fonction en dupliquant voire tripliquant son exécution et en comparant les sorties de ces différentes exécutions. Le chapitre 2 a montré

qu'une double injection de fautes avec un intervalle de temps équivalent à plusieurs milliers de cycles d'horloge entre les deux injections était parfaitement réalisable. Injecter deux fautes à un endroit bien choisi lors de deux exécutions successives d'un même algorithme a pu être réalisé en pratique. Néanmoins, comme présenté dans le chapitre 2, un délai minimal entre deux injections est nécessaire pour réaliser une telle double faute. Ainsi, bien que cela reste possible généralement à un coût plus élevé, il est bien plus difficile d'attaquer deux instructions séparées de quelques cycles d'horloge. La contre-mesure présentée dans ce chapitre est basée sur ce dernier point et propose une redondance locale et à l'échelle de l'instruction.

Cette contre-mesure qui vise à augmenter le niveau de sécurité d'un code générique se base donc à la fois sur un modèle de fautes par saut d'instruction et ce principe de redondance locale. Le schéma de contre-mesure repose sur une séquence de remplacement tolérante aux fautes pour chaque instruction du jeu d'instructions. Celui-ci peut se diviser en trois catégories d'instructions : les instructions idempotentes [1], celles qui peuvent être transformées en une suite d'instructions idempotentes, et enfin celles pour lesquelles nous n'avons trouvé aucune séquence de remplacement. Ces différentes classes d'instructions ainsi que leurs séquences de remplacement associées sont présentées en 4.2.

La séquence de remplacement doit conserver la sémantique de l'instruction initiale, que ce soit en présence d'une injection de faute ou non. Afin de garantir ces aspects pour la contre-mesure proposée, un schéma de vérification formelle à base de model-checking a été mis en place. Celui-ci permet de certifier l'efficacité théorique de la contre-mesure proposée face au modèle considéré. Ce schéma de vérification se base sur la modélisation d'une part des instructions à renforcer et d'autre part de leurs séquences de remplacement sous la forme de machines d'états synchrones. L'équivalence des deux modèles en termes de sémantique est exprimée sous forme de formules de logique temporelle puis vérifiée à l'aide d'un outil de model-checking . Le schéma de vérification ainsi que la modélisation des différents types d'instructions sont présentés en 4.3.

Enfin, le remplacement de chaque instruction par sa séquence de remplacement associée entraîne un surcoût en taille de code ainsi qu'en temps d'exécution. Comme la taille des séquences de remplacement varie selon le type d'instruction, une évaluation du surcoût apporté par la contre-mesure proposée est réalisée sur différentes implémentations en 4.4.

4.2 Présentation du schéma de contre-mesure

Cette section présente de façon détaillée le schéma de contre-mesure introduit dans ce chapitre. Pour cela, une présentation des différentes classes d'instructions est

1. Une instruction idempotente est une instruction dont l'exécution itérée produit le même résultat que l'exécution unique

proposée en 4.2.1. Ensuite, les différentes séquences de remplacement pour chacune de ces classes d'instructions sont présentées en 4.2.2. Enfin, un tableau récapitulatif sur la forme des séquences de remplacement pour chaque type d'instruction est présenté en 4.2.3.

4.2.1 Classes d'instructions

Une séquence de remplacement tolérante aux fautes a été définie pour la plupart des instructions et des encodages du jeu d'instructions Thumb-2 [2]. Pour une grande partie des instructions, la séquence de remplacement est très simple. Néanmoins, cette séquence peut devenir bien plus complexe pour certaines instructions spécifiques. Selon les séquences de remplacement qui ont été proposées, les instructions du jeu Thumb-2 peuvent être regroupées en trois classes. Celles-ci sont résumées dans le tableau 4.1.

Table 4.1. : Classes d'instructions dans le jeu d'instructions Thumb-2

Classe d'instructions	Exemples	Séquence de remplacement
Instructions idempotentes	`mov r1,r8` `add r3,r1,r2`	Duplication de l'instruction

Instructions séparables	`add r1,r1,#1`
	`bl fonction`
	`push {r1,r2,r3}`
		Utilisation de reg. supplémentaires et décomposition en séquence d'instructions idempotentes

Instructions sans séquence de remplacement	`yield` `mcr p0,#0,r8,r2,r3`	Aucune séquence de remplacement proposée

La première classe regroupe les instructions idempotentes qui peuvent donc directement être dupliquées pour permettre une tolérance aux fautes. La seconde classe regroupe les instructions qui ne sont pas idempotentes mais peuvent être remplacées par une séquence équivalente d'instructions idempotentes. La troisième classe regroupe les instructions qui ne peuvent pas être simplement remplacées par une liste d'instructions idempotentes mais néanmoins pour lesquelles une séquence de remplacement spécifique est possible. Enfin, une dernière classe contient les instructions pour lesquelles aucune séquence de remplacement qui assure à la fois une tolérance aux fautes et une exécution correcte dans tous les cas n'a été trouvée. Pour certaines de ces instructions, une séquence permettant de détecter une faute injectée peut être proposée.

4.2.2 Séquences de remplacement par classe d'instructions

Les paragraphes qui suivent donnent davantage de détails à propos de ces classes d'instructions et fournissent également plusieurs exemples de séquences de remplacement pour les différentes classes.

2. Le jeu d'instructions Thumb-2 est présenté de façon plus détaillée en 2.2.1

4.2.2.1. Instructions idempotentes

Les instructions idempotentes sont les instructions qui peuvent être exécutées indifféremment une ou plusieurs fois pour obtenir le même effet [3]. Si toutes les opérandes sources sont différentes des opérandes de destination [4], et si les valeurs écrites dans les opérandes de destination ne dépendent pas de l'adresse de l'instruction dans le code, alors une instruction peut être considérée comme idempotente. Pour de telles instructions, la contre-mesure consiste simplement à réaliser une duplication de l'instruction. Le surcoût pour ce type de duplication est double : un surcoût égal à la taille de l'instruction en termes de taille de code et un surcoût en termes de performance puisque le temps d'exécution est doublé. Le tableau 4.2 donne plusieurs exemples d'instructions idempotentes et de leurs séquences de remplacement.

Table 4.2. : Séquences de remplacement pour quelques instructions idempotentes

Instruction	Description	Remplacement
`mov r1,r8`	Copie `r8` dans `r1`	`mov r1,r8` `mov r1,r8`
`ldr r1,[r8,r2]`	Charge le mot binaire stocké à l'adresse `r8+r2` dans `r1`	`ldr r1,[r8,r2]` `ldr r1,[r8,r2]`
`str r3,[r2,#10]`	Stocke `r3` à l'adresse `r2+10`	`str r3,[r2,#10]` `str r3,[r2,#10]`
`add r3,r1,r2`	Met le résultat de `r1+r2` dans `r3`	`add r3,r1,r2` `add r3,r1,r2`

4.2.2.2. Instructions séparables

Dans le jeu d'instructions considéré, certaines instructions ne sont pas idempotentes dans leur sémantique mais peuvent être transformées sous la forme d'une séquence d'instructions idempotentes dont l'exécution aboutit exactement au même résultat pour les registres de destination et les *flags* du processeur en utilisant un ou plusieurs registres auxiliaires. Une fois que cette transformation est réalisée, chaque instruction idempotente de la séquence de remplacement peut être simplement dupliquée. Comme les registres supplémentaires nécessaires doivent être disponibles à cet emplacement dans le code, tout registre non-vivant [5] peut être utilisé. Dans les conventions d'appel et l'ABI, le registre `r12` est toujours non-vivant à l'entrée d'une fonction, il peut être utilisé pour stocker des valeurs intermédiaires et n'a pas besoin d'être sauvegardé sur la pile. Ainsi, ce registre peut être utilisé (à condition qu'il soit disponible) comme registre temporaire pour de telles séquences de remplacement. Si un nombre insuffisant de registres non-vivants est disponible en début de fonction et

3. Pour la mémoire et tous les registres autres que le compteur de programme
4. En incluant les *flags* du processeur
5. Un registre est dit *vivant* à un certain point dans un programme s'il contient une valeur définie à la suite d'une écriture dans ce registre et est amené à être lu plus tard avant d'être réécrit

si suffisamment de registres peuvent être placés sur la pile, celle-ci peut être utilisée pour stocker temporairement la valeur contenue dans certains registres.

Instructions séparables simples Le listing 4.1 montre la séquence de remplacement pour l'instruction `add r1,r1,r3`. Pour l'instruction `add r1,r1,r3`, un seul registre supplémentaire (noté `rx`) est nécessaire. De plus, 4 instructions sont utilisées au lieu d'une seule initialement, et le surcoût en termes de taille de code est compris entre 6 et 10 octets (selon l'encodage utilisé par l'instruction initiale et celui des instructions de remplacement).

Listing 4.1 : Séquence de remplacement pour l'instruction `add r1, r1, r3`

```
1 ; on suppose que rx est un registre disponible
2 mov rx, r1
3 mov rx, r1
4 add r1, rx, r3
5 add r1, rx, r3
```

Instructions de manipulation de la pile Certaines instructions d'accès mémoire peuvent mettre à jour le registre qui contient l'adresse mémoire à laquelle accéder avant ou après un accès mémoire. Ainsi, l'instruction `stmdb` stocke plusieurs registres en mémoire et décrémente le pointeur mémoire avant chaque accès. Symétriquement, l'instruction `ldmia` charge un segment mémoire dans plusieurs registres et incrémente le pointeur mémoire après chaque accès. Par conséquent, ce registre qui contient l'adresse mémoire à laquelle accéder est à la fois un registre source et un registre destination pour ce type d'instructions. C'est également le cas pour les instructions de manipulation de la pile `push` et `pop`. Celles-ci respectivement écrivent ou lisent sur la pile et décrémentent ou incrémentent le pointeur de pile. La solution consistant à dupliquer les instructions de `push` et de `pop` n'est pas valable. Elle poserait un problème en cas d'injection de faute (par exemple si deux `push` sont réalisés et un seul `pop`) car la pile pourrait se retrouver corrompue. Toutefois, les instructions de manipulation de la pile peuvent être transformées en une séquence d'instructions qui réalise une seule opération à la fois : soit un accès mémoire soit une mise à jour du pointeur de pile. L'instruction `push` peut ainsi être décomposée en une suite d'instructions qui d'abord écrivent les registres à sauvegarder sur la pile et ensuite décrémentent le pointeur de pile. Comme la mise à jour du pointeur de pile implique de lire puis écrire dans le même registre, cette opération est décomposée en deux étapes selon le principe présenté dans le paragraphe précédent. Une telle séquence de remplacement pour l'instruction `push` est présentée dans le listing 4.2. Cette séquence de remplacement nécessite un registre supplémentaires et a un surcoût en termes de taille de code et de performance de 5 instructions.

Listing 4.2 : Séquence de remplacement pour l'instruction `push {r1, r2, r3, lr}`

```
1 ; l'instruction push{} est equivalente a l'instruction stmdb sp !,{}
2 stmdb     sp, {r1, r2, r3, lr}   ; stockage des données sur la pile
```

```
3 stmdb     sp, {r1, r2, r3, lr}
4 sub       rx, sp, #16             ; calcul du nouveau pointeur de pile
5 sub       rx, sp, #16
6 mov       sp, rx                  ; mise à jour du pointeur de pile
7 mov       sp, rx
```

Le registre `lr` permet de stocker un pointeur de retour lors de l'appel à une sous-fonction. Toutefois, lorsqu'une sous-fonction est appelée à l'intérieur d'une autre sous-fonction, ce registre ne peut contenir qu'un seul des deux pointeurs de retour. Dans ce cas, le compilateur choisit généralement de stocker l'autre pointeur de retour sur la pile. Pour cela, une instruction `push` qui stocke le registre `lr` sur la pile est utilisée en début de fonction. Ainsi, à la fin d'une fonction dont le pointeur de retour est stocké sur la pile, une instruction `pop` est utilisée pour la restauration du contexte et le branchement vers le pointeur de retour. Par exemple, dans le cas d'une fonction pour laquelle l'instruction `push {r1,r2,r3,lr}` a été utilisée pour la sauvegarde du contexte, l'instruction `pop {r1, r2, r3, pc}` sera utilisée en fin de fonction. Dans ce cas, la séquence de remplacement proposée précédemment pour une instruction `pop` doit être modifiée pour permettre de réaliser l'opération de mise à jour du pointeur de pile. Un exemple d'une telle séquence de remplacement modifiée pour l'instruction `pop {r1,r2,r3,pc}` est présentée sur le listing 4.3.

Listing 4.3 : Séquence de remplacement pour l'instruction `pop {r1, r2, r3, pc}`

```
1 ; l'instruction pop{} est equivalente a l'instruction ldmia sp !,{}
2 ldmia     sp, {r1, r2, r3, lr} ; lecture des données depuis la pile
3 ldmia     sp, {r1, r2, r3, lr}
4 add       rx, sp, #16             ; calcul du nouveau pointeur de pile
5 add       rx, sp, #16
6 mov       sp, rx                  ; mise à jour du pointeur de pile
7 mov       sp, rx
8 bx        lr                      ; branchement de sortie de la fonction
9 bx        lr
```

Instruction `umlal` L'instruction `umlal`[6] multiplie les entiers relatifs stockés dans deux registres sources de 32 bits et leur ajoute le contenu de la concaténation des deux registres de destination. Le résultat est alors écrit dans les deux registres de destination. Ainsi, cette instructions a deux registres qui sont à la fois source et destination. Toutefois, elle peut être décomposée en une suite d'instructions qui comprend d'abord une instruction de multiplication dont le résultat est une valeur sur 64 bits. Ensuite, l'addition sur 64 bits doit elle même être décomposée. Pour cela, il est nécessaire d'additionner d'abord les 32 bits de poids faible entre-eux en propageant la retenue à l'aide d'une instruction `adds` qui écrit le flag correspondant à la retenue. Ensuite, les 32 bits de poids fort peuvent être additionnés entre-eux à l'aide d'une instruction `adc` qui utilise le flag de retenue écrit précédemment.

6. Unsigned Multiply Accumulate Long

Néanmoins, l'instruction `adds` écrit l'ensemble des *flags* alors que l'instruction initiale `umlal` ne le fait pas. Les *flags* qui étaient supposés être lus par une instruction qui suit pourraient alors avoir été écrasés par la séquence de remplacement. Dans ce cas, il n'y aurait plus d'équivalence entre l'instruction `umlal` et une telle séquence de remplacement. Par conséquent, pour garder cette équivalence, il est indispensable de sauvegarder les *flags* avant la séquence de remplacement et de les restaurer après. Le fait de sauvegarder les *flags* requiert 4 instructions supplémentaires. La séquence de remplacement complète pour l'instruction `umlal` est présentée dans le listing 4.4. Elle impose l'utilisation de 4 registres supplémentaires et remplace l'instruction initiale par 14 instructions. Ainsi, il s'avère que cette séquence de remplacement est la plus coûteuse de tout le jeu d'instructions Thumb-2, à la fois en termes de registres supplémentaires et d'instructions à ajouter.

Listing 4.4 : Séquence de remplacement pour l'instruction `umlal rlo, rhi, rn, rm`

```
1  ; umlal rlo,rhi,rn,rm réalise l'opération rhi :rlo = rn*rm + rhi :rlo
2  mrs   rt, apsr          ; sauvegarde des flags dans un registre
3  mrs   rt, apsr
4  umull rx, ry, rn, rm    ; utilisation de deux registres supplémentaires
5  umull rx, ry, rn, rm
6  adds  rz, rx, rlo       ; addition des 32 bits de poids faible
7  adds  rz, rx, rlo
8  addc  rx, ry, rhi       ; addition des 32 bits de poids fort
9  addc  rx, ry, rhi
10 mov   rlo, rz           ; utilisation d'un registre temporaire
11 mov   rlo, rz
12 mov   rhi, rx           ; utilisation d'un registre temporaire
13 mov   rhi, rx
14 msr   apsr, rt          ; restauration des flags
15 msr   apsr, rt
```

Instructions avec un décalage constant Dans le jeu d'instructions Thumb-2, plusieurs décalages constants peuvent être appliqués à l'opérande de certaines instructions avant que cette opérande ne soit traitée par l'instruction. Ces décalages sont `lsl` (décalage logique à gauche), `lsr` (décalage logique à droite), `asr` (décalage arithmétique à droite), `ror` (rotation à droite) et `rrx` (rotation à droite avec utilisation du flag de retenue). Parmi ceux-ci, le décalage `rrx` décale tous les bits du registre concerné d'un bit vers la droite et utilise le flag de retenue pour mettre à jour le bit de poids fort. Ainsi, une telle opération réalise une lecture du flag de retenue. Si l'instruction initiale qui utilise un décalage `rrx` écrit également les flags, alors celle-ci doit être décomposée. Un exemple de remplacement pour une instruction `subs r1,r2,r3,rrx` (qui applique ce décalage au registre `r3`) est proposé dans le listing 4.5.

Listing 4.5 : Séquence de remplacement pour une instruction `subs r1, r2, r3, rrx`

```
1  rrx   ry, r3
```

```
2 rrx  ry, r3
3 subs r1, r2, ry
4 subs r1, r2, ry
```

Instruction d'appel de sous-fonction `bl` L'instruction d'appel de sous-fonction `bl` réalise un branchement et écrit l'adresse de retour dans le registre `lr` (`r14`). À proprement parler, il n'y a aucun registre à la fois source et destination pour l'instruction `bl`, celle-ci ne rentre donc pas dans la catégorie des instructions séparables définie précédemment en 4.2.2.2. Pour autant, il est impossible de dupliquer cette instruction : cela entraînerait une double exécution de la sous-fonction si aucune attaque n'est réalisée. Une solution possible consiste à d'abord fixer le pointeur de retour dans le registre `lr` et ensuite réaliser un branchement inconditionnel. Comme le mode d'exécution Thumb sur les processeurs ARM requiert que le dernier bit d'un pointeur de retour soit mis à `1`, l'opération sur ce bit doit être ajoutée avant le branchement inconditionnel vers la sous-fonction. La séquence de remplacement proposée pour le remplacement de l'instruction `bl` est présentée dans le listing 4.6.

Listing 4.6 : Séquence de remplacement pour une instruction `bl`

```
1     adr r12, label_retour  ; enregistrement du pointeur de retour dans r12
2     adr r12, label_retour
3     add lr, r12, #1        ; lr=r12+1, le dernier bit du pointeur doit etre
4     add lr, r12, #1        ; mis a 1 pour le mode d'execution Thumb
5     b   fonction           ; branchement vers la fonction
6     b   fonction
7 label_retour
```

Première possibilité de remplacement pour les blocs `it` Le listing 4.7 présente un exemple de bloc *If-Then* `it` permettant une exécution conditionnelle. Ce bloc `it` est similaire à celui présenté en 2.2.1. La solution la plus simple pour ce type de bloc est de d'abord transformer le bloc `it` en une structure classique de type *if-then-else* à base de branchements (comme celle présentée dans le listing 4.8) puis d'appliquer le schéma de contre-mesure à chaque instruction (comme présenté dans le listing 4.9)

Listing 4.7 : Exemple de bloc `it`

```
1 itte NE
2 addne r1, r2, #10
3 eorne r3, r5, r1
4 moveq r3, #10
```

Listing 4.8 : Code équivalent au bloc `it` du listing

```
1     b.eq label_else
2     add r1, r2, #10
3     eor r3, r5, r1
4     b label_suite
```

```
label_else
    mov r3, #10
label_suite
```

Listing 4.9 : Code du listing renforcé avec la contre-mesure proposée

```
    b.eq label_else
    b.eq label_else
    add r1, r2, #10
    add r1, r2, #10
    eor r3, r5, r1
    eor r3, r5, r1
    b label_suite
    b label_suite
label_else
    mov r3, #10
    mov r3, #10
label_suite
```

Seconde possibilité de remplacement pour les blocs `it` Une autre possibilité de mode de remplacement pour les blocs `it` peut également être proposée. Le listing 4.10 donne un exemple de bloc `it` auquel ce remplacement alternatif va pouvoir être appliqué. La solution proposée consiste à d'abord appliquer le schéma de contre-mesure à chaque instruction du bloc `it`. Chaque instruction de la séquence de remplacement reçoit initialement la même condition que l'instruction qu'elle remplace. Cette étape est présentée dans le listing 4.11. La deuxième étape consiste à ajouter plusieurs instructions `it` de façon à recréer plusieurs blocs. Ces blocs sont constitués de deux instructions `it` consécutives, suivies d'au maximum 3 instructions, comme illustré dans le listing 4.12. Une telle construction correspond donc à un bloc `it` de 4 instructions dans lequel se trouve un bloc `it` de 3 instructions [7]. Enfin, les conditions définies dans les instructions `it` doivent être mises à jour pour respecter les conditions d'exécution des instructions qui les suivent, comme illustré dans le listing 4.13.

Listing 4.10 : Exemple de bloc `it`

```
itte  NE              ; une instruction IT avec la condition NE
addne r1, r2, #10
eorne r3, r5, r1
moveq r3, #10
```

Listing 4.11 : Première étape de remplacement pour le bloc `it` du listing

```
addne r1, r2, #10     ; les instructions du bloc it sont dupliquees
addne r1, r2, #10     ; en gardant leur condition d'execution
eorne r3, r5, r1
```

7. La spécification du jeu d'instructions Thumb-2 précise que les blocs `it` ne peuvent pas définir d'exécution conditionnelle pour plus de 4 instructions

```
4  eorne r3, r5, r1
5  moveq r3, #10
6  moveq r3, #10
```

Listing 4.12 : Deuxième étape de remplacement pour le bloc `it` du listing

```
1  i???? NE              ; deux instructions it sont insérées toutes les 3
2  i???  NE              ; instructions pour creer de nouveaux blocs
3  addne r1, r2, #10
4  addne r1, r2, #10
5  eorne r3, r5, r1
6  i???? NE
7  i???  NE
8  eorne r3, r5, r1
9  moveq r3, #10
10 moveq r3, #10
```

Listing 4.13 : Dernière étape de remplacement pour le bloc `it` du listing

```
1  itttt NE              ; les conditions des blocs it sont mises a jour
2  ittt  NE              ; selon les conditions d'execution des inst. du bloc
3  addne r1, r2, #10
4  addne r1, r2, #10
5  eorne r3, r5, r1
6  ittee NE
7  itee  NE
8  eorne r3, r5, r1
9  moveq r3, #10
10 moveq r3, #10
```

Il convient de noter qu'un bloc `it` n'est pas supposé apparaître dans un autre bloc `it`. Néanmoins, d'après un test expérimental réalisé sur le microcontrôleur visé, la seconde instruction `it` n'a aucun effet et est exécutée comme un `nop`. En cas de faute qui vise une des instructions `it` dupliquées, le programme agit comme s'il n'y avait eu qu'une seule instruction `it`. Sinon, la seconde instruction `it` est exécutée dans le bloc défini par la première instruction `it`.

Cette seconde méthode pour le remplacement des blocs `it` présente l'inconvénient d'être plus complexe à mettre en œuvre que la première car elle nécessite de reconstruire de nouveaux blocs `it` en tenant compte de la duplication des instructions à l'intérieur de ceux-ci. Néanmoins, elle présente l'avantage de ne pas être basée sur des instructions de branchement (qui obligent à vider le pipeline). Cette seconde solution semble donc à privilégier en vue de minimiser le temps d'exécution et donc le surcoût en performances apporté par la contre-mesure.

4.2.2.3. Instructions sans séquence de remplacement proposée

Cette troisième classe regroupe les instructions pour lesquelles aucune séquence de remplacement qui assure à la fois une tolérance aux fautes et une sémantique

équivalente à l'instruction initiale dans tous les cas n'a été trouvée. Cette classe comprend :

— les instructions qui lisent et écrivent les *flags* simultanément (il est possible de définir des séquences de remplacement sous certaines conditions sur la lecture ultérieure des *flags*)

— des instructions pour lesquelles aucune séquence de remplacement n'a été proposée en raison de leur spécificité

Instructions qui lisent et écrivent les flags Les instructions qui lisent et écrivent les *flags* posent un certain nombre de problèmes pour un éventuel remplacement par une séquence d'instructions tolérante aux fautes. Par exemple, l'instruction `adcs` réalise une addition entre deux opérandes source (deux registres ou bien un registre et une valeur immédiate) et le *flag* de retenue. Le résultat est écrit dans un registre destination et les *flags* (retenue, négatif, dépassement et zero) sont mis à jour. Le fait de dupliquer une telle instruction pose problème, car la seconde instruction `adcs` serait amenée à utiliser la retenue écrite par la première instruction `adcs` au lieu de la retenue initiale. Si les *flags* sont vivants après l'instruction `adcs`, alors nous n'avons trouvé aucune séquence de remplacement et le code doit être modifié. Sinon, dans le cas où les *flags* ne sont pas vivants après l'instruction `adcs`, alors une telle séquence peut être définie. Pour cela, il est possible de sauvegarder la valeur des *flags* avant la première instruction `adcs` et ensuite de restaurer leur valeur avant la seconde instruction `adcs`. Une telle séquence de remplacement est présentée dans le listing 4.14

Listing 4.14 : Séquence de remplacement pour une instruction `adcs r1, r2, r3`

```
1 mrs   rx, apsr   ; sauvegarde des flags dans un registre temporaire
2 mrs   rx, apsr
3 adcs  r1, r2, r3
4 msr   apsr, rx   ; restauration des flags
5 msr   apsr, rx
6 adcs  r1, r2, r3
```

Autres instructions sans séquence de remplacement proposée Pour certaines instructions, il n'est pas possible de définir une séquence de remplacement car celles-ci réalisent une interaction avec des circuits externes. Ces instructions regroupent les commandes pour les coprocesseurs (`mcr`, `lcr`, ...) ou les instructions de synchronisation avec des éléments externes (`sev`, `yield`, ...). Néanmoins, il est important de mentionner qu'il peut être possible de définir des séquences de remplacement pour ces instructions si les coprocesseurs ou éléments externes avec lesquelles celles-ci interagissent sont adaptés en conséquence.

4.2.3 Bilan sur les classes d'instructions définies

Un récapitulatif des différentes classes d'instructions qui ont été définies est présenté dans le tableau 4.3. Ce tableau montre les différents types d'instructions inclus dans chaque classe et donne quelques exemples pour chaque type d'instructions.

Table 4.3. : Bilan sur les classes d'instructions définies

Classe	Type d'instructions	Exemple
Instructions idempotentes	Opérations ALU	`add r1,r2,r3` - `subs r1,r2,#8`
	Inst. de *load*	`ldrh r1,[r2,r3]` - `ldrb r1,[r2,#8]`
	Inst. de *store*	`strh r1,[r2,r3]` - `strb r1,[r2,#8]`
	Inst. de branchement	`b label` - `bx lr`
	Inst. de comparaison	`cmp r1,#9` - `cne r3,r4`
Instructions séparables	Inst. avec un registre source et destination	`add r1,r1,#1` - `str r0,[r0,#0]`
	Inst. avec adressage pré ou post-indexé	`str r1,[r2,#8]!` - `str r1,[r2],#8`
	Inst. de manipulation de la pile	`push {r1,r2}` - `pop {r1,r2}`
	Inst. avec décalage `rrx` et écriture des *flags*	`adds r1,r2,r3,rrx`
	Inst. d'appel de sous-fonction	`bl fonction`
	Blocs If-Then (`it`)	`itte NE`
Instructions sans séquence de remplacement	Inst. de synchronisation avec des syst. externes	`sev` - `yield` - `svc`
	Inst. pour coprocesseurs	`mcr p0,#0,r8,r2,r3`
	Inst. qui lisent et écrivent les flags	`adcs r1,r2,r3` - `rrxs r1,r2`

4.3 Vérification formelle du schéma de contre-mesure

Dans les paragraphes qui suivent, l'approche utilisée pour vérifier formellement la spécification de tolérance aux fautes des séquences de remplacement est présentée. Pour commencer, un bref préambule sur la vérification formelle et le model-checking est proposé en 4.3.1. Ce préambule présente également quelques autres utilisations de techniques de vérification formelle dans le domaine de la sécurité des systèmes embarqués. Ensuite, le principe de vérification et la modélisation utilisés pour la vérification sont présentés en 4.3.2. Enfin, des exemples de vérification de séquences de remplacement sont présentés en 4.3.3.

4.3.1 Préambule sur la vérification formelle

Les paragraphes qui suivent commencent par faire une brève présentation sur les techniques de model-checking et décrivent ensuite plusieurs travaux récents issus de la littérature scientifique qui ont utilisé des techniques de vérification formelle dans le domaine de la sécurité des systèmes embarqués.

4.3.1.1. Approche par model-checking

Le model-checking regroupe un ensemble de techniques de vérification formelle qui permettent de démontrer si un système satisfait ou non une spécification fonctionnelle. Le système est exprimé sous la forme d'une machine d'états et les spécifications sous la forme de formules de logique temporelle. Les logiques temporelles utilisées incluent principalement la logique temporelle arborescente, ou Computation Tree

Logic (CTL), ainsi que la logique temporelle linéaire, ou Linear Temporal Logic (LTL) (BAIER et KATOEN, 2008). Les programmes de model-checking (également appelés *model-checkers*) construisent un graphe orienté des états du système à partir de la représentation utilisée en entrée. Ils évaluent ensuite la validité des propositions logiques considérées pour ce graphe d'états. La principale limitation des techniques de model-checking est intrinsèque à leur mode de résolution qui parcourt l'espace d'états du système à analyser. Les techniques de réduction d'ordre partiel, les approches à base de SAT puis plus récemment de Satisfiability Modulo Theories (SMT) combinées à des techniques d'abstraction toujours plus pertinentes permettent de repousser les limites de ces approches. En pratique, les outils actuels à base de Binary Decision Diagram (BDD) ou SAT peuvent traiter automatiquement des systèmes modélisés par quelques centaines voire quelques milliers de variables booléennes.

Dans la suite de ce chapitre, les techniques de model-checking sont utilisées pour vérifier l'équivalence en termes de sémantique entre une instruction initiale et sa séquence de remplacement, ainsi que pour garantir la robustesse des séquences de remplacement proposées. Pour cela, instructions et séquences de remplacement sont modélisées sous la forme de machines d'états synchrones sur lesquelles des formules de logique temporelle sont évaluées. Les techniques de model-checking à base de BDD ou SAT s'avèrent suffisantes pour modéliser l'effet de quelques instructions assembleur sur des registres et zones mémoire présentant un nombre limité de bits.

4.3.1.2. Autres travaux de vérification formelle

Les méthodes formelles et autres outils de preuve ont été utilisés dans plusieurs cas pour vérifier qu'une implémentation répondait bien à des spécifications fonctionnelles en termes de sécurité issues des Critères Communs (CHETALI et Q.-h. NGUYEN, 2008). Les paragraphes qui suivent présentent quelques approches formelles proposées récemment dans le cas d'attaque par injection de fautes, ainsi qu'une modélisation formelle de l'architecture ARMv7.

Logique de Hoare Dans l'article de Meola *et al.* (MEOLA et WALKER, 2010), les auteurs proposent de modéliser une injection de fautes en utilisant un langage simplifié de haut niveau et de la logique de Hoare. En modélisant ainsi un code, il devient possible d'effectuer des vérifications et d'élaborer des preuves de sécurité. Toutefois, le langage simplifié utilisé ne permet pas de modéliser des opérations complexes comme l'utilisation de pointeurs. Il ne permet donc pas de représenter un grand nombre de programmes réels. L'approche présentée dans cet article peut donc seulement permettre de réaliser des preuves de sécurité sur certaines portions sensibles de code qui utilisent seulement les opérations que peut modéliser ce langage simplifié.

Vérification sur un code source C L'article de Christofi *et al.* (CHRISTOFI et al., 2013) s'intéresse à la vérification formelle d'une contre-mesure face aux attaques par injection de faute. L'article se focalise sur le cas de l'injection d'une seule faute, mais la méthode présentée pourrait être généralisée au cas des attaques par injection de fautes multiples à plusieurs instants. Un modèle de fautes pour lequel un attaquant peut forcer à 0 des données écrites en mémoire ou modifier des données écrites dans des registres mais pas corrompre l'exécution des instructions a été utilisé. Dans cet article, les auteurs utilisent l'analyseur de programmes Frama-C. Frama-C[8] est une plate-forme dédiée à l'analyse statique de codes sources écrits en C. L'utilisateur de Frama-C peut décrire des spécifications fonctionnelles à l'aide d'un langage dédié, puis prouver que le code source testé satisfait ces spécifications. Frama-C est utilisé dans cet article pour vérifier la robustesse d'une implémentation de l'algorithme CRT-RSA renforcée à l'aide de la contre-mesure de Vigilant (VIGILANT, 2008). L'étude présentée dans cet article a permis de détecter un ensemble de fautes qui ne sont pas détectées par les contre-mesures de l'implémentation.

Vérification formelle de contre-mesures sur CRT-RSA Une autre contribution de méthodologie formelle au niveau d'un algorithme a été proposée par Rauzy *et al.* (RAUZY et GUILLEY, 2013). Dans cet article, les auteurs vérifient formellement la résistance aux fautes de plusieurs contre-mesures sur une implémentation de l'algorithme CRT-RSA face à des attaques par injection de faute similaires à celles présentées dans (BONEH et al., 1997). Cette implémentation est décrite dans un langage de haut niveau. Dans le modèle de fautes considéré, un attaquant peut induire des fautes au niveau du flot de données. Contrairement aux travaux présentés dans ce chapitre, les contre-mesures étudiées ont été définies au niveau algorithmique et visent à protéger l'algorithme CRT-RSA. Leur étude a permis de trouver plusieurs vulnérabilités dans la contre-mesure proposée par Shamir (SHAMIR, 1999) ou de montrer la résistance de la contre-mesure proposée par Aumüller *et al.* face aux attaques basées sur l'injection d'une seule faute. Ce travail a ensuite été poursuivi dans (RAUZY et GUILLEY, 2014), où la contre-mesure de Vigilant (VIGILANT, 2008) a été davantage étudiée. Dans cet autre article, les auteurs retrouvent à l'aide de méthodes formelles des vulnérabilités connues sur la contre-mesure de Vigilant mais montrent ensuite que sa version corrigée proposée par Coron *et al.* (CORON et al., 2010) contient des tests non nécessaires et peut être simplifiée.

Tolérance aux fautes de programmes complexes Pattabiraman *et al.* (PATTABIRAMAN et al., 2013) proposent une structure de model-checking pour vérifier la tolérance aux fautes de programmes complexes. Par rapport à l'analyse proposée dans ce chapitre, leur étude se place au niveau du programme assembleur complet (et non pas à l'échelle de l'instruction) et en étudie les sorties. Les programmes embarqués à vérifier sont représentés à l'aide d'un langage assembleur générique. Un modèle

8. http://www.frama-c.com

de fautes au niveau assembleur, qui est la conséquence d'un ensemble d'erreurs au niveau du matériel, est considéré. Par exemple, les erreurs considérées incluent des corruptions au niveau du décodage d'instruction ou du bus de données. Ces erreurs entraînent par exemple des fautes au niveau des registres ou de la mémoire, et ce sont ces fautes au niveau assembleur qui sont modélisées dans leur approche. Elles se rapprochent du modèle présenté dans le chapitre 3. La méthode proposée dans cet article vise ensuite à vérifier l'efficacité ou à trouver des vulnérabilités dans les mécanismes de détection d'erreurs. Enfin, les auteurs proposent une application de leur méthodologie sur le logiciel TCAS [9] et arrivent à détecter des cas où une faute injectée à un endroit du code peut mener à une sortie inacceptable pour le programme.

Analyse de robustesse de circuits soumis à des fautes transitoires Dans (Goerschwin FEY et DRECHSLER, 2008), les auteurs proposent une méthodologie permettant d'évaluer automatiquement la robustesse d'un circuit soumis à des fautes transitoires. Pour cela, le circuit à évaluer est décomposé en un ensemble de composants, dont la fonctionnalité est représentée par une fonction booléenne. Ensuite, trois modèles de fautes sont considérés et modélisent différents types de fautes transitoires. Le mécanisme de vérification formelle repose sur l'équivalence du comportement du système fauté avec le système de référence via la résolution d'un problème de type SAT. L'approche présentée dans cet article évalue l'impact qu'a une faute injectée dans un des composants sur la sortie globale du circuit, et permet ainsi de repérer les éléments potentiellement vulnérables du circuit. Ces travaux ont été poursuivis dans (Görschwin FEY et al., 2011), et le schéma de vérification formelle qui est présenté dans ce chapitre peut être vu comme une transposition au niveau assembleur du schéma présenté dans cet article. Dans (BAARIR, BRAUNSTEIN, CLAVEL et al., 2009), les auteurs étudient également le cas de la tolérance aux fautes de circuits synchrones décrits au niveau RTL en considérant un modèle de fautes transitoires de type *bit-flip*. Pour cela, ils utilisent à la fois un assistant de preuve pour réaliser des vérifications formelles sur la robustesse générale du circuit ainsi qu'un outil de model-checking pour obtenir des résultats quantifiés sur son degré de robustesse. Dans (BAARIR, BRAUNSTEIN, ENCRENAZ et al., 2011), les auteurs étudient également le cas de la tolérance de circuits synchrones face à des fautes transitoires de type *bit-flip* mais étudient plus spécifiquement les capacités d'auto-réparation d'un circuit (définies comme les capacités d'un circuit à retrouver un état normal à la suite d'une injection de faute dans l'un de ses éléments). Leur approche permet, à l'aide d'outils de vérification de type SAT et BDD, de repérer un ensemble d'éléments du circuit qui pourraient être renforcés ainsi que de comparer plusieurs implémentations d'un même circuit afin de sélectionner celles dont l'auto-réparation est la plus rapide.

9. Traffic Collision Avoidance System, système visant à éviter les collisions entre aéronefs

Modélisation formelle de l'architecture ARMv7 Dans (Fox et Myreen, 2010), Fox *et al.* ont également proposé un formalisme pour l'architecture ARMv7 basée sur l'assistant de preuve HOL4. Ce formalisme permet de modéliser la sémantique de chacune des instructions et de générer des preuves pour vérifier la sémantique de séquences de code. Un tel outil aurait également pu être utilisé pour vérifier formellement la contre-mesure proposée. Toutefois, les preuves générées par HOL4 sont bien plus complexes que celles nécessaires pour réaliser une équivalence de modèles finis et la vérification de propriétés simples sur ces modèles comme dans notre cas.

4.3.2 Modélisation et spécification à prouver

Un programme peut être vu comme l'application d'une suite de transformations des valeurs stockées dans un ensemble de registres ou dans la mémoire. Chaque instruction du programme agit ainsi comme une fonction qui prend une configuration des registres et de la mémoire en entrée et qui produit une nouvelle configuration des mêmes éléments en sortie. Le programme peut alors être représenté sous la forme d'un système de transitions dont les états correspondent aux configurations des registres et de la mémoire, et dans lequel chaque transition imite la transformation réalisée par l'exécution d'une instruction assembleur.

Au lieu de vérifier formellement la tolérance aux fautes pour un programme complet, l'approche proposée dans ce chapitre consiste à prouver la tolérance aux fautes pour chaque séquence de remplacement proposée précédemment. En effet, il est suffisant de certifier que l'état de sortie (caractérisé par l'état de la mémoire et des registres) après l'exécution d'une séquence de remplacement (avec ou sans injection de faute) est équivalent à l'état de sortie attendu après l'exécution de l'instruction initiale. Comme cet état de sortie correspond à l'état d'entrée pour l'instruction qui suit, une telle approche de vérification permet de garantir que l'instruction suivante utilisera une configuration correcte pour les registres et la mémoire. De plus, en divisant ainsi le problème il est ainsi possible de contourner la principale limitation des approches par model-checking liée à l'explosion du nombre d'états.

4.3.2.1. Modélisation d'une suite d'instructions par une machine d'états

Comme expliqué précédemment, il est possible de modéliser l'exécution d'une séquence d'instructions sous la forme d'un système de transitions m défini comme un automate fini interprété. L'automate interagit avec un espace de données externe (noté ici D), qui conditionne le franchissement des transitions de l'automate. En retour, le franchissement des transitions modifie les valeurs des éléments stockés dans cet espace de données externe. Cet espace de données est constitué des éléments mémorisants du programme (registres à l'exception du compteur de programme `pc` et *flags*). Les paragraphes qui suivent présentent les différents éléments modélisés.

Système de transitions Dans ce chapitre, le système de transitions m est un quintuplet $m = (S, T, S_0, S_f, L)$, avec :

— S l'ensemble des états

— T l'ensemble des transitions $T : S \to S$

— S_0 le sous-ensemble de S qui regroupe les états initiaux

— S_f le sous-ensemble de S qui regroupe les états finaux

— L un ensemble d'étiquettes qui correspondent aux valeurs que le compteur de programme peut prendre (donc aux adresses des instructions du programme)

Les états finaux de S_f sont des états absorbants [10].

Espace de données Chaque état est associé à une configuration de l'espace de données D. Une configuration $d \in D$ se caractérise par :

— la valeur des registres utilisés par le programme, regroupés dans un ensemble R de registres [11]. L'ensemble des configurations de R est noté χ_R.

— la valeur des *flags* du processeur, regroupés dans un ensemble F. L'ensemble des configurations de F est noté χ_F.

Une configuration de l'espace de données $d \in D$ est donc un élément du produit cartésien $(\chi_R * \chi_F)$.

Transitions de l'automate Une transition de l'automate modélise l'effet d'une instruction sur les éléments de D et sur le compteur de programme pc. Chaque transition possède une garde et une action.

Garde Une garde est une condition dépendante des valeurs des éléments de D, et régissant le franchissement d'une transition. La transition est franchissable si l'expression de la garde, évaluée sur la configuration courante de D, s'évalue en $True$.
$g : Config(D) \to \mathbb{B}$

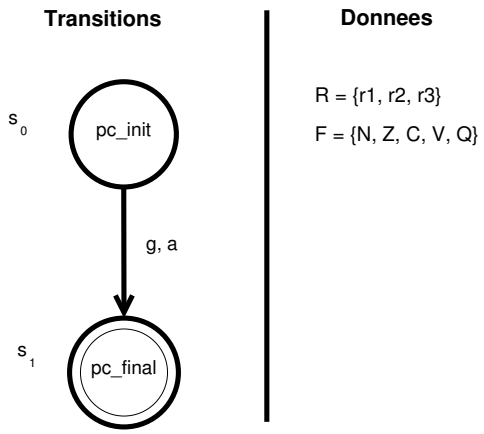
Action Une action modélise l'effet d'une instruction sur chacune des variables de l'espace de données D.
$a = \bigcup_{e \in D} f_e$ avec $f_e : Config(D) * S \to Config(D)$

Exemple de système de transitions Un exemple d'un tel système de transitions pour l'instruction `add r1,r2,r3` est présenté sur la figure 4.1.

4.3.2.2. Modélisation des éléments mémorisants

Les paragraphes qui suivent présentent les choix qui ont été faits pour mémoriser les registres, les flags ainsi que la mémoire.

10. Un état absorbant est un état que le système ne quitte plus une fois l'état atteint
11. Par exemple, pour une instruction `add r1,r2,r3`, l'ensemble R des registres comprend les registres `r1`, `r2`, `r3`



Avec :
$pc\_init, pc\_final \in L$
pc_init correspond à l'adresse de l'instruction `add r1, r2, r3`
pc_final correspond à l'adresse de l'instruction suivante
$S_0 = \{s_0\}$
$S_f = \{s_1\}$
Garde (g) : $[true]$
Action (a) : $\begin{cases} R'.r1 = R.r2 + R.r3 \\ F' = F \end{cases}$

Figure 4.1. : Système de transitions pour l'instruction `add r1,r2,r3`

Modélisation des registres et des flags Les registres de l'ensemble R peuvent comprendre des registres génériques (`r0`-`r12`), le pointeur de pile (`r13`) ou le *Link Register* (`r14`). Ces registres sont modélisés sous la forme de registres 4 bits. Les 5 *flags* du processeur sont : C (*carry*), N (*negative*), Z (*zero*), V (*overflow*), Q (*saturation*) [12]. Chaque *flag* est modélisé sous la forme d'un registre 1 bit. Ces tailles de registres suffisent à modéliser les opérations arithmétiques et logiques ainsi que les opérations sur les *flags* et permettent surtout de garder une complexité raisonnable pour le programme de model-checking. De plus, modéliser l'ensemble des registres n'est pas nécessaire étant donné qu'une instruction lit seulement un sous-ensemble des registres et écrit dans les registres de destination. De plus, d'après le modèle de fautes par saut d'instructions qui a été considéré, les registres qui ne sont pas modifiés par une instruction ne peuvent pas l'être par une faute. Ainsi, pour une instruction i ou sa séquence de remplacement c choisis, l'ensemble R des registres est simplement composé des registres qui sont manipulés par i ou par sa séquence de remplacement c. Les registres supplémentaires utilisés par c sont supposés être non-vivants après l'occurrence de l'instruction i dans le programme initial.

Modélisation de la mémoire Le modèle de fautes considéré suppose que la mémoire ne peut pas être directement corrompue par une faute. Ainsi, s'il n'est pas nécessaire

12. Ces *flags* peuvent être écrits par certaines instructions et sont utilisés par plusieurs autres. Les branchements conditionnels font partie des instructions qui utilisent ces *flags*

de modéliser la mémoire, il est quand même impératif de vérifier que les transferts vers et depuis la mémoire ont bien eu lieu dans le modèle initial et la séquence de remplacement. Pour s'assurer qu'une écriture mémoire a bien été exécutée à la bonne adresse, il suffit de vérifier que l'instruction correspondante a bien été exécutée au moins une fois. Comme expliqué plus tard dans cette partie, on ajoute pour cela une variable qui sert de compteur dans les systèmes de transitions de l'instruction initiale et de sa séquence de remplacement.

4.3.2.3. Construction des systèmes de transition

Pour vérifier qu'une séquence de remplacement pour une instruction est robuste face à une attaque par saut d'instruction, on construit donc deux systèmes de transition : un pour l'instruction initiale et un deuxième pour sa séquence de remplacement. Soit $m(i)$ le système de transitions de l'instruction i sur l'espace de données D^i et $m(c)$ le système de transitions de sa séquence de remplacement sur l'espace de données D^c.

On a donc :

— $m(i) = (S^i, T^i, S_0^i, S_f^i, L^i)$

— $m(c) = (S^c, T^c, S_0^c, S_f^c, L^c)$

Par ailleurs, soit $pc\_init\_i \in L^i$ l'étiquette correspondant à la valeur du compteur de programme lors de l'état initial pour l'instruction i et $pc\_init\_c \in L^c$ celle pour la séquence c. De même, soit $pc\_final\_i \in L^i$ l'étiquette correspondant à la valeur du compteur de programme lors de l'état final pour l'instruction i et $pc\_final\_c \in L^c$ celle pour la séquence c.

Construction du modèle pour l'instruction de référence Pour l'instruction i à renforcer, le système de transitions est construit selon les principes présentés en 4.3.2.1 et 4.3.2.2. On note $GenereSystemeTransitions(i)$ la procédure qui construit ce système de transitions pour une instruction i.

$$m(i) = GenereSystemeTransitions(i) \tag{4.1}$$

Construction du modèle pour la séquence de remplacement avec injection de faute Le système de transitions correspondant à la séquence de remplacement c (correspondant à l'instruction i) est également construit selon les principes présentés en 4.3.2.1 et 4.3.2.2. Cependant, des transitions sont ajoutées au modèle pour représenter un possible saut d'instruction par un attaquant. Celles-ci sont ajoutées à l'aide d'une procédure $AddNop()$, présentée dans l'algorithme 3.

$$m(c) = AddNop(GenereSystemeTransitions(c)) \tag{4.2}$$

Algorithme 3 : $AddNop(m(c))$

1 Ajouter $nbfaute = 0$ dans D^c
2 **foreach** *t transition de $m(c)$ ($t : s_i \rightarrow s_j$)* **do**
3 Créer une nouvelle transition t' ($t : s_i \rightarrow s_j$) avec :
4 - garde : [nbfaute = 0]
5 - action : $\begin{cases} \forall e \in D^c \backslash \{nbfaute\} \\ e' = e \\ nbfaute' = nbfaute + 1 \end{cases}$

Construction du modèle complet Le modèle complet est une composition synchrone des deux systèmes de transitions, avec une mise en cohérence des valeurs initiales des éléments de D^i et D^c. Ce modèle complet est un système de transitions $m = (S, T, S_0, S_f, L)$ sur l'espace de données D, avec :

— $D = D^i \uplus D^c$
— $S \subset (S^i * S^c)$
— $T \subset (T^i * T^c)$
— $S_0 = (S_0^i * S_0^c)$
— $S_f = (S_f^i * S_f^c)$
— $L = L^i \uplus L^c$
— $\forall e_i \in D^i$ et $e_c \in D^c$ (équivalent de e_i dans D^c), e_i et e_c sont initialisés à la même valeur

Pour chaque initialisation d'un couple (e_i, e_c), on utilise une variable libre différente.

Par exemple :
$\begin{cases} e_i^1 = x \\ e_c^1 = x \text{ (la même valeur)} \end{cases}$
mais
$\begin{cases} e_i^2 = y \\ e_c^2 = y \text{ (y peut avoir une valeur différente de x)} \end{cases}$
Un état de m est composé des états des automates $m(i)$ et $m(c)$ et une transition de m correspond au franchissement simultané d'une transition dans $m(i)$ et dans $m(c)$ avec des actions cohérentes sur D^i et D^c. On peut donc chercher à vérifier des propriétés de logique temporelle sur le graphe des états accessibles engendré par m.

4.3.2.4. Spécification à prouver

Pour prouver l'équivalence entre la sortie d'une instruction et celle de sa séquence de remplacement, il suffit de prouver que les modèles $m(i)$ et $m(c)$ atteignent toujours

un état final (absorbant par définition) et quand $m(i)$ et $m(c)$ atteignent un état final, les valeurs sur l'espace de données sont similaires. A titre d'illustration, ces propriétés à vérifier peuvent être exprimées à l'aide de la logique temporelle CTL via la formule 4.3.

$$AG[((i.pc = pc\_init\_i) \wedge (c.pc = pc\_init\_c)) \Rightarrow AF((i.pc = pc\_final\_i) \wedge (c.pc = pc\_final\_c) \wedge \forall x \in D, (i.x = c.x))] \quad (4.3)$$

Cette formule est de la forme $AG(a \Rightarrow AF(b \wedge c))$, avec :

- a : états initiaux
- b : états finaux
- c : égalité des données sur D
- $a \Rightarrow AF(b \wedge c)$: tous les chemins satisfaisant a atteignent inévitablement un état satisfaisant b et c
- $AG(\varphi)$: la proposition φ est vraie pour tous les états du graphe des états accessibles

Celle-ci sera décomposée en plusieurs formules plus simples pour les exemples présentés dans la partie qui suit. Cette décomposition permet de vérifier séparément différentes propriétés.

4.3.2.5. Logiciel de model-checking Vis

Pour vérifier le schéma de contre-mesure proposé, le logiciel de model-checking *Vis* [13] a été utilisé. Cet outil permet de spécifier un ensemble d'automates synchrones puis d'évaluer des propriétés de logique temporelle sur le graphe des états de cet ensemble d'automates. Ces automates synchrones peuvent être décrits à l'aide du langage Verilog, qui est particulièrement approprié pour modéliser des systèmes de transitions qui manipulent des registres et des vecteurs de bits. Le logiciel *Vis* supporte les techniques de model-checking symbolique et SAT, ce qui permet de réaliser la vérification sans énumérer l'ensemble des valeurs pour tous les registres.

4.3.3 Vérification formelle de séquences de remplacement

Les paragraphes qui suivent présentent plusieurs exemples de vérification de séquences de remplacement pour différents types d'instructions.

13. http://vlsi.colorado.edu/~vis

4.3.3.1. Instructions arithmétiques et logiques

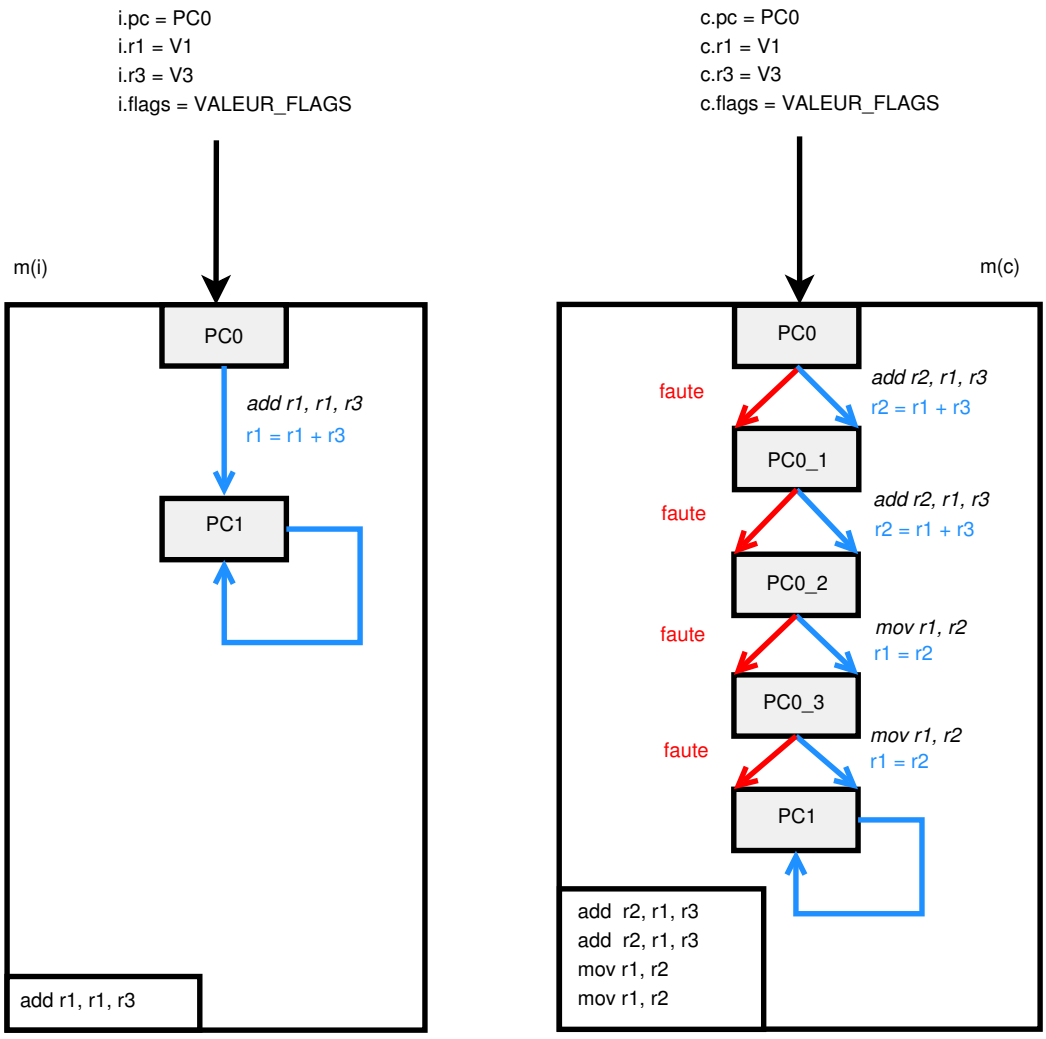
La partie gauche de la figure 4.2 montre la machine d'états qui correspond au système de transitions pour une instruction non-idempotente `add r1,r1,r3`. Le compteur de programme est mis à jour et, selon l'instruction, les registres ou les *flags* peuvent également être mis à jour. La séquence de remplacement utilise ici le registre non-vivant `r12` et deux instructions `mov` supplémentaires pour écrire le résultat dans le registre destination `r1`. Le système de transitions de la contre-mesure avec occurrence aléatoire d'une faute exactement est modélisé par la machine d'états de la partie droite de la figure 4.2. Pour prouver que la séquence de remplacement est tolérante à une faute sous la forme de saut d'instruction, les deux machines d'états commencent avec les mêmes valeurs pour les registres source (`r1` et `r3`) et les *flags*. `i.r1` et `c.r1` sont affectés à la même variable libre `V1`. De même, `i.r3` et `c.r3` sont affectés à la variable libre `V3`, et les *flags* des deux machines d'états affectés à la variable libre `VALEUR_FLAGS`. Ensuite, la validité de trois propriétés P1, P2 et P3 exprimées en logique temporelle CTL a été vérifiée à l'aide de l'outil *Vis*. Les propriétés P1 et P2 expriment le fait que, dans les deux machines d'états, tout chemin d'un état initial arrive dans un état final. La propriété P3 exprime le fait que dans cet état final, pour toutes les valeurs initiales possibles des registres source, les valeurs de `r1` et des *flags* sont identiques pour $m(i)$ et $m(c)$.

4.3.3.2. Instructions de lecture et écriture en mémoire

La figure 4.3 présente les systèmes de transitions pour l'écriture mémoire idempotente `str r3,[r1,r2]`, et pour sa séquence de remplacement. Dans ce cas, comme l'instruction écrit le contenu de `r3` dans la mémoire à l'adresse `r1+r2` et que l'on considère un modèle de saut d'instruction, aucune preuve n'est nécessaire sur les valeurs contenues dans les registres. Pour vérifier cette instruction et sa séquence de remplacement, il suffit de vérifier qu'au moins une instruction `str` a bien été exécutée. Une variable `cpt` qui sert de compteur est ajoutée à la définition d'un état. Ce compteur est initialement mis à 0 et est incrémenté à chaque transition qui correspond à une instruction `str`. Les propriétés P1 et P2 expriment le fait que tout chemin arrive forcément dans un état final. La propriété P3 exprime le fait que le nombre d'écritures faites par la séquence de remplacement doit être égal à 1 ou 2.

4.3.3.3. Instruction d'appel de sous-fonction

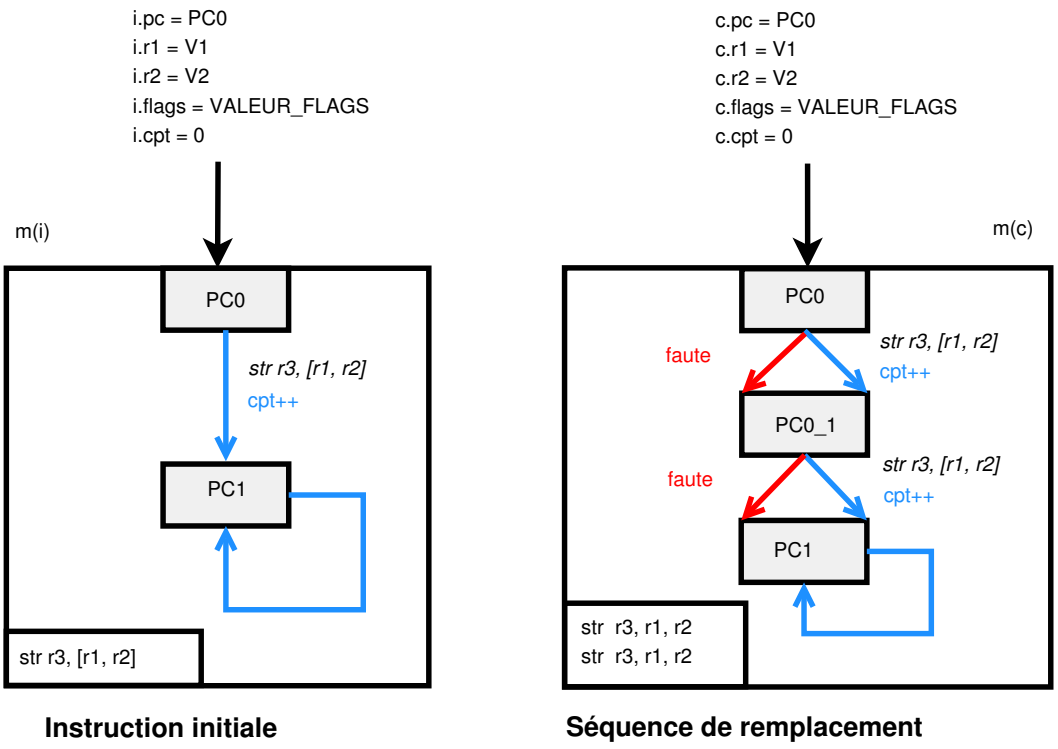
La figure 4.4 montre les machines d'états pour une instruction `bl` et sa séquence de remplacement. Dans les deux systèmes de transitions, un label $@sous\_fonction$ a été ajouté pour modéliser la sous-fonction cible du branchement. Les transitions depuis un état dans lequel $PC = @sous\_fonction$ mettent la valeur du *Link Register* dans le compteur de programme. Une telle transition modélise le retour de la fonction




```
P1 : AF(i.pc = PC1)
P2 : AF(c.pc = PC1)
P3 : AG((i.pc = PC1 * c.pc = PC1) => (i.r1 = c.r1 * i.flags = c.flags))
```

Figure 4.2. : Modélisation pour une instruction `add` non-idempotente et sa contre-mesure

et incrémente un compteur. Ensuite, les propriétés P1 et P2 à vérifier par l'outil de model-checking expriment le fait que tout chemin d'un état initial va à un état final. Elles assurent ainsi que le compteur de programme revient bien à la fonction appelante. La propriété P3 exprime le fait que dans un état final le nombre d'appels à la fonction est le même pour i et pour c et que ce nombre est égal à 1.



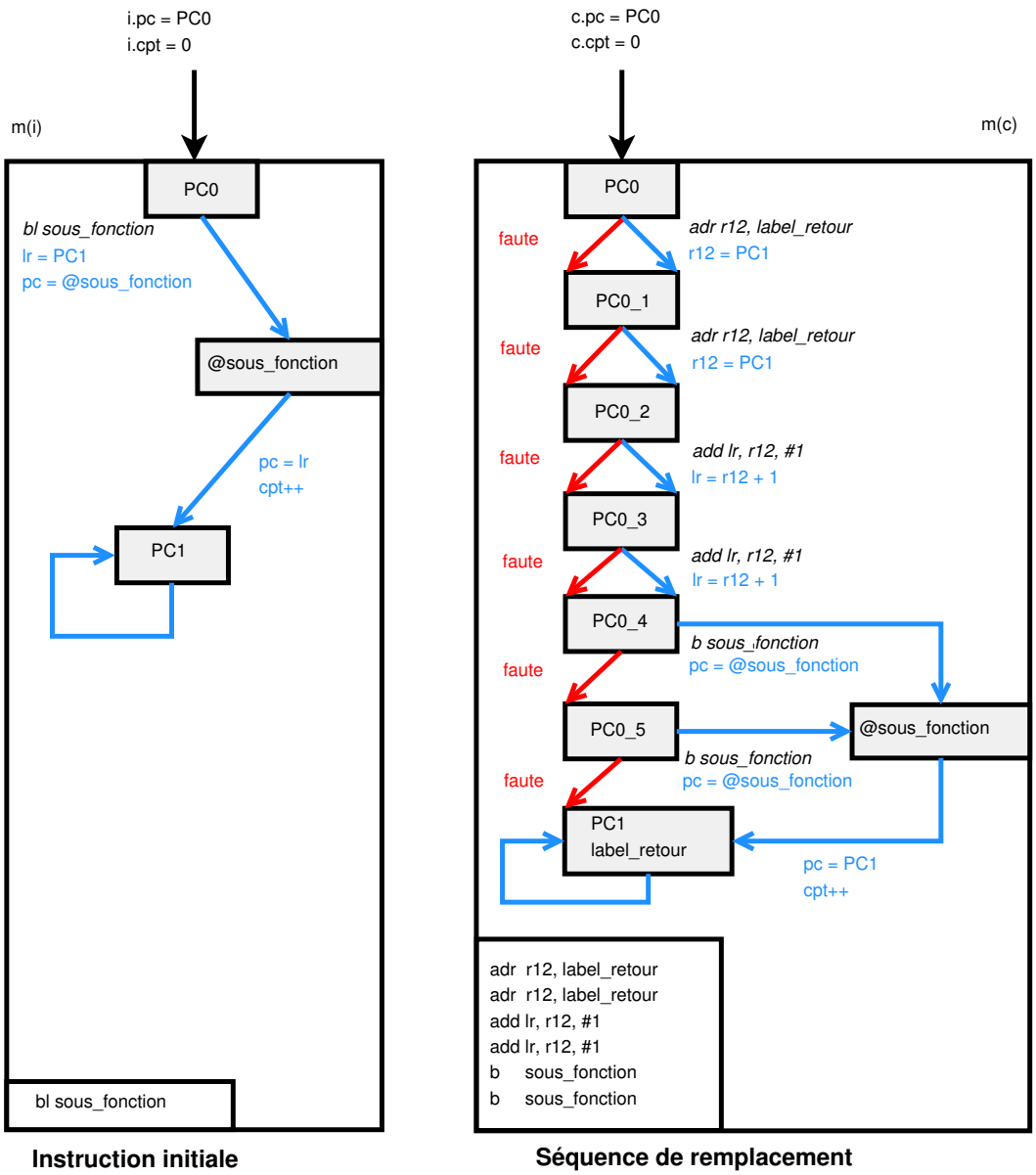
Instruction initiale **Séquence de remplacement**

```
P1 : AF(i.pc = PC1)
P2 : AF(c.pc = PC1)
P3 : AG((i.pc = PC1 * c.pc = PC1) => (c.cpt = 2 + c.cpt = 1))
```

Figure 4.3. : Modélisation pour une instruction idempotente `str` et sa contre-mesure

4.3.3.4. Instructions qui lisent et écrivent les flags

Pour l'instruction `adcs` et sa séquence de remplacement (présentée dans le listing 4.14), les propriétés en logique CTL sont les mêmes que celles utilisées pour une instruction `add`. Toutefois, la propriété qui garantit l'égalité des registres de destination et des *flags* n'est pas valide si une faute vise la dernière instruction `adcs`. Cette égalité des registres et *flags* est exprimée par la proposition `RESULT=1` dans la figure 4.5. Cette figure montre la sortie du logiciel Vis pour la vérification des propriétés énoncées précédemment. En revanche, il est possible de relâcher la contrainte sur les *flags* pour garantir seulement une égalité des registres en sortie. Cette égalité plus restreinte est exprimée par la proposition `LIGHT_RESULT=1` dans la figure 4.5. Comme illustré sur la figure, il est donc possible de garantir au moins l'égalité des registres en sortie. Cette contre-mesure peut donc seulement être utilisée si les *flags* ne sont pas vivants après l'instruction `adcs` initiale.




```
P1 : AF(i.pc = PC1)
P2 : AF(c.pc = PC1)
P3 : AG((i.pc = PC1 * c.pc = PC1) => (i.cpt = 1 * c.cpt = i.cpt))
```

Figure 4.4. : Systèmes de transitions pour une instruction `bl` et sa séquence de remplacement

4.3.3.5. Statistiques de vérification

Le processus de vérification est basé sur une modélisation des opérations arithmétiques et logiques réalisées par les instructions, et sur une modélisation de leurs effets sur les registres et *flags*. Ainsi, la taille minimale requise pour modéliser les registres est la taille minimale qui permette de modéliser précisément les effets sur

```
MC : formula passed - AG(AF(i.pc=PC1))
MC : formula passed - AG(AF(c.pc=PC1))
MC : formula failed - AG((i.pc=PC1*c.pc=PC1)->RESULT=1)
MC : formula passed - AG((i.pc=PC1*c.pc=PC1)->LIGHT_RESULT=1)
```

Figure 4.5. : Sortie du logiciel Vis pour la vérification de l'instruction `adcs`

les *flags*. Ainsi, pour ce type de vérification, utiliser des registres 4 bits est suffisant. Toutefois, le processus de vérification mène aux mêmes résultats pour des registres modélisés avec une taille de 4 bits ou pour des registres modélisés avec une taille supérieure.

Le temps d'exécution du processus de vérification et la taille du système de transitions dépendent de la taille des registres et des opérations modélisées. Le processus de vérification s'est exécuté en moins d'une seconde pour des registres de 4 bits pour toutes les instructions, et en moins d'une minute pour pratiquement toutes les instructions pour une taille de registres de 16 bits. Néanmoins, il y a eu besoin de 29 heures pour vérifier l'instruction `umlal` (la plus coûteuse en termes de séquence de remplacement) avec des registres de 16 bits. De plus, le logiciel Vis n'a pas pu construire de représentation interne ou réaliser la vérification pour des registres au-dessus de 24 bits. Des travaux en cours visent à lever cette limitation en utilisant des représentations SAT et SMT mieux adaptés que les BDD pour ces vérifications.

4.4 Application automatique de la contre-mesure

Pour appliquer la contre-mesure à des codes de grande taille, un algorithme d'automatisation a été conçu. Celui-ci est présenté en 4.4.1. Ensuite, cette contre-mesure a été appliquée à plusieurs codes pour en évaluer le surcoût global, à la fois en termes de taille de code et de nombre de cycles d'horloge. Les résultats de l'évaluation du surcoût sont présentés en 4.4.2.

4.4.1 Algorithme d'application automatique

Un algorithme d'application de la contre-mesure a été conçu. Celui-ci reçoit en entrée un code binaire désassemblé non sécurisé et réalise une traduction de code assembleur vers du code assembleur renforcé. Les étapes de l'algorithme sont détaillées en 4.4.1.1. Enfin, cet algorithme présente quatre limitations, décrites en 4.4.1.2.

4.4.1.1. Présentation des étapes de l'algorithme

L'algorithme est structuré en deux passes.

— la première reconstruit les adresses symboliques

— la seconde produit, pour chaque instruction du code à sécuriser, une séquence de remplacement et l'insère dans le programme

La première passe est nécessaire avant de pouvoir appliquer automatiquement la contre-mesure. Elle est composée de deux étapes.

1. Comme les branchements pointent directement vers une adresse, la première étape consiste à reconstruire des labels associés aux cibles des différents branchements. Ces labels servent ensuite à définir les adresses de destination des nouveaux branchements.

2. La seconde étape consiste à réécrire les instructions de chargement de données depuis une adresse en mémoire d'instructions (comme `ldr r0,[pc#40]`). Comme l'adresse de la donnée à charger est susceptible de changer, ces instructions doivent être réécrites sous la forme de pseudo-instructions de chargement direct (comme présenté en 2.2.9.2).

Cette passe préliminaire d'analyse est présentée dans l'algorithme 4.

Algorithme 4 : Première passe préliminaire d'analyse du code

Data : $instruction[n]$ une séquence de n instructions
Result : Séquence de $m > n$ instructions renforcée à l'aide de la contre-mesure

```
1 for i <= n do
2     if instruction[n] est un branchement vers une adresse then
3         Détecter l'instruction de destination de ce branchement
4         Créer un label avant cette instruction de destination
5         Remplacer instruction[n] par un branchement vers ce label
6     if instruction[n] est un chargement depuis une adresse relative au pc then
7         Récupérer la valeur stockée à cette adresse
8         Remplacer instruction[n] par une pseudo-instruction de chargement direct
          (du type ldr r0,=0x12345678)
```

Enfin, une fois cette passe préliminaire d'analyse achevée, la seconde passe d'application automatique de la contre-mesure à l'ensemble du code en utilisant les séquences de remplacement présentées dans ce chapitre peut être réalisée. Cette seconde passe de renforcement du code est présentée dans l'algorithme 5.

4.4.1.2. Limitations liées à l'utilisation d'un code assembleur compilé

Instructions non-remplaçables Une des limitations de l'algorithme est liée au cas des instructions non-remplaçables, pour lesquelles un remplacement à l'échelle de l'instruction est impossible. Dans ce cas, l'algorithme avertit simplement l'utilisateur de la présence d'une instruction non-remplaçable. Le programmeur peut alors choisir de ne pas remplacer l'instruction en question ou bien de réécrire manuellement une partie du code de la fonction dont fait partie cette instruction, de façon à utiliser uniquement des instructions remplaçables.

Algorithme 5 : Seconde passe de renforcement du code

```
1  for i <= n do
2      switch instruction[n] do
3          case Instruction idempotente
4              Dupliquer instruction[n]
5          case Instruction séparable
6              Détecter le nombre de registres supplémentaires nécessaires
7              if Des registres supplémentaires sont nécessaires then
8                  if Suffisamment de registres non-vivants sont disponibles then
9                      Sélectionner des registres non-vivants
10                 else
11                     if Suffisamment de registres peuvent être placés sur la pile then
12                         Modifier la séquence de remplacement de l'instruction push
                           en début de fonction pour libérer davantage de registres
13                         Modifier l'instruction de pop en fin de fonction pour restaurer
                           ces registres
14                     else
15                         Avertir l'utilisateur sur le manque de registres et l'impossibilité
                           d'appliquer la contre-mesure sur cette portion de code
16             if instruction[n] est une instruction de manipulation de la pile then
17                 Calculer le nombre d'octets lus ou écrits sur la pile
18                 if instruction[n] est une instruction pop qui écrit dans le registre pc
                   then
19                     Utiliser la séquence de remplacement pour une instruction pop
                       qui écrit sur le registre pc
20                 else
21                     Utiliser la séquence de remplacement pour une instruction de
                       manipulation de la pile
22             else if instruction[n] est une instruction umlal then
23                 Utiliser la séquence de remplacement pour une instruction umlal
24             else if instruction[n] utilise un décalage rrx et lit les flags then
25                 Réaliser l'opération de décalage rrx
26                 Utiliser la séquence de remplacement pour une instruction séparable
27             else
28                 Utiliser la séquence de remplacement pour une instruction séparable
29         case Instruction particulière
30             if instruction[n] est une instruction bl d'appel de sous-fonction then
31                 Utiliser la séquence de remplacement pour une instruction bl
32             else if instruction[n] est un bloc it then
33                 Utiliser la séquence de remplacement pour un bloc it
34         otherwise
35             Avertit l'utilisateur de la présence d'une instruction non-remplaçable
```

Adresses de branchement obtenues par calcul L'analyse statique de reconstruction des adresses symboliques n'ajuste pas les adresses calculées. Un exemple de ce cas de figure est présenté sur le listing 4.15. Dans cet exemple, l'adresse `0x08000123` est calculée par trois instructions (`mov`, `lsl` et `add`) et un branchement est ensuite réalisé vers cette adresse. L'application automatique de la contre-mesure présentée dans ce chapitre sur ce programme ne modifiera pas le résultat de ce calcul d'adresse (qui sera donc toujours `0x08000123`). Cependant, suite à l'application de la contre-mesure, l'adresse de l'instruction cible de ce changement peut très bien ne plus être `0x08000123`.

Listing 4.15 : Exemple de code qui réalise un calcul d'adresse

```
1 mov     r7 , #0x0800     ; place les 16 bits de poids fort dans r7
2 lsl     r7 , r7 , #16    ; decalage de 16 bits vers la gauche
3 add     r7 , r7 , #0x123 ; ajoute les 16 bits de poids faible a r7
4 bx      r7               ; branche vers l'adresse stockee dans r7
```

Détection de registres non-vivants Une autre limitation concerne les registres vivants : pour plusieurs séquences de remplacement, la contre-mesure proposée requiert un ou plusieurs registres supplémentaires. Ces registres peuvent être pris parmi les registres non-vivants à ce point du programme. Dans le cas où aucun registre n'est vivant à un point du programme et si suffisamment de registres peuvent être placés sur la pile, il est possible de modifier l'instruction de `push` au début de la fonction de façon à libérer davantage de registres en utilisant la pile. Néanmoins, le calcul des registres vivants à un point du programme est une opération faite nativement par les compilateurs mais complexe à réaliser sur un code assembleur déjà compilé. Ainsi, le programme de traduction utilisé pour les travaux présentés dans cette thèse ne suit pas exactement l'algorithme présenté précédemment. Ce programme utilise le registre `r12` pour les séquences qui nécessitent un registre supplémentaire et requiert une intervention de l'utilisateur pour les séquences qui nécessitent plusieurs registres.

Analyse sémantique et optimisations L'algorithme d'application automatique présenté ne réalise aucune analyse sémantique sur le code assembleur source car de telles analyses sont difficiles à réaliser sur un code compilé. L'algorithme applique donc la contre-mesure instruction par instruction. Néanmoins, l'application de la contre-mesure sans analyse à une échelle plus large peut aboutir à la génération de code non-optimisé. Par exemple, le listing 4.16 présente deux instructions sur lesquelles l'algorithme d'automatisation doit être appliqué. L'instruction `cmp` lit le résultat de l'instruction `add` pour réaliser une comparaison, ce résultat est placé dans le registre `r1`. Après application de l'algorithme, le code obtenu est présenté dans le listing 4.17. Comme l'instruction `add r1,r1,#1` est une instruction séparable, le registre `r12` est utilisé comme registre temporaire et le résultat final est placé dans `r1`. Toutefois, en termes de sémantique, il est strictement équivalent pour

l'instruction `cmp` de lire `r1` ou bien de lire `r12` puisque les deux registres contiennent la même information. Ainsi, il serait possible de supprimer les deux instructions `mov` et d'utiliser `r12` comme registre source pour l'instruction `cmp` tout en obtenant un résultat équivalent. Cette optimisation n'est pas réalisée par l'algorithme présenté précédemment. Le code optimisé à la suite de cette opération est présenté dans le listing 4.18.

Listing 4.16 : Exemple de code à renforcer

```
1 add     r1 , r1 , #1      ; ajoute 1 a r1
2 cmp     r1 , #9           ; compare la valeur dans r1 a 9
```

Listing 4.17 : Application de la contre-mesure instruction par instruction

```
1 add     r12 , r1 , #1
2 add     r12 , r1 , #1
3 mov     r1 , r12          ; la sequence de remplacement ecrit dans r1
4 mov     r1 , r12          ; pour permettre a l'instruction cmp de lire r1
5 cmp     r1 , #9
6 cmp     r1 , #9
```

Listing 4.18 : Optimisation possible en analysant sémantiquement le code

```
1 add     r12 , r1 , #1
2 add     r12 , r1 , #1
3 cmp     r12 , #9          ; r12 peut directement etre utilise par l'inst. cmp
4 cmp     r12 , #9
```

4.4.2 Résultats en termes de surcoût

Quatre implémentations ont été considérées pour une mesure du surcoût apporté par la contre-mesure proposée dans ce chapitre. Les deux premières sont des implémentations de l'algorithme AES dans sa version 128 bits et la troisième est une implémentation de l'algorithme SHA-0. Nous avons développé le premier code AES à partir de la spécification (NIST, 2001). Pour celui-ci, chaque clé de ronde est calculée avant l'opération d'`AddRoundKey` associée. Le second code AES et celui de SHA-0 proviennent de la suite de codes MiBench (GUTHAUS et al., 2001). Enfin, la quatrième implémentation correspond à l'implémentation AES que nous avons développée et pour laquelle la contre-mesure a seulement été appliquée sur les deux dernières rondes du chiffrement. Cette dernière implémentation n'a pas pour but de représenter un cas réel de renforcement d'une implémentation AES mais vise à montrer la possibilité de réduire significativement les surcoûts avec une bonne connaissance des parties sensibles de l'algorithme à renforcer. Par ailleurs, il est important de mentionner que toutes les instructions des codes testés ont pu être remplacées à l'aide de séquences de remplacement présentées dans ce chapitre. De plus, suffisamment de registres non-vivants étaient disponibles pour toutes les séquences de remplacement qui nécessitaient un ou plusieurs registres supplémentaires. Le

surcoût en termes de taille de code et nombre de cycles d'horloge pour ces quatre implémentations est présenté dans le tableau 4.4.

Table 4.4. : Surcoût apporté par la contre-mesure pour plusieurs implémentations

Implémentation	Sans contre-mesure		Avec contre-mesure			
	Cycles	Taille code	Cycles	Surcoût	Taille code	Surcoût
AES	9595	490 o	20503	113.7 %	1480 o	202 %
MiBench AES	9294	3372 o	26618	186.4 %	9776 o	189.9 %
MiBench SHA-0	4738	746 o	10558	122.8 %	2076 o	178.2 %
AES avec contre-mesure sur les 2 dernières rondes	9595	490 o	11374	18.6 %	1874 o	282.5 %

Le surcoût apporté par la contre-mesure est élevé pour les trois premières implémentations avec, selon les implémentations, un nombre de cycles doublé voire triplé et une taille de code quasiment triplée. Néanmoins, en termes de nombre de cycles, un tel surcoût reste relativement comparable à ceux apportés par des approches de sécurisation de code (basées sur une duplication à l'échelle de l'algorithme) ou de tolérance aux fautes (basées au moins sur une triplication). La quatrième implémentation montre également la possibilité de minimiser le surcoût en nombre de cycles en choisissant les fonctions à renforcer, au détriment d'un surcoût important en taille de code. Ce surcoût est lié à la structure particulière de l'algorithme AES qui réalise plusieurs rondes de chiffrement : deux implémentations pour chacune des fonctions de l'algorithme (une non-renforcée pour les 8 premières rondes et une renforcée pour les deux dernières) doivent alors être placées en mémoire.

4.5 Conclusion et perspectives

Dans ce chapitre, un schéma de contre-mesure qui permet de renforcer un programme embarqué et le rendre tolérant aux fautes par saut d'instructions a été présenté. Une séquence de remplacement a été proposée pour presque toutes les instructions du jeu d'instructions Thumb-2. Les instructions de Thumb-2 peuvent être divisées en trois classes, les deux premières ayant chacune leurs séquences de remplacement dédiées et la dernière correspondant aux instructions sans séquence de remplacement proposée. Enfin, un processus de vérification pour garantir la validité et la tolérance à une faute des séquences de remplacement a été mis en place pour chaque classe d'instructions.

Ce schéma de contre-mesure n'a pas pour prétention de proposer une protection totale face aux attaques par injection de fautes. Pour obtenir cela, plusieurs niveaux de contre-mesure (matérielle, logicielle, algorithme) devraient être ajoutés pour un coût bien plus élevé et un temps de développement accru. Néanmoins, cette contre-mesure présente trois principaux avantages : la possibilité d'être appliquée de façon automatique à un code générique pour en renforcer la sécurité, son surcoût comparable à des approches de sécurisation face aux attaques en faute par redondance à l'échelle de l'algorithme, et enfin le fait que la redondance soit apportée au

niveau de l'instruction et non plus de l'algorithme ou la fonction. Elle permet donc théoriquement de renforcer de manière très simple et pour un coût raisonnable un code face aux attaques en faute et de réduire l'efficacité de ces attaques.

L'algorithme d'application automatique de la contre-mesure présenté en fin de chapitre permet d'utiliser directement la contre-mesure sur des codes complexes mais présente encore quelques limitations. Ces limitations sont dues au fait que l'algorithme utilise en entrée un code binaire désassemblé et réalise la transformation du code instruction par instruction. Or, un compilateur reçoit en entrée un code source sur lequel il peut réaliser des analyses sémantiques, possède des représentations intermédiaires contenant davantage d'informations que le code assembleur finalement produit, il décide de l'attribution des registres et peut être configuré de façon à ne pas utiliser certaines instructions pour lesquelles aucune séquence de remplacement n'a été trouvée. Une application de la contre-mesure directement au sein au compilateur permettrait de résoudre l'ensemble des problèmes soulevés en fin de chapitre et permettrait de rendre l'application de la contre-mesure totalement transparente pour le programmeur.

Si la robustesse face au saut d'une instruction et la sémantique des séquences de remplacement ont pu être vérifiées formellement dans ce chapitre, il reste nécessaire d'évaluer expérimentalement l'efficacité de cette contre-mesure et sa résistance face à des injections de fautes réelles. De plus, cette contre-mesure ne propose aucune protection au niveau du flot de données, alors que certaines expérimentations présentées dans cette thèse ont montré la possibilité de corrompre des transferts de données. Il est donc nécessaire de compléter cette contre-mesure de façon à également protéger le flot de données. Enfin, les aspects liés aux attaques par observation n'ont pas été traités dans ce chapitre, et il est important de vérifier si cette contre-mesure peut avoir un effet sur les fuites d'information lorsqu'elle est appliquée à une implémentation cryptographique. Ces différents points liés à l'évaluation expérimentale de la contre-mesure sont l'objet du chapitre suivant.

5 Évaluation expérimentale de la contre-mesure proposée

*Certains des travaux présentés dans ce chapitre ont fait l'objet d'un article présenté lors de la conférence IEEE HOST 2014 (*MORO, HEYDEMANN, DEHBAOUI *et al., 2014).*

Sommaire

5.1 Introduction

Dans le chapitre précédent, une contre-mesure a été élaborée face à un modèle simplifié d'attaquant. Ce modèle avait lui-même été extrait des résultats expérimentaux des chapitres 2 et 3. La contre-mesure proposée fournit une méthode de sécurisation du flot de contrôle d'un programme assembleur pour le modèle de fautes par saut d'instruction considéré. Cette contre-mesure ne répond au problème de la sécurisation du flot de données, qui doit passer par une protection des transferts de données.

Cet aspect a en revanche été adressé par Barenghi *et al.* (BARENGHI, BREVEGLIERI, KOREN, PELOSI et al., 2010), qui propose une contre-mesure de détection de fautes au niveau assembleur. Cette contre-mesure pourrait ainsi compléter la contre-mesure proposée dans le chapitre précédent.

Nous proposons de combiner ces deux contre-mesures. A cet effet, il est nécessaire de mieux connaître leurs points forts et faibles. Ce chapitre vise à évaluer expérimentalement la robustesse de ces deux contre-mesures. Pour cela, ce chapitre présente de façon plus détaillée la contre-mesure de Barenghi *et al.* puis réalise cette évaluation des deux contre-mesures en les appliquant telles qu'elles ont été définies. L'évaluation est d'abord réalisée sur des instructions isolées, puis sur des codes plus complexes issus d'une implémentation de FreeRTOS. A partir des résultats de cette étude, ces deux contre-mesures sont combinées pour renforcer un code assembleur extrait d'une implémentation de l'algorithme AES. Enfin, une première analyse simple de l'impact de cette contre-mesure face aux attaques par observation est proposée en fin de chapitre.

5.2 Évaluation expérimentale face aux injections de faute

Les sous-parties qui suivent commencent par présenter la contre-mesure de détection proposée par Barenghi *et al.* en 5.2.1. Ensuite, le choix d'une métrique de robustesse pour l'évaluation des contre-mesures est discuté en 5.2.2. Les paramètres expérimentaux utilisés pour l'évaluation expérimentale des contre-mesures sont présentés en 5.2.3. Une première évaluation expérimentale de robustesse sur des instructions isolées est alors présentée en 5.2.4. Enfin, une évaluation sur des codes plus complexes provenant d'une implémentation de FreeRTOS est présentée en 5.2.5.

5.2.1 Contre-mesure de détection de fautes

La contre-mesure proposée par Barenghi *et al.* vise a détecter un grand nombre de fautes uniques (*i.e.* de fautes non-multiples) sur une petite portion de code de quelques instructions. Plus précisément, cette contre-mesure permet de détecter le saut d'une instruction, certains cas de remplacements d'instructions ainsi que l'injection d'une faute sur le flot de données (par chargement de donnée corrompue ou modification de la valeur du registre destination). Le modèle de fautes considéré est plus large que celui considéré pour la contre-mesure proposée dans le chapitre 4.

L'exemple donné par Barenghi *et al.* (BARENGHI, BREVEGLIERI, KOREN, PELOSI et al., 2010) pour protéger une instruction `ldr` est reproduit dans le listing 5.1. La contre-mesure est basée sur une duplication d'instruction associée à un stockage du résultat dans un registre supplémentaire. Ensuite, une comparaison est réalisée pour détecter

toute différence entre les deux registres de destination. Si une erreur est détectée, le programme branche vers une sous-fonction d'erreur.

Ce principe de détection peut directement être appliqué à plusieurs instructions arithmétiques et logiques. Il est à noter que les auteurs de l'article font eux-mêmes mention de l'existence d'instructions pour lesquelles cette contre-mesure ne peut pas être appliquée mais n'en donnent pas de liste. Nous avons néanmoins pu observer que cette contre-mesure ne s'applique pas en l'état aux instructions qui utilisent les *flags*, aux instructions de branchement et aux instructions de manipulation de la pile [1].

Listing 5.1 : Contre-mesure de détection pour une instruction `ldr`

```
1 ldr   r0 , [pc, #40]     ; instruction ldr initiale
2 ldr   r1 , [pc, #38]     ; instruction ldr dupliquée
3 cmp   r0 , r1            ; comparaison entre r0 et r1
4 bne   error              ; si r0 != r1, déclenche une erreur
```

5.2.2 Définition d'une métrique de robustesse

Nous proposons de définir une métrique pertinente pour évaluer l'efficacité pratique des contre-mesures considérées. Comme les séquences de remplacement proposées par les deux contre-mesures ajoutent des instructions au code, le temps d'exécution d'une séquence de remplacement complète devient plus long que le temps nécessaire pour exécuter l'instruction initiale. Ainsi, le nombre de points vulnérables, *i.e.* les instants d'injection pour lesquels une tentative d'injection de fautes amène à l'observation d'une sortie fautée, est susceptible d'augmenter. Par conséquent, comparer le nombre de sorties fautées par rapport au nombre d'injections réalisées pourrait être une métrique intéressante pour comparer deux ensembles de données avec un nombre différent de mesures. Par exemple, considérons le cas d'un code qui s'exécute en $1\ \mu s$ pour lequel 100 sorties fautées ont été obtenues, et considérons le même code après application d'une contre-mesure, qui s'exécute en $2\ \mu s$ et pour lequel 150 sorties fautées ont été obtenues. En considérant ce rapport entre le nombre de sorties fautées et le temps d'exécution, on pourrait donc considérer que la contre-mesure contribue à renforcer le code.

Néanmoins, l'augmentation en valeur absolue du nombre de sorties fautées signifie donc que davantage de points vulnérables peuvent être exploités par un éventuel attaquant, même si ces points vulnérables sont davantage espacés dans le temps. Nous pensons donc que la contre-mesure est réellement efficace si elle réussit à diminuer ce nombre de points vulnérables en valeur absolue. Nous choisissons donc de comparer le nombre total de sorties fautées pour l'exécution d'une fonction

1. Les instructions de manipulation de la pile peuvent néanmoins être transformées en suites de `ldr/str` sur lequel il serait possible d'appliquer cette contre-mesure de détection. Néanmoins, cette solution serait particulièrement coûteuse et n'a pas été étudiée par les auteurs de l'article

donnée car nous pensons que cette métrique indique le nombre de vulnérabilités potentielles d'un code embarqué.

5.2.3 Paramètres expérimentaux utilisés pour l'évaluation

Les évaluations de cette section ont été réalisées en répétant un même procédé expérimental. Les paramètres utilisés pour celui-ci sont présentés dans le tableau 5.1. La position d'antenne est fixée et correspond à celle utilisée pour les expérimentations des chapitres 2 et 3. Les exceptions matérielles sont utilisées pour connaître l'instruction atteinte par une injection de faute. Elles servent donc également à définir l'intervalle temporel qui sera balayé pour faire varier l'instant d'injection. Pour l'amplitude de l'impulsion, des tensions négatives allant de -210 V à -170 V ont été utilisées, ainsi que des tensions positives allant de 130 V à 150 V. Enfin, le processus expérimental a été répété deux fois pour chaque instant d'injection et tension d'impulsion.

Table 5.1. : Paramètres expérimentaux utilisés dans cette section

Position de l'antenne	Fixée (trouvée par un processus d'essai et erreur)
Instant d'injection	Durée définie à l'aide des exceptions matérielles (`UsageFault`, `BusFault`, similaire au chapitre 3) Intervalle temporel parcouru par pas de : - 200 ps pour les instructions isolées - 500 ps pour les implémentations de FreeRTOS
Amplitude de l'impulsion	De -210 V à -170 V, par pas de 10 V De 130 V à 150 V, par pas de 10 V
Nombre d'injections	2 essais par instant d'injection et tension d'impulsion

5.2.4 Évaluation expérimentale de la robustesse sur une instruction

Les paragraphes qui suivent évaluent l'efficacité des deux contre-mesures pour des instructions isolées. Les résultats pour la contre-mesure de tolérance sont présentés en 5.2.4.1 et ceux pour la contre-mesure de détection en 5.2.4.2.

5.2.4.1. Contre-mesure de tolérance à un saut d'instruction

L'exemple considéré pour cette première évaluation est présenté dans le listing 5.2. Cette séquence remplace l'instruction `bl` d'appel de sous-fonction. Elle imite donc l'effet d'une instruction `bl` en sauvegardant le pointeur de retour dans le registre `lr` et en branchant vers la fonction appelée. Même si aucune faute n'est injectée, la sous-fonction n'est appelée qu'une fois car l'adresse de retour pointe vers le label après les deux instructions `b`.

Listing 5.2 : Contre-mesure de tolérance pour une instruction `bl fonction`

```
1      adr r12, label_retour  ; enregistrement du pointeur de retour dans r12
2      adr r12, label_retour
3      add lr, r12, #1        ; lr=r12+1, le dernier bit du pointeur doit etre
4      add lr, r12, #1        ; mis a 1 pour le mode d'execution Thumb
5      b   fonction           ; branchement vers la fonction
6      b   fonction
7  label_retour
```

Dans le listing 5.2, les deux instructions `adr` ainsi que les deux instructions `b` sont encodées sur 16 bits par défaut. Comme les deux instructions `add` utilisent le registre `lr` (`r14`), elles sont nécessairement encodées sur 32 bits.

Quatre variantes de ce code ont été évaluées pour cette série d'expérimentations :

— une instruction `bl` sans contre-mesure (temps d'exécution de 100 ns, 1000 impulsions envoyées par valeur de tension)

— une instruction `bl.w` avec encodage forcé sur 32 bits sans contre-mesure (temps d'exécution de 100 ns, 1000 impulsions envoyées par valeur de tension)

— la séquence de remplacement du listing 5.2 (temps d'exécution de 400 ns, 4000 impulsions envoyées par valeur de tension)

— la séquence de remplacement du listing 5.2 avec encodage forcé sur 32 bits pour toutes les instructions de ce code (temps d'exécution de 400 ns, 4000 impulsions envoyées par valeur de tension)

La sous-fonction appelée charge une valeur dans le registre `r0`. Pour cet exemple, nous avons choisi d'observer deux grandeurs que nous avons considéré comme significatives : le nombre de fautes obtenues dans `r0` et le nombre de fautes sur au moins un registre [2].

Les résultats issus des injections de fautes sont présentés sur la figure 5.1. Pour des impulsions de tension négative, 608 réalisations avec une faute sur au moins un registre ont été obtenues pour l'implémentation sur 16 bits sans contre-mesure, contre 244 réalisations pour l'implémentation sur 16 bits avec contre-mesure. Dans l'ensemble des autres cas (avec la métrique du nombre de fautes dans `r0` ou pour des impulsions de tension positive), l'implémentation sur 16 bits avec contre-mesure s'est révélée plus vulnérable que celle sur 16 bits sans contre-mesure. Il ressort donc que l'application de la contre-mesure sans encodage sur 32 bits n'a globalement pas été efficace pour cette expérimentation. Ce résultat peut s'expliquer par le fait que cette contre-mesure n'a pas été conçue pour résister à deux sauts d'instruction consécutifs. En raison de l'alignement mémoire utilisé pour cette expérimentation, les deux instructions `adr` et plus tard les deux instructions `b` de la séquence de

2. Identifie les sorties pour lesquelles au moins un registre contient une valeur différente de la valeur attendue

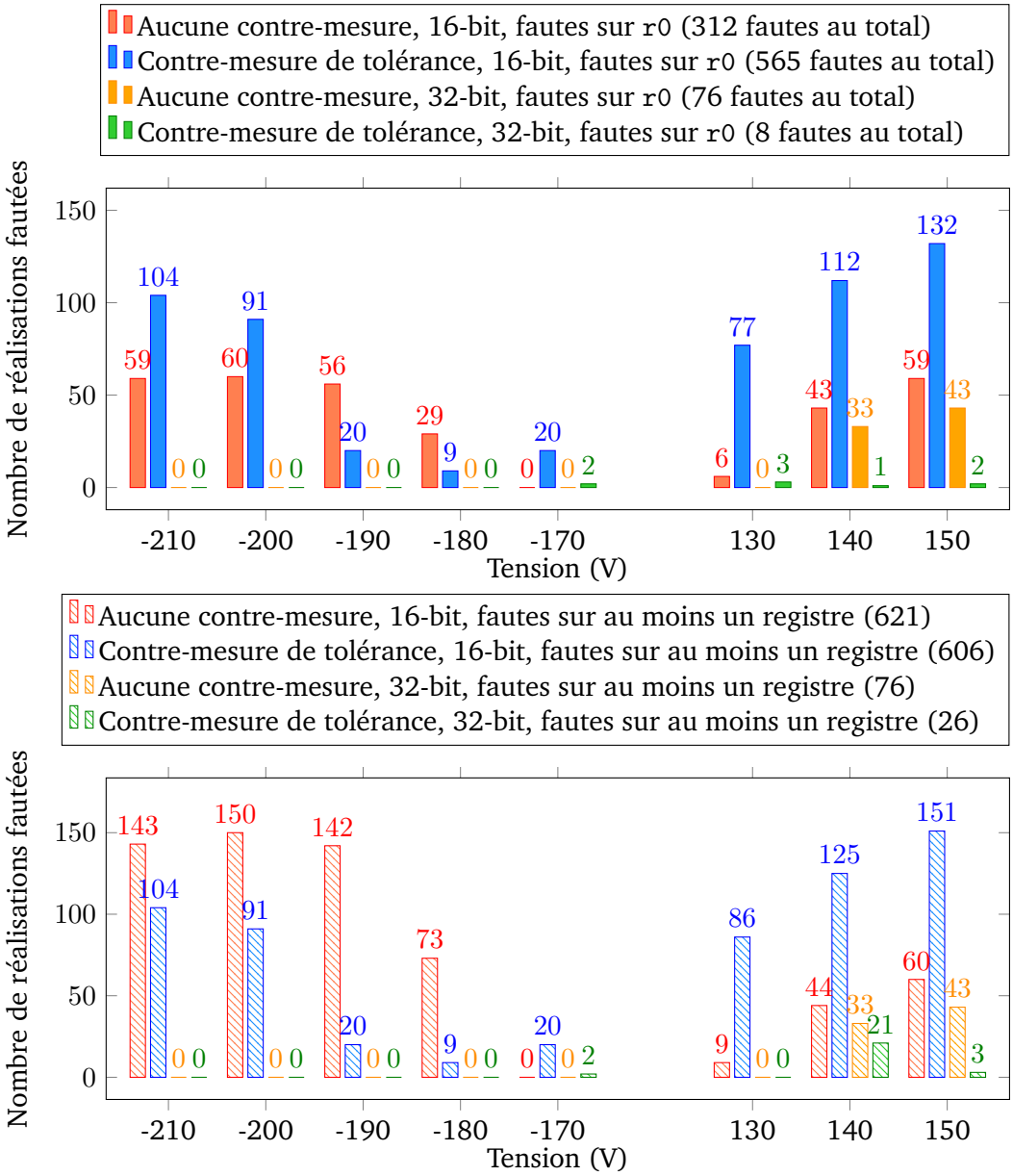


Figure 5.1. : Résultats d'injection de fautes pour la contre-mesure de tolérance

remplacement sont chargées dans une seule étape de *fetch*. Il semble donc qu'une telle double corruption ait eu lieu à la suite de ces attaques.

Pour l'expérimentation sur la séquence de remplacement du listing 5.2 et un encodage forcé sur 32 bits, un total de 26 sorties pour lesquelles au moins un registre a été fauté ont été obtenues (toutes tensions confondues), contre 621 sans contre-mesure et avec un encodage de 16 bits. De même, un total de 8 fautes sur `r0` a été obtenu pour la version renforcée avec encodage sur 32 bits, contre 312 fautes pour la version sans contre-mesure sur 16 bits. La version du code avec contre-mesure et encodage forcé à 32 bits a donc été de façon très nette la variante de code la plus résistante sur cette campagne de tests.

De manière intéressante, des déclenchements d'exceptions ont été obtenus mais aucune sortie fautée n'a été observée pour des impulsions de tension négative sur une instruction 32-bit `bl.w`. Néanmoins, cette instruction a pu être fautée en utilisant des impulsions de tension positive. Une explication pour un tel résultat peut être trouvée dans la façon avec laquelle les instructions sont encodées. Le sous-ensemble des instructions sur 16 bits est très compact et la plupart des valeurs sur 16 bits correspondent à une instruction. En revanche, le sous-ensemble des instructions sur 32 bits est creux (par rapport à l'espace des configurations sur 32 bits de taille 2^{32}). Ainsi, quelques bits fautés peuvent changer une instruction sur 16 bits en une autre instruction valide. En revanche, la plupart des modifications de bits sur une instruction 32 bits génèrent des instructions invalides.

5.2.4.2. Contre-mesure de détection de fautes

L'exemple considéré pour cette première évaluation de la contre-mesure de détection est celui présenté dans le listing 5.1. L'évaluation a ensuite été réalisée sur quatre codes :

— une instruction `ldr` qui charge une valeur depuis la mémoire Flash (temps d'exécution de 150 ns, 1500 impulsions envoyées par valeur de tension)

— une instruction `ldr.w` avec encodage forcé sur 32 bits (temps d'exécution de 150 ns, 1500 impulsions envoyées par valeur de tension)

— la séquence de remplacement du listing 5.1 (temps d'exécution de 300 ns, 3000 impulsions envoyées par valeur de tension)

— la séquence de remplacement du listing 5.1 avec encodage forcé sur 32 bits (temps d'exécution de 500 ns, 5000 impulsions envoyées par valeur de tension)

Les résultats issus des injections de faute sont présentées sur la figure 5.2. Pour l'expérimentation sur la séquence de remplacement du listing 5.1 et un encodage forcé sur 32 bits, un total de 16 réalisations fautées pour `r0` ont été obtenues (toutes tensions confondues), contre 1286 sans contre-mesure et un encodage de 16 bits, 921 avec contre-mesure et un encodage de 16 bits et 1441 sans contre-mesure avec un encodage de 32 bits. On peut observer que, pour des valeurs de tension négatives, l'application de la contre-mesure sans encodage forcé sur 32 bits crée davantage de vulnérabilités que l'instruction `ldr` initiale et n'apporte aucune sécurité. De plus, l'utilisation d'un encodage des instructions sur 32 bits sans contre-mesure n'apparaît efficace ni pour des impulsions de tension négative, ni pour des impulsions de tension positive. Ce résultat peut s'expliquer par le fait que si le fait de forcer un encodage sur 32 bits pour une instruction peut renforcer celle-ci face aux corruptions d'instructions (conformément aux résultats présentés en 5.2.4.1), cela ne fournit aucune protection au niveau du transfert de données réalisé par

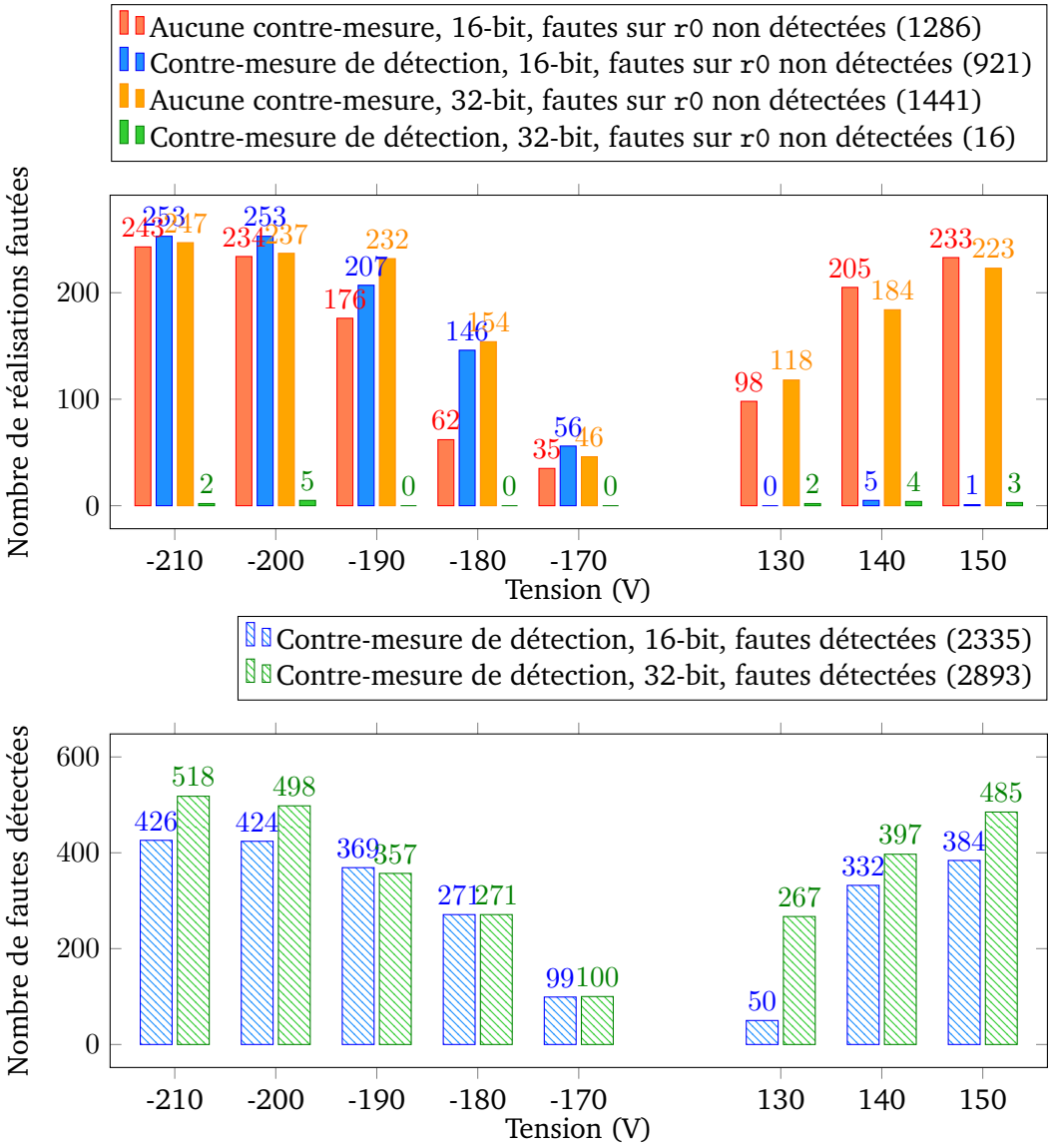


Figure 5.2. : Résultats d'injection de fautes pour la contre-mesure de détection

l'instruction `ldr`. Inversement, l'application de la contre-mesure sans encodage forcé sur 32 bits permet de détecter un grand nombre de sorties fautées mais ne suffit pas à augmenter la robustesse du code pour des tensions d'impulsion négatives. Par ailleurs, le diagramme montrant le nombre de *fautes détectées* indique le nombre d'appels à la sous-fonction d'erreur. Ce second graphique montre que les deux implémentations de la contre-mesure (sur 16 et 32 bits) ont réussi à détecter un grand nombre de fautes et que le mécanisme de détection semble efficace. La version du code avec contre-mesure et encodage forcé à 32 bits a donc été de façon très nette la variante de code la plus résistante sur cette campagne de tests pour l'ensemble des valeurs de tension testées.

5.2.5 Évaluation expérimentale sur une implémentation de FreeRTOS

Les paragraphes qui suivent présentent l'évaluation expérimentale de la robustesse des deux contre-mesures appliquées sur différentes fonctions d'une implémentation de FreeRTOS-MPU. FreeRTOS [3] est un système d'exploitation temps réel (RTOS) portable et *open-source* pour les systèmes embarqués. Il est multitâche, est écrit en C et a été conçu pour rendre plus simple la gestion d'applications temps réel. FreeRTOS utilise un ordonnanceur pour décider de la tâche à exécuter. A chaque interruption de l'horloge système, cet ordonnanceur lance l'exécution à la tâche de plus haute priorité. FreeRTOS-MPU est une version spéciale de FreeRTOS qui utilise la Memory Protection Unit (MPU) et les différents niveaux de privilège disponibles sur le processeur Cortex-M3. FreeRTOS-MPU est donc capable de créer des tâches en mode privilégié ou non-privilégié. Par ailleurs, les programmes embarqués des paragraphes qui suivent qui utilisent FreeRTOS ont été développés à l'aide de la suite IAR WorkBench [4].

Nous avons choisi d'utiliser une implémentation de FreeRTOS pour montrer l'intérêt en pratique que ce type de contre-mesures peuvent avoir pour des codes complexes qui ne peuvent pas être directement conçus en assembleur et pour lesquels peu ou pas de contre-mesures au niveau algorithmique existent. Dans ce cas, ces contre-mesures peuvent être appliquées directement à un binaire compilé pour renforcer sa résistance face aux attaques en faute et permettent un compromis très intéressant entre le renforcement du code et le temps de développement supplémentaire nécessaire. Un cas d'utilisation de la contre-mesure de tolérance est donc proposé en 5.2.5.1 et un autre cas pour la contre-mesure de détection est proposé en 5.2.5.2.

5.2.5.1. Contre-mesure de tolérance à un saut d'instruction

Au démarrage du système, les tâches définies par le programmeur sont initialisées et le processeur exécute cette initialisation en mode privilégié. Ensuite, avant de lancer l'exécution de la première tâche, la fonction `prvRestoreContextOfFirstTask` est appelée. Cette fonction utilise différents types d'instructions et prépare le contexte d'exécution pour lancer la première tâche. Elle fait également basculer le processeur en mode non-privilégié si la première tâche est une tâche qui s'exécute en mode non-privilégié. Si le processeur n'exécute pas de tâche en mode privilégié, le processeur ne re-bascule jamais en mode privilégié comme l'ordonnanceur fonctionne également en mode non-privilégié. Une instruction précise réalise le changement de mode en écrivant dans le registre `CONTROL` [5]. Ainsi, la fonction `prvRestoreContextOfFirstTask` est théoriquement vulnérable à une attaque en faute : un saut d'instruction peut

3. http://www.freertos.org
4. http://www.iar.com
5. Voir en 2.2.2 pour une description du registre `CONTROL`

empêcher l'exécution de l'instruction `msr` [6] qui réalise le basculement vers le mode non-privilégié. Une telle faute permettrait donc l'exécution de toutes les tâches en mode privilégié, autorisant donc certaines actions potentiellement néfastes. Le code compilé correspondant aux dernières instructions de cette fonction est présenté dans le listing 5.3. Néanmoins, l'évaluation de la contre-mesure a été réalisée sur l'ensemble de la fonction, qui comprend 18 instructions.

Listing 5.3 : Code compilé correspondant à la fonction `prvRestoreContextOfFirstTask`

```
1 msr control, r3      ; bascule le processeur en mode non-privilégié
2 msr psp, r0          ; initialise le pointeur de pile
3 mov r0, #0
4 msr basepri, r0      ; met a 0 le reg. pour les exceptions masquees
5 ldr lr, =0xfffffffd  ; prépare le reg. lr pour basculer en Thread mode
6 bx lr                ; sort de la fonc. prvRestoreContextOfFirstTask
```

Des tests d'injection de fautes ont été réalisés sur la fonction complète, pour deux configurations expérimentales :

— Le code sans contre-mesure (temps d'exécution de 2 µs, 8000 impulsions envoyées par valeur de tension)

— Le code renforcé avec la contre-mesure de tolérance et un encodage sur 32 bits pour toutes les instructions (temps d'exécution de 5 µs, 20000 impulsions envoyées par valeur de tension)

Les résultats sont présentés sur la figure 5.3. En totalisant l'ensemble des fautes obtenues en sortie dans le registre `CONTROL` pour les différentes valeurs de tension, on obtient un total de 417 fautes pour le code sans contre-mesure et 307 fautes pour le code avec contre-mesure. Toutefois, en séparant les fautes obtenues pour les tensions positives et négatives, on obtient :

Version sans contre-mesure 20 fautes pour des tensions négatives, 397 pour des tensions positives

Version avec contre-mesure 99 fautes pour des tensions négatives, 208 pour des tensions positives

Ces résultats montrent donc un apport réel de la contre-mesure seulement pour les impulsions de tension positive, avec une légère diminution de la résistance du code pour des tensions négatives. Bien que la modification du registre `CONTROL` constitue le but du scénario d'attaque considéré, l'observation des valeurs de ce registre en sortie rend surtout compte de l'efficacité de la contre-mesure pour renforcer l'instruction `msr` qui écrit dans ce registre `CONTROL`. Nous avons donc également considéré une autre métrique en observant les fautes sur au moins un registre. Cette seconde métrique permet d'observer plus globalement la résistance du code complet de la fonction. En analysant cette métrique, on obtient :

6. L'instruction `msr` copie le contenu d'un registre générique (`r0-r14`) vers un registre spécial

Version sans contre-mesure 1436 fautes pour des tensions négatives, 818 pour des tensions positives

Version avec contre-mesure 1233 fautes pour des tensions négatives, 397 pour des tensions positives

Cette seconde métrique montre un effet très limité de la contre-mesure pour des tensions négatives, et un renforcement du code pour des tensions positives. En faisant la synthèse des résultats obtenus pour ces deux métriques, on peut donc affirmer que la contre-mesure a été efficace pour les impulsions de tension positive, mais n'a pas réellement été efficace pour les impulsions de tension négative.

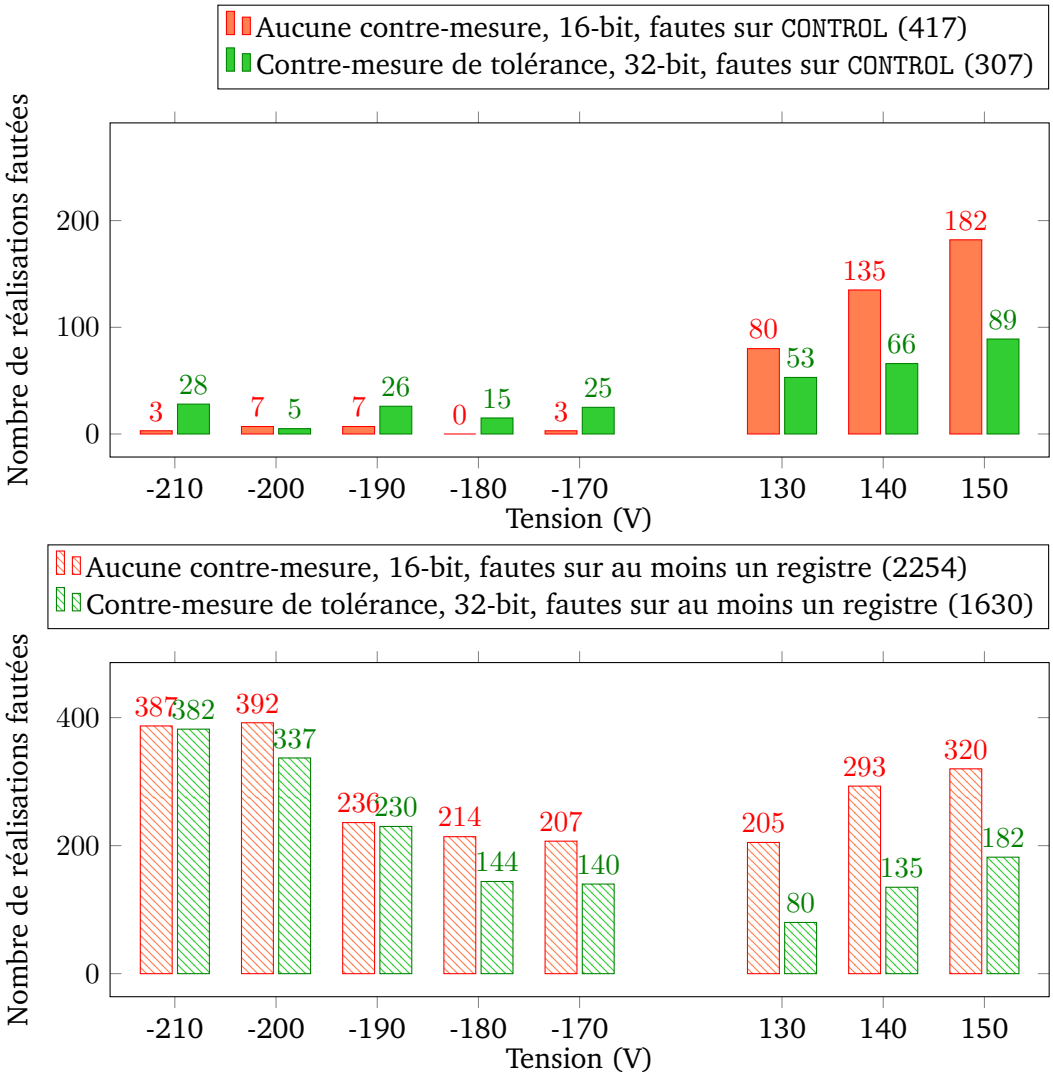


Figure 5.3. : Résultats d'injection de fautes sur la fonction `prvRestoreContextOfFirstTask` d'une implémentation de FreeRTOS

5.2.5.2. Contre-mesure de détection de fautes

Chaque tâche du système est définie avec un niveau de priorité. Une constante, nommée `configMAX_PRIORITIES`, est utilisée pour définir le niveau maximum de priorité que les tâches peuvent prendre. Ainsi, si le système tente de créer une tâche avec un niveau de priorité supérieur, celle-ci est créée avec le niveau de priorité égal à `configMAX_PRIORITIES`. De plus, la fonction `xTaskCreateRestricted` qui réalise la création d'une tâche prend en argument d'entrée une structure qui contient les paramètres de la tâche à créer. Elle utilise ensuite le contenu de cette structure pour appeler la fonction `xTaskGenericCreate`. Dans cette structure, l'entier `uxPriority` est utilisé pour définir le niveau de priorité de la tâche à créer. Pour les tâches qui sont créées à l'initialisation du système, cette structure de configuration est généralement fixée et stockée en mémoire Flash. Ainsi, dans ce cas, l'entier `uxPriority` est chargé via une instruction `ldr` (ligne 7 du listing 5.4). Une injection de faute pourrait donc corrompre cette instruction `ldr` et aboutir à une modification du niveau de priorité pour une tâche. Nous avons choisi cette instruction `ldr` comme instruction à renforcer à l'aide de la contre-mesure de détection. Le code visé pour cette série d'expérimentations correspond au code qui réalise le transfert des arguments à la fonction `xTaskGenericCreate`. Celui-ci est présenté dans le listing 5.4.

Listing 5.4 : Dernières instructions avant l'appel à `xTaskGenericCreate`

```
1  movs r0, #0          ; pointeur vide
2  str  r0, [sp, #12]   ; (correspond aux zones mem. a acces restreint)
3  movs r0, #0          ; pointeur vide
4  str  r0, [sp, #8]    ; (zone mém. utilisée pour la pile de la tâche)
5  movs r0, #0          ; pointeur vide
6  str  r0, [sp, #4]    ; (fonction de handler pour la tâche)
7  ldr  r0, =adresse_uxPriority
8  ldr  r0, [r0, #0]    ; charge uxPriority dans r0
9  str  r0, [sp, #0]    ; met uxPriority sur la pile
10 movs r3, #0          ; pointeur vide (autres paramètres)
11 movs r2, #128        ; profondeur de pile pour la tâche
12 movs r1, #0          ; chaîne vide (nom de la tâche)
13 ldr  r0, =adresse_fonction_tache  ; fonction de la tâche
14 bl   xTaskGenericCreate
```

Des attaques par injection de fautes ont donc été réalisées sur ces 14 instructions assembleur (qui incluent 3 instructions `ldr` dont 2 qui chargent des données depuis la mémoire Flash) qui transmettent les arguments d'entrée pour la fonction `xTaskGenericCreate`. Le point auquel les données sont observées a été fixé au début de la fonction `xTaskGenericCreate`. A ce point, on observe la valeur de `uxPriority` transmise sur la pile à la fonction `xTaskGenericCreate`. Trois configurations expérimentales ont été utilisées :

— Sans contre-mesure (temps d'exécution de 1 µs, 4000 impulsions envoyées par valeur de tension)

— Avec la contre-mesure de détection et un encodage forcé sur 32 bits pour les instructions `ldr` seulement (temps d'exécution de 2 µs, 8000 impulsions envoyées par valeur de tension)

— Avec la contre-mesure de détection et un encodage forcé sur 32 bits pour toutes les instructions (temps d'exécution de 4 µs, 16000 impulsions envoyées par valeur de tension)

Les résultats sont présentés sur la figure 5.4. Le premier graphique montre que 5216 réalisations fautées ont été obtenues pour l'implémentation sans contre-mesure, contre 2480 pour celle pour laquelle les `ldr` ont été renforcés et 83 pour celle pour laquelle toutes les instructions ont été renforcées. Le second graphique montre que le mécanisme de détection s'est révélé très efficace, avec un très grand nombre de fautes détectées pour les deux implémentations testées. Ces résultats permettent donc d'affirmer que la contre-mesure seulement appliquée aux instructions `ldr` contribue à réduire le nombre de fautes obtenues en sortie sans pour autant l'annuler. Dans ce cas, les fautes restantes sont dues à la corruption des autres instructions visées. Le résultat obtenu en appliquant la contre-mesure à toutes les instructions confirme clairement cette affirmation, puisque dans ce cas la contre-mesure est très efficace, avec au maximum 16 réalisations fautées obtenues pour chaque valeur de tension testée sur 16000 impulsions envoyées. Néanmoins, le fait d'appliquer cette contre-mesure à l'ensemble du code testé a nécessité de multiplier par 4 le nombre total d'instructions, le surcoût en termes de performance et taille de code est donc relativement important.

5.2.6 Bilan sur l'évaluation des contre-mesures

La contre-mesure de tolérance à un saut d'instruction proposée au chapitre 4 s'est montrée très efficace pour protéger une instruction d'appel de sous-fonction `bl` isolée. Il semble donc qu'une instruction sensible comme `bl` puisse être significativement renforcée contre les attaques en faute. Néanmoins, sur le code plus complexe qui a été testé précédemment, ses résultats ont été plus mitigés. Ce résultat vient très certainement du modèle simplifié de saut d'instruction qui ne couvre peut-être pas une partie des fautes qui ont été obtenues sur ce code. Une meilleure compréhension du modèle de faute pourrait certainement contribuer à améliorer cette contre-mesure.

La contre-mesure de détection de fautes a été conçue pour protéger un ensemble plus réduit d'instructions. Celle-ci s'est montrée très efficace sur les cas testés. Sur un code plus complexe qui contient seulement des instructions pour lesquelles elle peut être utilisée, cette contre-mesure augmente grandement le niveau de sécurité. Néanmoins, ses principaux inconvénients sont liés au fait que certaines instructions ne peuvent pas être protégées avec cette contre-mesure (notamment l'instruction `bl`

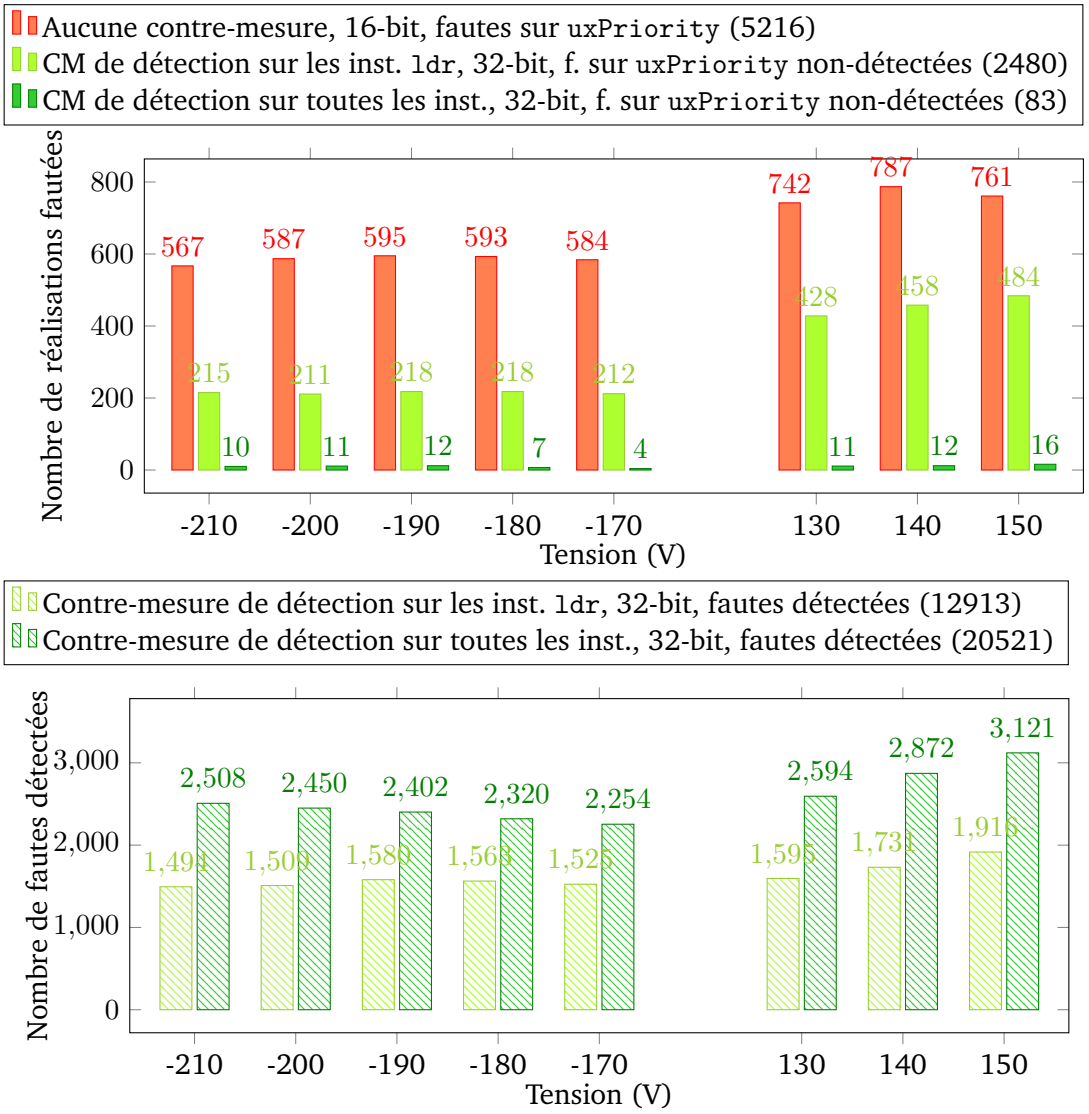


Figure 5.4. : Résultats d'injection de fautes sur les instructions qui précèdent l'appel à la fonction `xTaskGenericCreate` d'une implementation de FreeRTOS

pour laquelle la contre-mesure de tolérance s'est montrée efficace) ou à son surcoût très élevé.

5.3 Application combinée des deux contre-mesures

La partie précédente a évalué séparément les deux contre-mesures sur des codes pour lesquelles elles ont été pensées. Dans cette section, nous proposons d'exploiter les complémentarités de ces deux contre-mesures pour renforcer un même code.

5.3.1 Présentation de l'implémentation à renforcer

Même si ces contre-mesures au niveau assembleur ne se destinent pas spécifiquement au renforcement d'implémentations cryptographiques (notamment parce que leur

impact au niveau des attaques par observation n'a pas encore été évalué), nous avons choisi de les appliquer à une implémentation de l'algorithme AES pour faire écho aux attaques présentées en 2.4. Plus spécifiquement, on s'intéresse ici au renforcement de la fonction `addRoundKey` de la dernière ronde et au renforcement de l'instruction `bl` qui réalise le branchement vers cette fonction. Par ailleurs, on considère une implémentation pour laquelle la boucle de la fonction `addRoundKey` a été déroulée. Enfin, une seule clé et un seul texte ont été utilisés, de la même manière qu'en 2.4.2. La structure générale du code à renforcer est présentée dans le listing 5.5. Par ailleurs, suite aux résultats obtenus en 5.2, un encodage sur 32 bits a été utilisé pour toutes les instructions (à l'exception de l'instruction `bx lr` qui ne dispose pas d'encodage sur 32 bits). Comme il s'agit de l'appel à la fonction `addRoundKey` de la dernière ronde, celle-ci est directement suivie d'une sortie de la fonction de chiffrement AES.

Listing 5.5 : Implémentation à renforcer d'une fonction addRoundKey

```
1      .....                    ; parties précédentes de l'AES
2      mov.w    r1, sp          ; pointeur vers le tableau de clé
3      mov.w    r0, r6          ; pointeur vers le tableau de texte
4      bl.w     addRoundKey     ; branchement vers la fonction
5      bx       lr              ; sortie de la fonction de chiffrement
6
7
8  addRoundKey
9      ldrb.w   r2, [r0, #0]    ; chargement du premier octet de texte
10     ldrb.w   r3, [r1, #0]    ; chargement du premier octet de clé
11     eors.w   r2, r2, r3      ; OU EXCLUSIF entre les deux octets
12     strb.w   r2, [r0, #0]    ; stockage du résultat en mémoire
13
14     ldrb.w   r2, [r0, #1]    ; chargement du deuxième octet de texte
15     ldrb.w   r3, [r1, #1]    ; chargement du deuxième octet de clé
16     eors.w   r2, r2, r3      ; OU EXCLUSIF entre les deux octets
17     strb.w   r2, [r0, #1]    ; stockage du résultat en mémoire
18
19     .....                    ; suite de la fonction addRoundKey
20     bx       lr              ; sortie de la fonction addRoundKey
```

5.3.2 Évaluation préliminaire de l'implémentation non renforcée

Une première évaluation expérimentale du code présenté dans le listing 5.5 a été réalisée. Les paramètres expérimentaux utilisés pour celle-ci sont similaires à ceux présentés en 5.2.4, à l'exception de l'intervalle d'injection, la tension de l'impulsion et le nombre d'injections par instant. Les nouvelles valeurs pour ces paramètres sont présentées dans le tableau 5.2.

Pour cette étape d'évaluation, les exceptions matérielles de type `UsageFault` ont été utilisées pour retrouver l'instruction correspondant à un instant d'injection.

Table 5.2. : Paramètres expérimentaux utilisés pour le renforcement de l'implémentation d'AES

Instant d'injection	Intervalle de 2.8 µs parcouru par pas de 500 ps
Tension de l'impulsion	−210 V
Nombre d'injections	1 essai par instant d'injection et tension de l'impulsion

L'intervalle d'injection a ainsi été défini de façon à couvrir l'ensemble des instructions de la fonction `addRoundKey`, ainsi que les instructions d'appel à celle-ci.

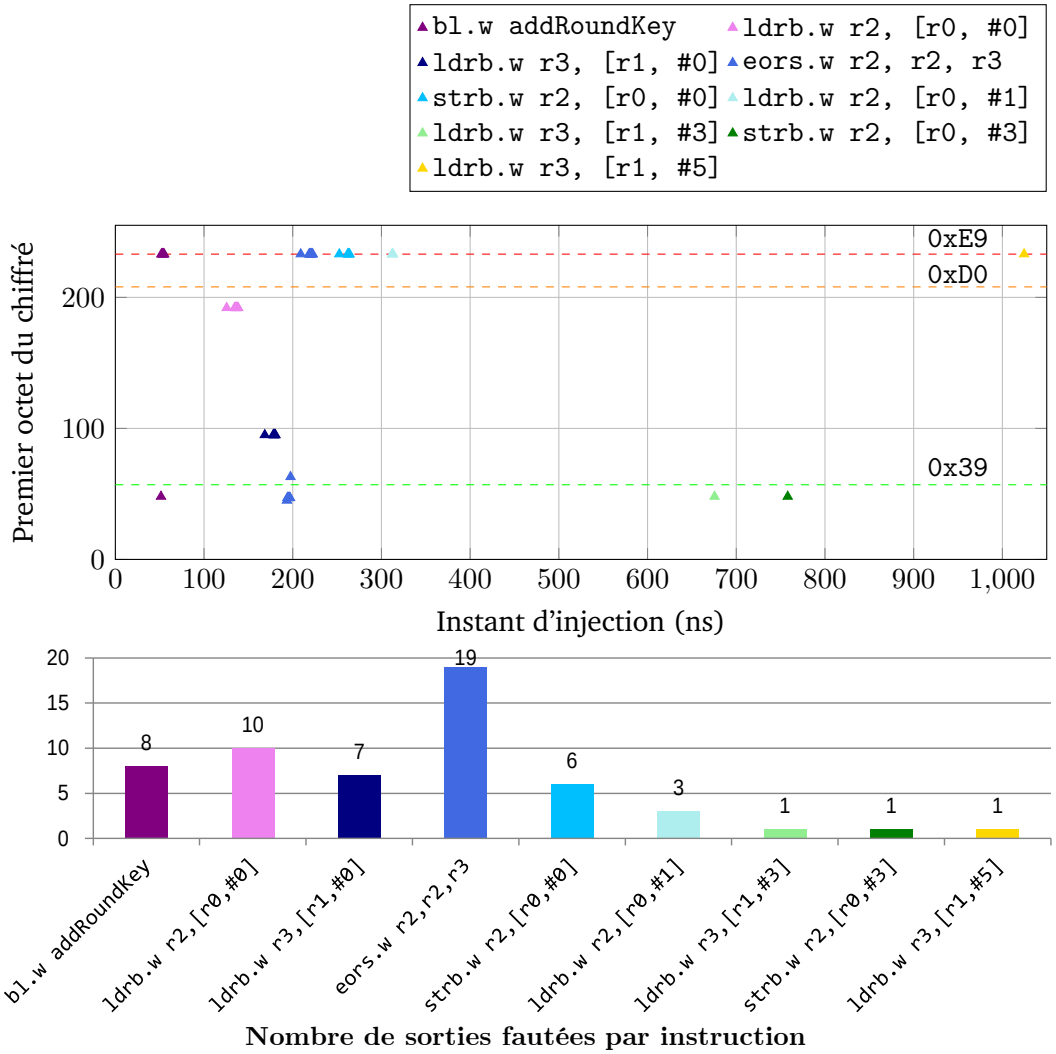


Figure 5.5. : Résultats d'injection de fautes sur la fonction `addRoundKey` sans contre-mesure pour le premier octet du texte chiffré

Pour l'injection de fautes sur ce code embarqué, un intervalle temporel de 2.8 µs a été parcouru. Un aperçu d'une partie des résultats d'injection est présenté sur la figure 5.5. Ce graphique représente les différentes valeurs fautées obtenues pour le premier octet du texte chiffré. La valeur attendue en sortie est `0x39` (qui correspond à la ligne verte sur le graphique), la valeur pour laquelle une cryptanalyse immédiate est possible est `0xE9` (ligne rouge sur le graphique), et la valeur du premier octet

de la clé de ronde est `0xD0` (ligne orange sur le graphique). Par ailleurs, un second graphique répartit les valeurs fautées en plusieurs groupes qui correspondent chacun à une instruction. Pour un nombre total de 3816 injections de faute sur l'ensemble de la fonction, 57 fautes ont été obtenues pour ce premier octet, correspondant à 7 valeurs fautées différentes. La valeur `0xE9` a été obtenue pour 28 sorties fautées. Comme cette valeur permet de réaliser une cryptanalyse immédiate, ce type de faute est très intéressant pour un attaquant et le fait d'obtenir cette valeur à la suite d'une injection de fautes indique que la fonction présente des vulnérabilités. Au total, 653 sorties de chiffrement comptant au moins un octet fauté ont été obtenues. Pour certaines de ces sorties, l'opération d'addition de clé de ronde n'a été effectuée sur aucun octet, ce qui correspond à une non-exécution de l'ensemble de la fonction. Par ailleurs, on peut observer qu'un plus grand nombre de sorties fautées sur le premier octet est obtenu pour les 4 premières instructions de la fonction `addRoundKey` qui traitent ce premier octet.

5.3.3 Application de la contre-mesure de tolérance au saut d'une instruction

A la suite des résultats d'injection illustrés précédemment sur l'implémentation non renforcée, une première étape de renforcement a été appliquée de manière automatique. Celle-ci consiste à appliquer le schéma de contre-mesure présenté dans le chapitre 4 au code visé. La structure du code qui résulte de cette transformation est présentée dans le listing 5.6. La contre-mesure a donc été utilisée pour protéger à la fois le branchement vers la fonction `addRoundKey` ainsi que les instructions qui composent la fonction, et toutes les instructions ont été renforcées.

Listing 5.6 : Implémentation d'une fonction addRoundKey avec une première étape de renforcement

```
1      .....                          ; parties précédentes de l'AES
2      adr.w r12, label_retour
3      adr.w r12, label_retour
4      add.w lr, r12, #1
5      add.w lr, r12, #1
6      b.w   addRoundKey
7      b.w   addRoundKey
8
9  label_retour
10     bx       lr                    ; sortie de la fonction d'AES
11     bx       lr
12
13
14 addRoundKey
15     ldrb.w  r2, [r0, #0]
16     ldrb.w  r2, [r0, #0]
17     ldrb.w  r3, [r1, #0]
18     ldrb.w  r3, [r1, #0]
```

```
19      eors.w  r12, r2, r3
20      eors.w  r12, r2, r3
21      strb.w  r12, [r0, #0]
22      strb.w  r12, [r0, #0]
23
24      ldrb.w  r2, [r0, #1]
25      .....
26
27      .....                           ; suite de la fonction addRoundKey
28      bx      lr
29      bx      lr
```

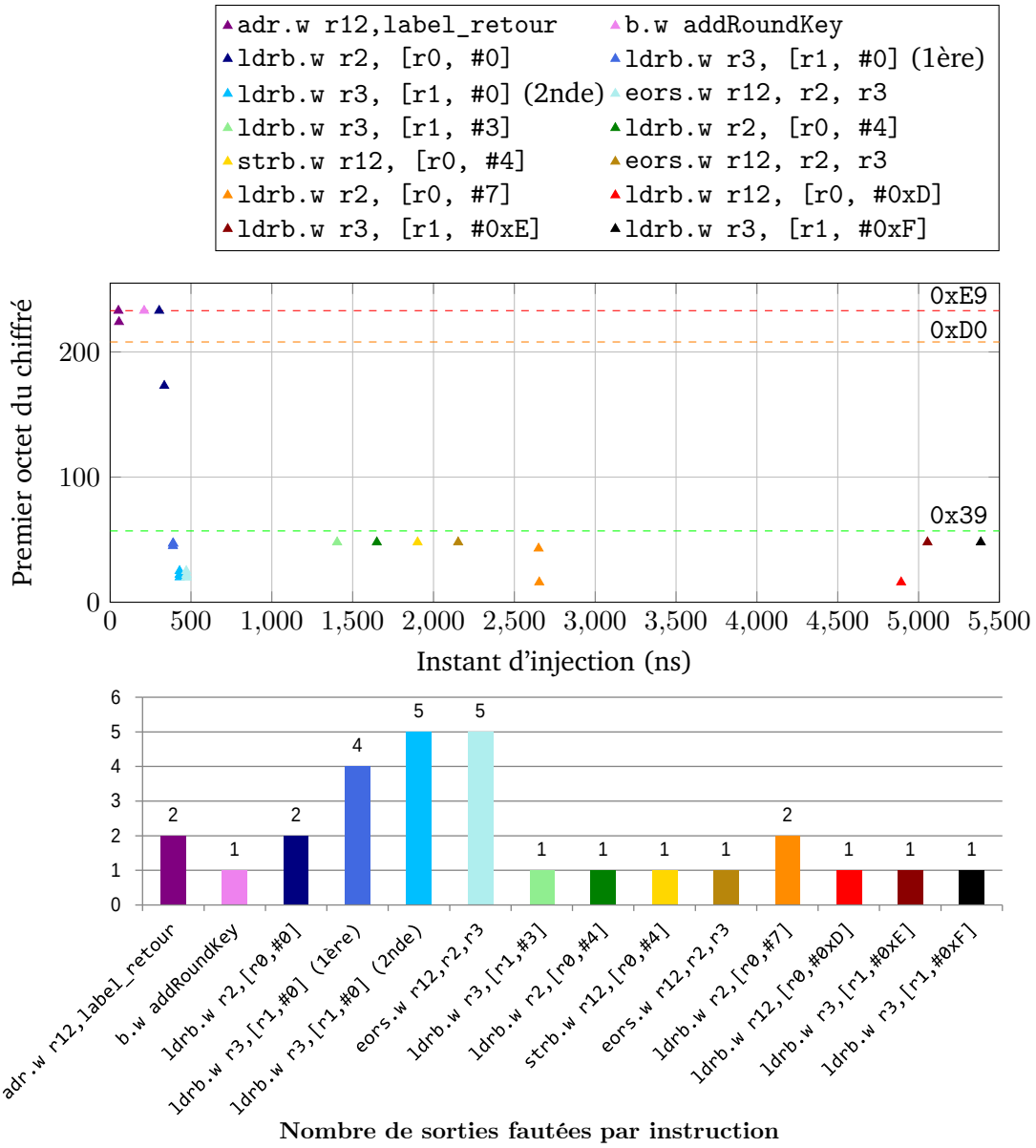


Figure 5.6. : Résultats d'injection de fautes pour le premier octet du texte chiffré sur la fonction `addRoundKey` avec application automatique de la contre-mesure de tolérance

Pour l'évaluation réalisée à la suite de cette première étape de renforcement, un intervalle de 5.6 µs a été parcouru. Un aperçu d'une partie des résultats d'injection est présenté sur la figure 5.6. Comme pour la partie précédente, ce graphique représente les différentes valeurs fautées obtenues pour le premier octet du texte chiffré. Pour 11200 injections de fautes sur l'ensemble de la fonction, 352 sorties contenant au moins un octet fauté ont été obtenues. Plus spécifiquement, 28 fautes au lieu de 57 ont été obtenues sur le premier octet, correspondant à 11 valeurs fautées. La valeur `0xE9` a été obtenue seulement pour 3 injections de faute. Il y a donc une diminution globale du nombre de fautes obtenues par rapport à l'implémentation sans contre-mesure.

Comme l'application automatique de la contre-mesure augmente le nombre d'instructions du code, une partie significative des fautes obtenues pour cette campagne l'ont été suite à la perturbation d'instructions qui ne traitaient pas le premier octet. On peut également constater qu'une grande partie des fautes restantes ont été obtenues suite à la perturbation d'instructions du type `ldrb`. Nous proposons de renforcer de manière supplémentaire le code en appliquant un deuxième niveau de contre-mesure.

5.3.4 Renforcement à l'aide de la contre-mesure de détection de fautes

Un second niveau de renforcement est donc appliqué sur le code précédent. Celui-ci consiste à appliquer la contre-mesure de détection sur les points vulnérables à la suite de l'étape de renforcement précédente. Contrairement à la première étape de renforcement qui a été appliquée de manière automatique, cette seconde étape est appliquée manuellement de manière à renforcer plus spécifiquement les deux instructions les plus vulnérables à la suite de l'évaluation précédente (`ldrb.w r3,[r1,#0]` et `eors.w r3,r2,r3`). Ce second niveau de renforcement est illustré sur le listing 5.7.

Listing 5.7 : Implémentation d'une fonction addRoundKey avec une seconde étape de renforcement

```
1  addRoundKey
2      ldrb.w  r2, [r0, #0]
3      ldrb.w  r2, [r0, #0]
4      ldrb.w  r3, [r1, #0]     ; instruction renforcée avec
5      ldrb.w  r12, [r1, #0]    ; la contre-mesure de détection
6      cmp.w   r3, r12
7      bne.w   error
8      eors.w  r12, r2, r3      ; instruction renforcée avec
9      eors.w  r3, r2, r3       ; la contre-mesure de détection
10     cmp.w r3, r12
11     bne.w error
12     strb.w  r12, [r0, #0]
```

```
13      strb.w  r12, [r0, #0]
14
15      ldrb.w  r2, [r0, #1]
16      .....
```

Pour l'évaluation réalisée à la suite de cette seconde étape de renforcement, un intervalle de 8.54 µs a été parcouru. Pour 17080 injections de fautes sur l'ensemble de la fonction, 30 fautes ont été obtenues sur le premier octet, correspondant à 3 valeurs fautées. En revanche, parmi ces 30 fautes, 24 ont été détectées par la contre-mesure. Il reste donc un total de 6 fautes non-détectées correspondant à 2 valeurs fautées.

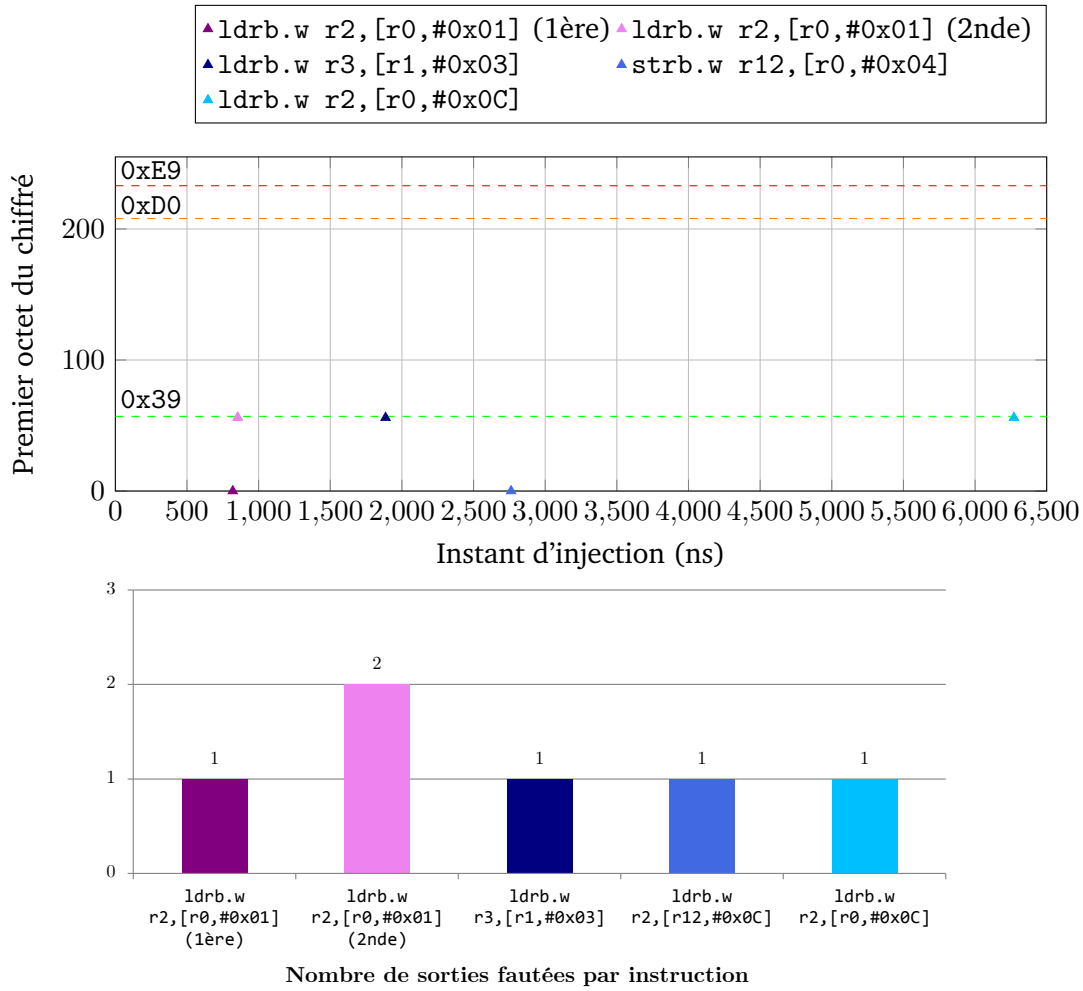


Figure 5.7. : Résultats d'injection de fautes pour le premier octet du texte chiffré sur la fonction `addRoundKey` avec application d'un second niveau de renforcement

Un aperçu d'une partie des résultats d'injection est présenté sur la figure 5.7. Sur ce graphique, il est intéressant de constater que l'ensemble des fautes non-détectées obtenues sur cette fonction ont perturbé des instructions qui ne traitaient pas le premier octet. Parmi celles-ci, 3 sorties fautées ont été obtenues suite à la perturbation d'instructions `ldrb` seulement protégées avec la contre-mesure de tolérance. Ces instructions pourraient donc probablement également être renforcées avec la

contre-mesure de détection pour davantage d'efficacité. Il est également intéressant de constater que la valeur `0xE9` n'a jamais été obtenue en sortie du premier octet : aucune des fautes obtenues sur ce code renforcé ne permet de cryptanalyse immédiate.

5.3.5 Bilan sur l'application combinée des deux contre-mesures

Cette application combinée des deux contre-mesures pour renforcer un code embarqué a permis de montrer que, sur le code testé et dans les conditions expérimentales considérées, la contre-mesure de tolérance permettait de réduire le nombre de fautes en sortie mais ne suffisait pas à protéger totalement l'exécution de l'algorithme. Sur ce code testé, la contre-mesure de tolérance a semblé ne pas suffire pour protéger les instructions `ldrb` ou `eors` mais s'est montrée efficace pour la protection du branchement vers la fonction d'`addRoundKey`. Une deuxième étape de renforcement, à l'aide de la contre-mesure de détection, a été nécessaire.

Bien entendu, les résultats présentés dans cette section doivent être considérés comme une contribution à l'évaluation expérimentale de ces contre-mesures mais il convient de rester prudent vis-à-vis de toute généralisation. Le cas d'une fonction cryptographique présente des contraintes particulières au niveau des types de fautes pouvant permettre une cryptanalyse. De plus, le code précédent n'a été testé que pour un seul texte clair et une seule valeur de clé. Malgré ces limitations, les expérimentations présentées dans cette section s'inscrivent dans la lignée de celles présentées en 5.2 et on retrouve clairement une certaine cohérence au niveau des résultats obtenus. A la suite de ces résultats expérimentaux, les différents avantages et inconvénients des deux contre-mesures sont résumés dans le tableau 5.3.

La contre-mesure de tolérance semble efficace pour la protection des appels de sous-fonction, et celle-ci présente l'avantage de pouvoir automatiquement être appliquée à un code générique. Pour un coût en temps de développement particulièrement minime et un coût en termes de performances acceptable, il est donc possible de renforcer un code sans connaissance a priori de ses vulnérabilités. Comme cette contre-mesure ne protège pas face aux fautes au niveau du flot de données, une protection des instructions de type `ldr` depuis la mémoire Flash à l'aide de méthodes de détection est impérative pour en améliorer la sécurité.

La contre-mesure de détection, bien que visiblement plus efficace sur les instructions arithmétiques et logiques, reste bien plus difficile à appliquer automatiquement. Celle-ci peut nécessiter davantage de registres supplémentaires et ne s'applique pas à certaines instructions importantes (comme les instructions de branchement ou de manipulation de la pile). Enfin, elle réalise une détection de fautes et non une tolérance, ce qui oblige à concevoir un système de récupération en cas d'erreur détectée.

Au vu des résultats obtenus dans ce chapitre, si le but est de maximiser le niveau de sécurité sans contrainte forte sur le temps de développement, l'utilisation de la contre-mesure de détection combinée avec des séquences de remplacement issues de la contre-mesure de tolérance pour les instructions où la contre-mesure de détection ne peut pas s'appliquer nous paraît être une solution pertinente pour protéger un code embarqué dans les conditions expérimentales considérées pour ces travaux de thèse.

Néanmoins, ces observations sur les forces et faiblesses des deux contre-mesures sont la conséquence d'un nombre réduit d'expérimentations, et il est certain que des campagnes de tests plus poussées sur davantage de types de codes seraient nécessaires pour évaluer plus rigoureusement ces deux contre-mesures. De même, si la contre-mesure de détection semble s'être révélée plus efficace pour renforcer la majeure partie des instructions des codes testés, il serait intéressant de réaliser des tests similaires à ceux présentés dans ce chapitre en utilisant d'autres moyens d'injection de fautes voire d'autres types de processeurs embarqués.

Table 5.3. : Avantages et inconvénients des deux contre-mesures testées

	Contre-mesure de tolérance	**Contre-mesure de détection**
Avantages	— S'applique à tout le jeu d'instructions — Facilement automatisable — Très efficace sur une instruction `bl`	— Très efficace sur les codes testés — Possiblement automatisable
Inconvénients	— Moyennement efficace sur le code complexe testé — Surcoût élevé — Ne protège pas le flot de données	— Pas définie pour plusieurs instructions importantes (notamment `push` et `bl`) — Surcoût très élevé

5.4 Étude de vulnérabilité face aux attaques par observation

Dans les parties précédentes de ce chapitre, la résistance face aux attaques par injection de faute de la contre-mesure proposée dans le chapitre 4 a été étudiée. Or, pour cette contre-mesure une même donnée peut être réutilisée par un même type d'instruction sur deux cycles d'horloge consécutifs. Il est donc possible que la contre-mesure augmente la vulnérabilité d'une implémentation cryptographique face aux attaques par observation. Cette section présente donc une analyse simple des effets de cette contre-mesure concernant les attaques par observation au premier ordre. Pour réaliser cette étude, des attaques par observation sont réalisées sur deux

implémentations de l'algorithme AES dans sa version AES-256, l'une d'entre-elles ayant été renforcée contre les attaques en faute à l'aide de la contre-mesure présentée dans cette thèse. L'implémentation AES-256 de base correspond à celle utilisée pour le concours *DPA Contest V4* [7], que nous avons portée sur Cortex-M3 et pour laquelle le masque appliqué à la première ronde a été fixé à 0.

5.4.1 Paramètres utilisés

Processus d'attaque par observation Les attaques ont été réalisées au moment du traitement du premier octet de texte par la fonction `SubBytes` pendant la première ronde du chiffrement. Pour cette première ronde de chiffrement, le premier octet de la clé a la valeur `0x00`. L'attaque réalisée est une analyse de type Correlation Power Analysis (CPA) et utilise donc la corrélation de Pearson comme distingueur statistique (Brier et al., 2004). Par ailleurs, un modèle de poids de Hamming a été utilisé pour le calculs des prédictions.

Paramètres expérimentaux Ces attaques sont réalisées par analyse des rayonnements électromagnétiques émis par le circuit, avec un banc d'analyse similaire à celui présenté dans (Aboulkassimi et al., 2011). Une antenne d'analyse est placée au-dessus du circuit, celle-ci est reliée à un oscilloscope.

5.4.2 Résultats expérimentaux

2000 courbes de mesures sont utilisées pour chacune des implémentations, et les acquisitions ont été réalisées avec une fréquence d'échantillonnage de 20 Gigasample/s, pendant une période de $1\ \mu s$ pour l'implémentation sans contre-mesure et une période de $2\ \mu s$ pour la période avec contre-mesure. Ces intervalles temporels ont été choisis de façon à pouvoir observer le passage du premier octet à travers la fonction `SubBytes`. Un signal de déclenchement (analogue à celui présenté en 2.3.1.1) a été utilisé pour définir ces fenêtres temporelles.

5.4.2.1. Implémentation sans contre-mesure de tolérance

Les résultats de l'attaque pour l'implémentation sans contre-mesure de tolérance au saut d'une instruction sont présentés sur la figure 5.8. Ce graphique montre les coefficients de corrélation associés à chaque hypothèse pour le premier octet de la clé en fonction de l'instant d'échantillonnage. Le plus haut coefficient de corrélation est obtenu pour l'octet de clé `0x00` (dont la courbe de corrélation associée est représentée en rose). Pour cette attaque sur une version sans contre-mesure, le premier octet de clé a donc pu être retrouvé avec 2000 courbes et un modèle de poids de Hamming.

7. http://www.dpacontest.org

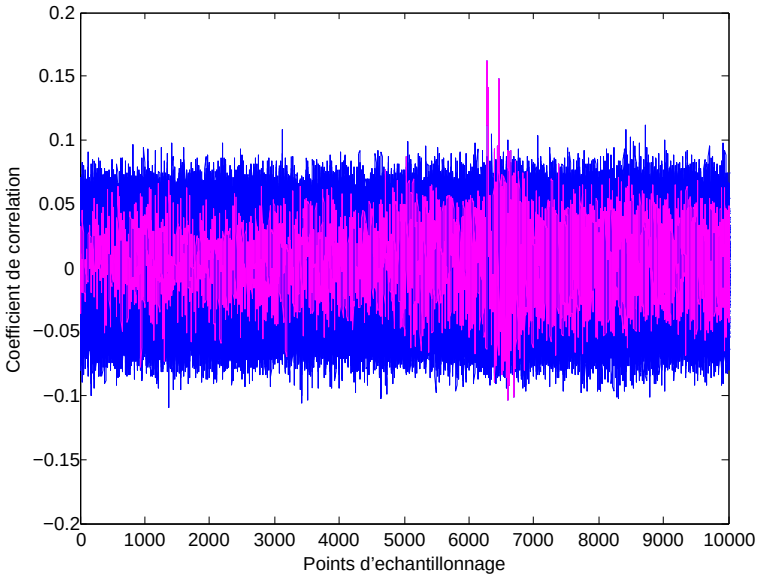


Figure 5.8. : Courbes de corrélation pour l'implémentation d'AES sans contre-mesure

5.4.2.2. Implémentation avec contre-mesure de tolérance

Les résultats de l'attaque pour l'implémentation avec contre-mesure de tolérance sont présentés sur la figure 5.9. Le plus haut coefficient de corrélation est obtenu pour l'octet de clé `0xAB` (dont la courbe de corrélation associée est représentée en rose). Pour cet octet `0xAB`, le coefficient maximal obtenu est de $0,2$. L'hypothèse de clé `0x00` a un coefficient de corrélation maximal de $0,14$ et arrive en $55^{\text{ème}}$ position parmi les différentes hypothèses de clé. Pour cette implémentation avec contre-mesure de tolérance, l'attaque n'a pas permis de retrouver le premier octet de la clé. L'ajout de la contre-mesure n'a donc ici pas rajouté de vulnérabilité particulière face à une attaque par observation.

5.4.3 Bilan

L'étude de vulnérabilité face aux attaques par observation présentée dans cette section contribue à renforcer la confiance accordée à la contre-mesure de tolérance présentée dans cette thèse. Bien entendu, cette étude n'est qu'une ébauche d'analyse et une campagne de tests plus avancée doit être mise en place pour réellement pouvoir conclure quant à l'influence de cette contre-mesure au niveau des attaques par observation.

5.5 Conclusion et perspectives

Dans ce chapitre, une première évaluation du schéma de contre-mesure présenté dans cette thèse a été réalisée. Au niveau des attaques par injection de fautes, cette évaluation a été réalisée sur des instructions isolées puis des codes plus complexes

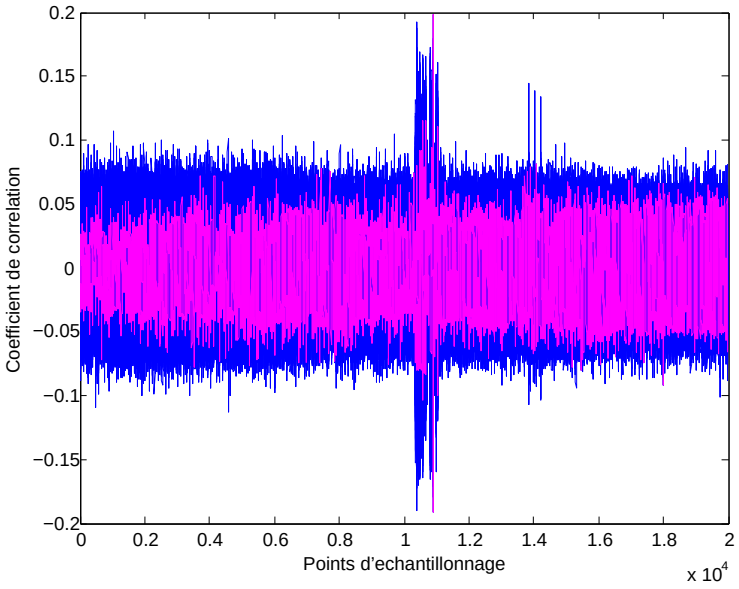


Figure 5.9. : Courbes de corrélation pour l'implémentation d'AES avec contre-mesure

issus d'une implémentation de FreeRTOS. Ensuite, la contre-mesure a été complétée à l'aide d'une contre-mesure de détection pour renforcer par étapes un code embarqué. Enfin, une évaluation simple a été réalisée au niveau des attaques par observation.

Les expérimentations de ce chapitre ont tout d'abord permis de montrer qu'il était impératif de forcer un encodage des instructions sur 32 bits pour ne pas annuler l'efficacité de toute éventuelle contre-mesure. Cette observation tend notamment à confirmer les résultats sur le modèle de fautes obtenus dans le chapitre 3. Les résultats obtenus montrent que la contre-mesure de tolérance contribue à réduire le nombre de fautes obtenues sur un code embarqué. Toutefois, si elle s'est montrée très efficace pour protéger des instructions de branchement vers une sous-fonction, son efficacité pour protéger des instructions sur un exemple de code complexe a été plus limitée. En revanche, une contre-mesure de détection qui avait été envisagée comme complément à celle proposée dans cette thèse s'est révélée être très efficace pour les exemples de codes testés.

Une combinaison de ces deux contre-mesures a permis de grandement renforcer l'implémentation de la fonction `addRoundKey` présentée dans le chapitre 2. Avec la plate-forme expérimentale et le microcontrôleur utilisés dans cette thèse, la contre-mesure de tolérance au saut d'une instruction peut être envisagée comme un complément à la contre-mesure de détection. Elle peut être utilisée pour toutes les instructions qu'il n'est pas possible de protéger à l'aide de la contre-mesure de détection. Cependant, des tests complémentaires sur un plus grand nombre de codes ainsi qu'en utilisant différents moyens d'injection de fautes voire différents types d'architectures permettraient de mieux évaluer l'efficacité en pratique de ces deux contre-mesures.

Enfin, les attaques par observation réalisées n'ont pas montré de vulnérabilité évidente pour un code renforcé à l'aide de la contre-mesure de tolérance au saut d'une instruction. Néanmoins, davantage de tests en utilisant différents types d'attaques par observation seraient nécessaires pour pouvoir mieux évaluer l'impact de la contre-mesure au niveau des attaques par observation.

Conclusion et perspectives

6

L'objectif de cette thèse était de contribuer à renforcer la sécurité de codes embarqués face aux nouvelles menaces que sont les attaques par injection électromagnétique. Nous rappelons brièvement notre démarche et nos contributions et présentons les principales perspectives de ce travail.

6.1 Conclusion

Dans le premier chapitre de cette thèse, une brève présentation du domaine de la sécurité des systèmes embarqués et un état de l'art plus détaillé des différentes attaques par canaux auxiliaires et des contre-mesures existantes sur circuits intégrés ont été réalisés. Les attaques par canaux auxiliaires peuvent ainsi se diviser en deux catégories : celles dites par *observation* et celles dites par *injection de fautes*. Par la suite, un positionnement par rapport a cet état de l'art a été présenté, visant à justifier le cadre de travail utilisé dans cette thèse. Nous avons donc choisi d'étudier plus spécifiquement les nouvelles menaces que sont les attaques par impulsion électromagnétique, contre lesquelles très peu de contre-mesures ont été proposées jusqu'à présent. Nous avons également choisi, pour étudier ces nouvelles menaces, de considérer un circuit à l'état de l'art en utilisant un microcontrôleur basé sur le processeur ARM Cortex-M3.

Le deuxième chapitre s'intéresse ensuite à la mise en place d'un banc d'injection de fautes par impulsion électromagnétique et en présente les différents éléments constitutifs. Ce banc utilise une antenne d'injection sous forme de solénoïde associée à un générateur de signaux pulsés. Pour pouvoir évaluer l'impact d'une injection de faute sur un code embarqué, un processus expérimental spécifique a été mis en place. Celui-ci permet ainsi de récupérer plusieurs informations sur l'état du microcontrôleur et se base sur un programme de pilotage du banc d'injection qui a été développé pour les besoins de la thèse. Deux exemples d'attaques permettant de vérifier le bon fonctionnement du banc sont présentées en fin de chapitre. La première d'entre-elles a perturbé l'incrémentation du compteur de ronde lors d'un chiffrement AES et montré la possibilité pour ce banc de corrompre l'exécution d'une structure de boucle. La seconde a visé une fonction de la dernière ronde d'un chiffrement AES muni d'une contre-mesure de redondance temporelle à l'échelle de la fonction. Elle a permis de montrer la faisabilité d'attaques utilisant une double faute à l'aide de ce banc d'injection si l'intervalle entre les deux fautes est suffisamment important.

Le troisième chapitre étudie plus en détail les effets de la technique d'injection en vue d'établir un modèle de fautes au niveau assembleur. Pour cela, ce chapitre

commence par étudier l'influence des différents paramètres expérimentaux sur les fautes obtenues. Cette étude permet de recentrer l'analyse sur le cas particulier des transferts sur les bus d'instructions et de données du microcontrôleur. A partir de nouveaux résultats expérimentaux liés aux transferts sur ces bus, ce chapitre présente ensuite un modèle de fautes au niveau RTL qui explique les fautes réalisées à l'aide de la technique d'injection. Enfin, la dernière partie du chapitre étudie le cas de l'abstraction de ce modèle vers un modèle de plus haut niveau dans lequel un attaquant peut réaliser un saut d'instruction.

Le quatrième chapitre se base sur le modèle simplifié de saut d'instruction étudié précédemment pour proposer une contre-mesure permettant de renforcer un code embarqué. Cette contre-mesure utilise une approche de tolérance aux fautes, et exploite un principe de redondance locale à l'échelle de l'instruction afin de permettre une meilleure résistance face aux attaques basées sur des fautes multiples. Pour cela, elle propose une séquence de remplacement tolérante à une faute par saut d'instruction pour presque toutes les instructions du jeu Thumb-2. Un processus de vérification formelle de ces séquences de remplacement face à une faute par saut d'instruction ainsi que de leur équivalence sémantique par rapport aux instructions qu'elles remplacent a ensuite été mis en place et présenté. Celui-ci utilise l'outil de model-checking Vis. Enfin, une évaluation du surcoût apporté par cette contre-mesure en termes de performance et taille de code est réalisée pour différentes implémentations. Cette évaluation a permis de montrer que la contre-mesure avait un surcoût en termes de performance comparable à ceux des schémas de redondance classiques à l'échelle de l'algorithme.

Enfin, le cinquième chapitre s'intéresse à l'utilisation en pratique de cette contre-mesure et à son efficacité face à des attaques par injection électromagnétique. Comme la contre-mesure présentée ne propose aucune protection face aux fautes affectant les chargements de données, une contre-mesure au niveau assembleur proposée dans la littérature scientifique et basée sur un principe de détection est utilisée en complément. Ces deux contre-mesures sont d'abord évaluées sur des instructions isolées puis sur des codes complexes issus d'une implémentation de FreeRTOS-MPU. Les expérimentations ont tout d'abord mis en avant la nécessité de forcer un encodage sur 32 bits pour les instructions utilisées, ce qui tend par ailleurs à confirmer les résultats obtenus dans le troisième chapitre sur la définition du modèle. Les résultats expérimentaux montrent ensuite que la contre-mesure de tolérance proposée dans cette thèse semble efficace et contribue à renforcer un code embarqué face aux injections de fautes. Néanmoins, celle-ci s'avère globalement moins efficace que la contre-mesure de détection pour renforcer des codes composés de différents types d'instructions. Le fait qu'elle se soit montrée particulièrement efficace pour protéger des instructions d'appel de sous-fonctions et qu'elle puisse renforcer des instructions pour lesquelles la contre-mesure de détection n'est pas applicable permettent néanmoins de la voir comme un complément efficace à la

contre-mesure de détection. Enfin, une analyse basique de la résistance face aux attaques par observation n'a pas montré de vulnérabilité évidente pour cette contre-mesure.

6.2 Perspectives

Les travaux présentés dans cette thèse permettent d'ouvrir plusieurs pistes de recherche pour améliorer la sécurité des systèmes embarqués. Nous avons identifié plusieurs de ces pistes potentielles, celles-ci sont présentées dans les paragraphes qui suivent.

Amélioration de la précision des modèles de fautes La définition d'un modèle de fautes précis dans la première partie de cette thèse a été limitée par l'impossibilité d'accéder à des informations spécifiques sur le fonctionnement interne du processeur. Nous pensons donc qu'une perspective intéressante à la suite de ces travaux consisterait à analyser à l'aide d'outils de mesure plus performants les transferts sur les bus et l'impact d'une injection de fautes lors du passage d'instructions dans le pipeline. Une meilleure compréhension des fautes réalisées au niveau des étages du pipeline permettrait ainsi de réaliser des progrès considérables dans la définition de contre-mesures à la fois logicielles et matérielles. De tels travaux visant à améliorer la précision des modèles de fautes sont par exemple présentés dans (Balasch et al., 2011) ou (Hummel, 2014).

Vérification formelle de contre-mesures Pour les travaux de vérification formelle de contre-mesures présentés dans cette thèse, plusieurs autres approches auraient pu être utilisées. On peut notamment mentionner la modélisation formelle de l'architecture ARMv7-M proposée dans (Fox et Myreen, 2010), qui permet de générer des preuves pour vérifier la sémantique de portions de code. L'approche présentée dans (Pattabiraman et al., 2013) permet de tester la résistance de larges portions de codes face à des attaques par injection de fautes avec un modèle de fautes visant à la fois le flot de contrôle et le flot de données. Cette approche représente donc une perspective très intéressante pour vérifier formellement la résistance de codes embarqués renforcés à l'aide de contre-mesures. Enfin, un outil comme celui présenté dans (Rauzy et Guilley, 2013) pour modéliser des fautes au niveau du flot de données d'un algorithme pourrait également être adapté pour modéliser des fautes au niveau du flot de contrôle.

Intégration des contre-mesures dans des schémas de sécurité à base de code auto-modifiant La contre-mesure de tolérance au saut d'une instruction présentée dans cette thèse ainsi que la contre-mesure de détection de fautes présentée dans le cinquième chapitre présentent l'avantage de pouvoir s'ajouter aux techniques de polymorphisme et de réécriture de code utilisées par des schémas de sécurité à base de code auto-modifiant. Plusieurs exemples de tels schémas de sécurité ont

été présentés dans la littérature scientifique ces dernières années, notamment dans (AMARILLI et al., 2011) ou (AGOSTA et al., 2012). Des contre-mesures comme celles présentées dans cette thèse pourraient être automatiquement appliquées en cas de détection d'attaque par certains des capteurs installés sur le circuit. Contrairement aux approches consistant à détruire le circuit en cas de détection d'attaque, le fait de renforcer une partie du code à l'aide de contre-mesures si une attaque est détectée permet d'éviter une destruction du circuit en cas de faux positif.

Définition de stratégies de contre-mesures plus élaborées Des stratégies de protection plus élaborées pourraient être définies à l'aide des contre-mesures présentées dans cette thèse. Des protections au niveau bloc d'instructions, combinant détection et tolérance, ou encore utilisant des solutions pour contrôler l'intégrité du flot de contrôle d'un programme peuvent par exemple être envisagées. Par exemple, des contre-mesures face aux attaques en fautes pourraient se combiner avec le principe de logique double-rail à précharge au niveau du logiciel présenté en (RAUZY, GUILLEY et NAJM, 2014). Le cas des combinaisons entre contre-mesures matérielles et logicielles est également une piste possible, sur le modèle par exemple des travaux présentés dans (M. H. NGUYEN et al., 2011).

Application automatique des contre-mesures par les compilateurs Une des principales perspectives des contre-mesures au niveau assembleur présentées dans cette thèse consiste à les intégrer dans le flot de conception de logiciels embarqués, en ajoutant la possibilité pour les compilateurs de générer du code ainsi renforcé pour certaines fonctions choisies par le programmeur. Un renforcement du code assembleur pourrait ainsi être fait avec un surcoût en temps de développement minimal pour des codes complexes. De telles applications automatisées pour des contre-mesures face aux attaques par observation ont par exemple été proposées récemment dans (MOSS et al., 2012).

Étude de l'impact des contre-mesures face aux attaques par observation La question de la résistance des contre-mesures proposées face aux attaques par observation doit également plus largement être étudiée, notamment afin de déterminer si elles ont pour effet secondaire d'introduire des vulnérabilités face à ces attaques. Dans tous les cas, ces contre-mesures pourraient être appliquées à la grande majorité des implémentations non-cryptographiques.

A

Montage expérimental pour processeur ATmega128

Les premières expérimentations à la base de ces travaux de thèse ont été réalisées sur un microcontrôleur Atmel ATmega128. Les microcontrôleurs 8 bits de la famille ATmega sont basés sur une architecture RISC de type Harvard modifiée. Le modèle ATmega128 est réalisé en technologie $0.35\ \mu m$ et possède 128Kio de mémoire Flash, 4Kio de mémoire EEPROM et 4Kio de mémoire de données SRAM. Les registres `X`, `Y` et `Z` sont des registres 16 bits, chacun d'entre eux étant en réalité constitué par deux registres 8 bits : un registre pour les bits de poids fort et un pour ceux de poids faible. Ils sont utilisés comme pointeurs pour le mode d'adressage indirect, permettant ainsi de manipuler des adresses mémoire sur 16 bits.

Le circuit ciblé exécute un chiffrement AES, et nous avons cherché à attaquer la fonction `AddRoundKey` de la dernière ronde, qui réalise le OU exclusif entre la dernière clé de ronde et la matrice d'état. Cette fonction est la dernière avant la fin du chiffrement. L'ATmega128 est un microcontrôleur 8 bits : la fonction `AddRoundKey` réalise donc l'opération de XOR octet par octet.

Listing A.1 : Calcul de XOR sur ATmega128

```
1 ldd   r24 , Y+i  ; chargement de l'octet de clé
2 ld    r25 , X    ; chargement de texte
3 eor   r24 , r25  ; ou exclusif entre les deux octets
4 std   Z+i , r24  ; stockage du résultat en mémoire
```

Pour différents textes clairs en entrée, et en faisant varier les paramètres de l'impulsion électromagnétique injectée, deux types de fautes sur la valeur de sortie de l'octet ciblé ont été obtenues :

— des fautes constantes indépendantes du texte clair

— des fautes dépendant du texte clair

Pour les fautes dépendant du texte clair, la valeur de sortie de l'octet correspondait à la valeur du dernier octet de la dernière clé de ronde.

Ce type d'effet peut s'expliquer par deux hypothèses :

— des fautes au niveau des données

— des fautes au niveau des instructions

Fautes au niveau des données Fautes admissibles :

— `r25` a été fauté à `0x00`, provoquant donc un `eor r24, 0x00`

— `r26` ou `r27` (qui définissent le pointeur `X`) ont été fautés, provoquant donc le chargement d'une mauvaise valeur dans `r25`

Fautes au niveau des instructions Des effets au niveau des instructions ont notamment été obtenus par perturbations transitoires du signal d'horloge sur le même type de microcontrôleur dans (BALASCH et al., 2011). En partant de leur hypothèse selon laquelle une instruction peut être remplacée par un *nop*, un certain nombre de sauts d'instructions peuvent expliquer la présence d'un octet de la clé sur la valeur de sortie. Nous avons utilisé le simulateur SimulAVR [1] pour vérifier les effets d'un saut d'instruction sur la sortie finale.

Fautes admissibles :

— l'instruction `eor` a été remplacée par un `nop`, entraînant donc le stockage du registre `r24` dans la mémoire

— l'instruction `ld` qui charge l'adresse pointée par `X` dans `r25` a été remplacée par un `nop`, entraînant donc l'exécution d'un `eor` entre `r24` et la valeur précédente de `r25`

Il est toutefois impossible que l'opération de stockage de `r24` à l'adresse pointée par `Z+i` ait été perturbée : l'adresse `Z+i` pointe sur une case mémoire qui contient `0x00` avant l'exécution de l'instruction.

Distinction entre les deux types d'effet Au vu des outils de mesure dont nous disposons sur ce circuit (qui ne nous permettent pas d'avoir des informations sur son état interne), il nous est impossible de faire la distinction entre une faute entraînant le remplacement d'une instruction par un *nop* et une faute sur la donnée manipulée par cette instruction. Il en va très certainement de même pour un certain nombre de modèles de fautes sur microcontrôleur proposés dans la littérature. Les fautes sur un octet des données peuvent en réalité très bien avoir été la conséquence d'une perturbation d'instruction. Pour arriver à faire cette distinction, il nous aurait fallu pouvoir avoir accès à davantage d'informations sur l'état interne du circuit à la fin de l'opération.

1. http://www.nongnu.org/simulavr

Références bibliographiques

ABOULKASSIMI, Driss, AGOYAN, Michel, FREUND, Laurent, FOURNIER, Jacques, ROBISSON, Bruno et TRIA, Assia (2011). « ElectroMagnetic analysis (EMA) of software AES on Java mobile phones ». In : *2011 IEEE International Workshop on Information Forensics and Security*. IEEE, p. 1–6. DOI : 10.1109/WIFS.2011.6123131 (cf. p. 131).

AGOSTA, Giovanni, BARENGHI, Alessandro et PELOSI, Gerardo (2012). « A code morphing methodology to automate power analysis countermeasures ». In : *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*. New York, New York, USA : ACM Press, p. 77. DOI : 10.1145/2228360.2228376 (cf. p. 17, 138).

AGOYAN, Michel, DUTERTRE, Jean-max, NACCACHE, David, ROBISSON, Bruno et TRIA, Assia (2010). « When Clocks Fail : On Critical Paths and Clock Faults ». In : *CARDIS 2010*. Sous la dir. de Dieter GOLLMANN, Jean-Louis LANET et Julien IGUCHI-CARTIGNY. T. 6035. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg. DOI : 10.1007/978-3-642-12510-2 (cf. p. 20).

AGRAWAL, Dakshi, ARCHAMBEAULT, Bruce, RAO, Josyula R, ROHATGI, Pankaj et HEIGHTS, Yorktown (2003). « The EM Side-Channel(s) ». In : *Cryptographic Hardware and Embedded Systems - CHES 2002*. Sous la dir. de Burton S. KALISKI, çetin K. KOÇ et Christof PAAR. T. 2523. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 29–45. DOI : 10.1007/3-540-36400-5 (cf. p. 12).

AMARILLI, Antoine, MÜLLER, Sascha, NACCACHE, David, PAGE, Daniel, RAUZY, Pablo et MICHAEL TUNSTALL (2011). « Can Code Polymorphism Limit Information Leakage ? » In : *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication* (cf. p. 17, 138).

ANDERSON, Ross J et KUHN, Markus (1996). « Tamper resistance - a cautionary note ». In : *Proceedings of the second Usenix workshop on electronic commerce*, p. 1–11 (cf. p. 4).

ARM (2006). *AMBA 3 AHB-Lite Protocol* (cf. p. 36, 64, 65).

AUMÜLLER, C., BIER, P., FISCHER, W., HOFREITER, P. et SEIFERT, JP (2003). « Fault attacks on RSA with CRT : Concrete results and practical countermeasures ». In : *Cryptographic hardware and embedded systems - CHES 2002*. Sous la dir. de Burton S. KALISKI, çetin K. KOÇ et Christof PAAR. T. 2523. Lecture Notes in Computer

Science. Springer Berlin Heidelberg. DOI : 10.1007/3-540-36400-5_20 (cf. p. 25).

BAARIR, Souheib, BRAUNSTEIN, Cécile, CLAVEL, Renaud, ENCRENAZ, Emmanuelle, ILIÉ, Jean-Michel, LEVEUGLE, Régis, MOUNIER, Isabelle, PIERRE, Laurence et POITRENAUD, Denis (2009). « Complementary Formal Approaches for Dependability Analysis ». In : *2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, p. 331–339. DOI : 10.1109/DFT.2009.21 (cf. p. 91).

BAARIR, Souheib, BRAUNSTEIN, Cécile, ENCRENAZ, Emmanuelle, ILIÉ, Jean-Michel, MOUNIER, Isabelle, POITRENAUD, Denis et YOUNES, Sana (2011). « Feasibility analysis for robustness quantification by symbolic model checking ». In : *Formal Methods in System Design* 39.2, p. 165–184. DOI : 10.1007/s10703-011-0121-5 (cf. p. 91).

BAIER, Christel et KATOEN, Joost-Pieter (2008). *Principles Of Model Checking*. T. 950. DOI : 10.1093/comjnl/bxp025 (cf. p. 89).

BALASCH, Josep, GIERLICHS, Benedikt et VERBAUWHEDE, Ingrid (2011). « An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs ». In : *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 105–114. DOI : 10.1109/FDTC.2011.9 (cf. p. 22, 71, 137, 140).

BARBU, Guillaume, DUC, Guillaume et HOOGVORST, Philippe (2011). « Java card operand stack : fault attacks, combined attacks and countermeasures ». In : *CARDIS 2011*, p. 297–313 (cf. p. 22).

BAR-EL, H., CHOUKRI, H., NACCACHE, D., TUNSTALL, M. et WHELAN, C. (2006). « The Sorcerer's Apprentice Guide to Fault Attacks ». In : *Proceedings of the IEEE* 94.2, p. 370–382. DOI : 10.1109/JPROC.2005.862424 (cf. p. 25, 48).

BARENGHI, Alessandro, BREVEGLIERI, Luca, KOREN, Israel et NACCACHE, David (2012). « Fault Injection Attacks on Cryptographic Devices : Theory, Practice, and Countermeasures ». In : *Proceedings of the IEEE* 100.11, p. 3056–3076. DOI : 10.1109/JPROC.2012.2188769 (cf. p. 18, 21, 22, 26).

BARENGHI, Alessandro, BREVEGLIERI, Luca, KOREN, Israel, PELOSI, Gerardo et REGAZZONI, Francesco (2010). « Countermeasures against fault attacks on software implemented AES ». In : *Proceedings of the 5th Workshop on Embedded Systems Security - WESS '10*. New York, New York, USA : ACM Press, p. 1–10. DOI : 10.1145/1873548.1873555 (cf. p. 25, 110).

BAYRAK, Ali Galip, REGAZZONI, Francesco, BRISK, Philip, STANDAERT, François-xavier et IENNE, Paolo (2011). « A First Step Towards Automatic Application of Power Analysis Countermeasures Categories and Subject Descriptors ». In : p. 230–235 (cf. p. 16).

BERNSTEIN, Daniel J (2005). « Cache-timing attacks on AES ». In : (cf. p. 7).

BERTHOME, Pascal, HEYDEMANN, Karine, KAUFFMANN-TOURKESTANSKY, Xavier et LALANDE, Jean-Francois (2012). « High Level Model of Control Flow Attacks for Smart Card Functional Security ». In : *2012 Seventh International Conference on*

Availability, Reliability and Security. IEEE, p. 224–229. DOI : 10.1109/ARES.2012.79 (cf. p. 22).

BERTONI, G., ZACCARIA, V., BREVEGLIERI, L., MONCHIERO, M. et PALERMO, G. (2005). « AES power attack based on induced cache miss and countermeasure ». In : *International Conference on Information Technology : Coding and Computing (ITCC'05) - Volume II*, 586–591 Vol. 1. DOI : 10.1109/ITCC.2005.62 (cf. p. 7).

BIHAM, Eli et SHAMIR, Adi (1997). « Differential Fault Analysis of Secret Key Cryptosystems ». In : *Proceedings of the 17th Annual International Cryptology Conference*. September 1996. Santa Barbara, California, USA. DOI : 10.1007/BFb0052259 (cf. p. 23).

BLÖMER, J, SILVA, RG da, GÜNTHER, P, KRÄMER, J et SEIFERT, JP (2014). *A Practical Second-Order Fault Attack against a Real-World Pairing Implementation*. Rapp. tech. (cf. p. 23).

BONEH, Dan, DEMILLO, Richard A et LIPTON, Richard J (1997). « On the Importance of Checking Cryptographic Protocols for Faults ». In : *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*. EUROCRYPT'97 1233, p. 37–51 (cf. p. 17, 23, 90).

BOUFFARD, Guillaume, IGUCHI-CARTIGNY, J et LANET, JL (2011). « Combined software and hardware attacks on the java card control flow ». In : *CARDIS 2011*, p. 283–296 (cf. p. 22, 26).

BRIER, Eric, CLAVIER, Christophe et OLIVIER, Francis (2004). « Correlation power analysis with a leakage model ». In : *Cryptographic Hardware and Embedded Systems* 3156, p. 16–29. DOI : 10.1007/978-3-540-28632-5_2 (cf. p. 13, 131).

BROUCHIER, Julien, KEAN, Tom, MARSH, Carol et NACCACHE, David (2009). « Temperature Attacks ». In : *IEEE Security & Privacy Magazine* 7.2, p. 79–82. DOI : 10.1109/MSP.2009.54 (cf. p. 8).

CANIVET, G., MAISTRI, P., LEVEUGLE, R., CLÉDIÈRE, J., VALETTE, F. et RENAUDIN, M. (2010). « Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA ». In : *Journal of Cryptology* 24.2, p. 247–268. DOI : 10.1007/s00145-010-9083-9 (cf. p. 19).

CARPI, Rafael Boix, PICEK, Stjepan, BATINA, Lejla, MENARINI, Federico, JAKOBOVIC, Domagoj et GOLUB, Marin (2013). « Glitch It If You Can : Parameter Search Strategies for Successful Fault Injection ». In : *CARDIS 2013* (cf. p. 20).

CHARI, Suresh, RAO, J et ROHATGI, Pankaj (2003). « Template attacks ». In : *Cryptographic Hardware and Embedded Systems - CHES 2002*. Sous la dir. de Burton S. KALISKI, çetin K. KOÇ et Christof PAAR. T. 2523. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 13–28. DOI : 10.1007/3-540-36400-5_3 (cf. p. 13).

CHETALI, Boutheina et NGUYEN, Quang-huy (2008). « Industrial Use of Formal Methods for a High-Level Security Evaluation ». In : *FM 2008 : Formal Methods*, p. 198–213. DOI : 10.1007/978-3-540-68237-0_15 (cf. p. 89).

CHOUKRI, Hamid et TUNSTALL, Michael (2005). « Round Reduction Using Faults ». In : *Fault Diagnosis and Tolerance in Cryptography 2005*, p. 13–24 (cf. p. 45).

CHRISTOFI, Maria, CHETALI, Boutheina, GOUBIN, Louis et VIGILANT, David (2013). « Formal verification of a CRT-RSA implementation against fault attacks ». In : *Journal of Cryptographic Engineering*. DOI : 10.1007/s13389-013-0049-3 (cf. p. 90).

CLAVIER, Christophe (2004). *Side Channel Analysis for Reverse Engineering (SCARE)– An Improved Attack Against a Secret A3/A8 GSM Algorithm*. Rapp. tech. (cf. p. 14).

CLAVIER, Christophe (2007a). « De la sécurité physique des crypto-systèmes embarqués ». Thèse de doct. UVSQ (cf. p. 11).

CLAVIER, Christophe (2007b). « Secret External Encodings Do Not Prevent Transient Fault Analysis ». In : *Clavier*, p. 181–194. DOI : 10.1007/978-3-540-74735-2_13 (cf. p. 23).

CLAVIER, Christophe et WURCKER, Antoine (2013). « Reverse Engineering of a Secret AES-like Cipher by Ineffective Fault Analysis ». In : *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, p. 119–128. DOI : 10.1109/FDTC.2013.16 (cf. p. 24).

COPPENS, Bart, VERBAUWHEDE, Ingrid, BOSSCHERE, Koen De et SUTTER, Bjorn De (2009). « Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors ». In : *2009 30th IEEE Symposium on Security and Privacy*, p. 45–60. DOI : 10.1109/SP.2009.19 (cf. p. 16).

CORON, Jean-Sébastien, GIRA, Christophe, MORIN, Nicolas, PIRET, Gilles et VIGILANT, David (2010). « Fault Attacks and Countermeasures on Vigilant's RSA-CRT Algorithm ». In : *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 89–96. DOI : 10.1109/FDTC.2010.9 (cf. p. 25, 90).

CYRIL ROSCIAN (2013). « Cryptanalyse physique de circuits cryptographiques à l'aide de sources LASER ». Thèse de doct. (cf. p. 19).

DANGER, Jean-Luc, GUILLEY, Sylvain, BHASIN, Shivam et NASSAR, Maxime (2009). « Overview of Dual rail with Precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors ». In : *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*. IEEE, p. 1–8. DOI : 10.1109/ICSCS.2009.5412599 (cf. p. 16).

DEBUSSCHERE, E et MCCAMBRIDGE, M (2012). *Modern Game Console Exploitation* (cf. p. xiii).

DEHBAOUI, Amine (2011). « Analyse Sécuritaire des Emanations Electromagnétiques des Circuits Intégrés ». Thèse de doct. Université Montpellier 2 (cf. p. 21).

DEHBAOUI, Amine, DUTERTRE, Jean-Max, ROBISSON, Bruno, ORSATELLI, Philippe, MAURINE, Philippe et TRIA, Assia (2012). « Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system ». In : *Cryptology ePrint Archive* (cf. p. 20).

DEHBAOUI, Amine, DUTERTRE, Jean-Max, ROBISSON, Bruno et TRIA, Assia (2012). « Electromagnetic Transient Faults Injection on a Hardware and a Software Im-

plementations of AES ». In : *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 7–15. DOI : 10.1109/FDTC.2012.15 (cf. p. 20, 29, 39, 43, 52).

DEHBAOUI, Amine, ORDAS, Thomas, LOMNE, Victor, MAURINE, Philippe, TORRES, Lionel et ROBERT, Michel (2010). « Incoherence analysis and its application to time domain EM analysis of secure circuits ». In : *2010 Asia-Pacific International Symposium on Electromagnetic Compatibility*, p. 1039–1042. DOI : 10.1109/APEMC.2010.5475481 (cf. p. 6, 7).

DHEM, Jean-Francois, KOEUNE, Francois, LEROUX, Philippe-Alexandre, MESTRÉ, Patrick, QUISQUATER, Jean-Jacques et WILLEMS, Jean-Louis (1998). « A practical implementation of the timing attack ». In : *Smart Card Research and Advanced Applications*, p. 167–182 (cf. p. 11).

DIFFIE, W. et HELLMAN, M. (1976). « New directions in cryptography ». In : *IEEE Transactions on Information Theory* 22.6, p. 644–654. DOI : 10.1109/TIT.1976.1055638 (cf. p. 2).

DUTERTRE, Jean-Max, FOURNIER, Jacques J.a., MIRBAHA, Amir-Pasha, NACCACHE, David, RIGAUD, Jean-Baptiste, ROBISSON, Bruno et TRIA, Assia (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)*, p. 1–6. DOI : 10.1109/DTIS.2011.5941421 (cf. p. 18, 24).

DUTERTRE, Jean-Max, MIRBAHA, Amir-Pasha, NACCACHE, David, RIBOTTA, Anne-Lise, TRIA, Assia et VASCHALDE, Thierry (2012). « Fault Round Modification Analysis of the advanced encryption standard ». In : *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, p. 140–145. DOI : 10.1109/HST.2012.6224334 (cf. p. 23, 45).

EISENBARTH, Thomas, PAAR, Christof et WEGHENKEL, Björn (2010). « Building a side channel based disassembler ». In : *Transactions on computational science*, p. 1–21 (cf. p. 15).

EUROSMART (2012). *World Card Summit* (cf. p. xiii).

FEY, Goerschwin et DRECHSLER, Rolf (2008). « A Basis for Formal Robustness Checking ». In : *9th International Symposium on Quality Electronic Design (isqed 2008)*. IEEE, p. 784–789. DOI : 10.1109/ISQED.2008.4479838 (cf. p. 91).

FEY, Görschwin, SULFLOW, André, FREHSE, Stefan et DRECHSLER, Rolf (2011). « Effective Robustness Analysis Using Bounded Model Checking Techniques ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8, p. 1239–1252. DOI : 10.1109/TCAD.2011.2120950 (cf. p. 91).

FOUQUE, Pierre-Alain, LERESTEUX, Delphine et VALETTE, Frédéric (2012). « Using faults for buffer overflow effects ». In : *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*. New York, New York, USA : ACM Press, p. 1638. DOI : 10.1145/2245276.2232038 (cf. p. 23).

Fox, Anthony et Myreen, M. (2010). « A trustworthy monadic formalization of the ARMv7 instruction set architecture ». In : *Interactive Theorem Proving*, p. 243–258. doi : 10.1007/978-3-642-14052-5_18 (cf. p. 92, 137).

Fumaroli, Guillaume, Martinelli, Ange, Prouff, Emmanuel et Rivain, Matthieu (2010). « Affine masking against higher-order side channel analysis ». In : *Selected Areas in Cryptography - SAC 2010*. T. 6544 LNCS. doi : 10.1007/978-3-642-19574-7_18 (cf. p. 17).

Gandolfi, Karine, Mourtel, Christophe et Olivier, Francis (2001). « Electromagnetic Analysis : Concrete Results ». In : *Cryptographic Hardware and Embedded Systems - CHES 2001*, p. 251–261 (cf. p. 6).

Genkin, Daniel, Shamir, Adi et Tromer, Eran (2013). « RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis ». In : *IACR Cryptology ePrint Archive*, p. 1–57 (cf. p. 8).

Gierlichs, Benedikt, Batina, Lejla et Tuyls, Pim (2008). « Mutual information analysis ». In : *Cryptographic Hardware and Embedded Systems - CHES 2008*, p. 1–16 (cf. p. 10, 12, 13).

Giraud, Christophe (2005). « DFA on AES ». In : *4th International Conference, AES 2004*. Bonn, Germany, p. 27–41. doi : 10.1007/11506447_4 (cf. p. 23).

Giraud, Christophe et Thiebeauld, Hugues (2004). « A survey on fault attacks ». In : *CARDIS 2004* (cf. p. 17).

Goldack, Martin (2008). « Side-channel based reverse engineering for microcontrollers ». Thèse de doct. Ruhr-University Bochum (cf. p. 15).

Guilley, Sylvain, Hoogvorst, Philippe et Pacalet, Renaud (2004). « Differential power analysis model and some results ». In : *CARDIS 2004* (cf. p. 5).

Guilley, Sylvain, Sauvage, Laurent, Danger, Jean-Luc, Graba, Tarik et Mathieu, Yves (2008). « Evaluation of Power-Constant Dual-Rail Logic as a Protection of Cryptographic Applications in FPGAs ». In : *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, p. 16–23. doi : 10.1109/SSIRI.2008.31 (cf. p. 16).

Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. et Brown, R.B. (2001). « MiBench : A free, commercially representative embedded benchmark suite ». In : *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4*. IEEE, p. 3–14. doi : 10.1109/WWC.2001.990739 (cf. p. 106).

Hummel, T (2014). « Exploring Effects of Electromagnetic Fault Injection on a 32-bit High Speed Embedded Device Microprocessor ». Master Thesis. University of Twente (cf. p. 22, 71, 137).

Joye, Marc (2000). « Checking before output may not be enough against fault-based cryptanalysis ». In : *IEEE Transactions on Computers* 49.9, p. 967–970. doi : 10.1109/12.869328 (cf. p. 23).

JOYE, Marc (2012). « A Method for Preventing "Skipping" Attacks ». In : *2012 IEEE Symposium on Security and Privacy Workshops*. IEEE, p. 12–15. DOI : 10.1109/SPW.2012.14 (cf. p. 25).

JOYE, Marc, PAILLIER, Pascal et SCHOENMAKERS, Berry (2005). « On second-order differential power analysis ». In : *Cryptographic Hardware and Embedded Systems - CHES 2005*, p. 293–308 (cf. p. 17).

KOCHER, Paul (1996). « Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems ». In : *Advances in Cryptology - CRYPTO'96*, p. 104–113 (cf. p. 3, 6, 11).

KOCHER, Paul, JAFFE, J et JUN, Benjamin (1999). « Differential power analysis ». In : *Proceedings of the 19th Annual International Cryptology Conference*. Santa Barbara, California, USA, p. 1–10. DOI : 10.1007/3-540-48405-1_25 (cf. p. 3, 10, 11, 13, 14).

KÖMMERLING, Oliver et KUHN, Markus G (1999). « Design Principles for Tamper-Resistant Smartcard Processors ». In : *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, p. 9–20 (cf. p. 4).

LASHERMES, Ronan, REYMOND, Guillaume, DUTERTRE, Jean-Max, FOURNIER, Jacques, ROBISSON, Bruno et TRIA, Assia (2012). « A DFA on AES Based on the Entropy of Error Distributions ». In : *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, p. 34–43. DOI : 10.1109/FDTC.2012.18 (cf. p. 23).

LE BOUDER, Hélène, GUILLEY, Sylvain, ROBISSON, Bruno et TRIA, Assia (2013). « Fault Injection to Reverse Engineer DES-like Cryptosystems ». In : *Sixth International Symposium on Foundations & Practice of Security FPS'2013* (cf. p. 24).

LENSTRA, Arjen K et VERHEUL, Eric R (2001). « Selecting Cryptographic Key Sizes ». In : *Journal of Cryptology* 14.4, p. 255–293. DOI : 10.1007/s00145-001-0009-4 (cf. p. 44).

LI, Yang, SAKIYAMA, Kazuo et GOMISAWA, Shigeto (2010). « Fault sensitivity analysis ». In : *Cryptographic Hardware and Embedded Systems - CHES 2010*, p. 320–334 (cf. p. 9).

MARKETTOS, A Theodore (2011). « Active electromagnetic attacks on secure hardware ». Thèse de doct. University of Cambridge, p. 217 (cf. p. 20).

MEOLA, Matthew L et WALKER, David (2010). « Faulty Logic : Reasoning about Fault Tolerant Programs ». In : *19th European Symposium on Programming, ESOP 2010*. T. 6012. Lecture Notes in Computer Science. Springer Berlin Heidelberg, p. 468–487. DOI : 10.1007/978-3-642-11957-6_25 (cf. p. 89).

MESSERGES, Thomas S (2001). « Securing the AES Finalists Against Power Analysis Attacks ». In : *Fast Software Encryption*, p. 150–164 (cf. p. 17).

MOLNAR, David, PIOTROWSKI, Matt, SCHULTZ, David et WAGNER, David (2006). « The Program Counter Security Model : Automatic Detection and Removal of Control-Flow Side Channel Attacks ». In : *ICISC'05 Proceedings of the 8th international conference on Information Security and Cryptology*. Springer, p. 156–168. DOI : 10.1007/11734727_14 (cf. p. 16).

MORADI, Amir, SHALMANI, Mohammad T Manzuri et SALMASIZADEH, Mahmoud (2006). « A Generalized Method of Differential Fault Attack Against AES Cryptosystem ». In : *Cryptographic Hardware and Embedded Systems - CHES 2006*, p. 91–100 (cf. p. 23).

MOSS, Andrew, OSWALD, Elisabeth, PAGE, Daniel et TUNSTALL, Michael (2012). « Compiler Assisted Masking ». In : *Cryptographic Hardware and Embedded Systems - CHES 2012*. DOI : 10.1007/978-3-642-33027-8_4 (cf. p. 138).

MOUNIER, Benjamin, RIBOTTA, Anne-Lise et FOURNIER, Jacques (2012). « EM Probes Characterisation for Security Analysis ». In : *Cryptography and Security : From Theory to Applications*, p. 248–264 (cf. p. 6).

NACCACHE, David (2005). « Finding Faults ». In : *IEEE Security and Privacy Magazine* 3.5, p. 61–65. DOI : 10.1109/MSP.2005.122 (cf. p. 22).

NGUYEN, Minh Huu, ROBISSON, Bruno, AGOYAN, Michel et DRACH, Nathalie (2011). « Low-cost recovery for the code integrity protection in secure embedded processors ». In : *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, p. 99–104. DOI : 10.1109/HST.2011.5955004 (cf. p. 138).

NIST (1977). *Data Encryption Standard (DES) - FIPS 46* (cf. p. 44).

NIST (2001). *Advanced Encryption Standard (AES) - FIPS 197* (cf. p. 44, 106).

NOVAK, Roman (2003). « Side-channel attack on substitution blocks ». In : *Applied Cryptography and Network Security*, p. 307–318 (cf. p. 14).

PATTABIRAMAN, Karthik, NAKKA, Nithin M., KALBARCZYK, Zbigniew T. et IYER, Ravishankar K. (2013). « SymPLFIED : Symbolic Program-Level Fault Injection and Error Detection Framework ». In : *IEEE Transactions on Computers* 62.11, p. 2292–2307. DOI : 10.1109/TC.2012.219 (cf. p. 90, 137).

PEETERS, Eric, STANDAERT, François-Xavier et QUISQUATER, Jean-jacques (2007). « Power and electromagnetic analysis : Improved model, consequences and comparisons ». In : *Integration, the VLSI Journal* 40.1, p. 52–60. DOI : 10.1016/j.vlsi.2005.12.013 (cf. p. 6, 9).

PIRET, Gilles et QUISQUATER, Jean-jacques (2003). « A Differential Fault Attack Technique against SPN Structures , with Application to the AES and KHAZAD ». In : *Cryptographic Hardware and Embedded Systems CHES 2003*. Lecture Notes in Computer Science 2779.CHES'2003. Sous la dir. de C D WALTER, Ç K KOÇ et C PAAR, p. 77–88 (cf. p. 23).

POUCHERET, F, CHUSSEAU, L, ROBISSON, B et MAURINE, P (2011). « Local electromagnetic coupling with CMOS integrated circuits ». In : *2011 8th Workshop on Electromagnetic Compatibility of Integrated Circuits* (cf. p. 20).

QUISQUATER, Jean-jacques et SAMYDE, David (2001). « ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards ». In : *Smart Card Programming and Security*. Springer Berlin Heidelberg, p. 200–210 (cf. p. 6, 12, 20).

RAUZY, Pablo et GUILLEY, Sylvain (2013). « A formal proof of countermeasures against fault injection attacks on CRT-RSA ». In : *Journal of Cryptographic Engineering*. DOI : 10.1007/s13389-013-0065-3 (cf. p. 90, 137).

RAUZY, Pablo et GUILLEY, Sylvain (2014). « Formal Analysis of CRT-RSA Vigilant's Countermeasure Against the BellCoRe Attack ». In : *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW'14*. New York, New York, USA : ACM Press, p. 1–10. DOI : 10.1145/2556464.2556466 (cf. p. 90).

RAUZY, Pablo, GUILLEY, Sylvain et NAJM, Zakaria (2014). « Formally Proved Security of Assembly Code Against Power Analysis ». In : *PROOFS Workshop* (cf. p. 16, 138).

RIVEST, R. L., SHAMIR, A. et ADLEMAN, L. (1978). « A method for obtaining digital signatures and public-key cryptosystems ». In : *Communications of the ACM* 21.2, p. 120–126. DOI : 10.1145/359340.359342 (cf. p. 2).

ROBISSON, Bruno et MANET, Pascal (2007). « Differential Behavioral Analysis ». In : *Cryptographic Hardware and Embedded Systems - CHES 2007*, p. 413–426 (cf. p. 23).

ROSCIAN, Cyril, SARAFIANOS, Alexandre, DUTERTRE, Jean-Max et TRIA, Assia (2013). « Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells ». In : *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 89–98. DOI : 10.1109/FDTC.2013.17 (cf. p. 18, 19).

SAN PEDRO, Manuel, SOOS, Mate et GUILLEY, Sylvain (2011). « FIRE : fault injection for reverse engineering ». In : *5th IFIP WG 11.2 International Workshop, WISTP 2011*, p. 280–293 (cf. p. 24).

SARAFIANOS, Alexandre, LISART, Mathieu, GAGLIANO, Olivier, SERRADEIL, Valerie, ROSCIAN, Cyril, DUTERTRE, Jean-Max et TRIA, Assia (2013). « Robustness improvement of an SRAM cell against laser-induced fault injection ». In : *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. T. 6. IEEE, p. 149–154. DOI : 10.1109/DFT.2013.6653598 (cf. p. 24).

SCHLÖSSER, Alexander, NEDOSPASOV, Dmitry, KRÄMER, Juliane, ORLIC, Susanna et SEIFERT, Jean-Pierre (2012). « Simple Photonic Emission Analysis of AES Photonic Side Channel Analysis for the Rest of Us ». In : *Cryptographic Hardware and Embedded Systems - CHES 2012*, p. 41–57 (cf. p. 8).

SCHMIDT, Jörn-Marc et HERBST, Christoph (2008). « A Practical Fault Attack on Square and Multiply ». In : *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. Sous la dir. de L BREVEGLIERI, S GUERON, I KOREN, D NACCACHE et J P SEIFERT. IEEE, p. 53–58. DOI : 10.1109/FDTC.2008.10 (cf. p. 23).

SCHMIDT, Jörn-Marc et HUTTER, Michael (2007). « Optical and EM Fault-Attacks on CRT-based RSA : Concrete Results ». In : *Proceedings of the 15th Austrian Workhop on Microelectronics - Austrochip 2007*. Graz, Austria (cf. p. 18, 20).

SCHMIDT, Jörn-Marc et HUTTER, Michael (2013). « The Temperature Side Channel and Heating Fault Attacks ». In : *CARDIS 2013* (cf. p. 8, 9, 19).

SELMANE, Nidhal, BHASIN, Shivam, GUILLEY, Sylvain, GRABA, Tarik et DANGER, Jean-Luc (2009). « WDDL is Protected against Setup Time Violation Attacks ». In : *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. Umr 5141. IEEE, p. 73–83. DOI : 10.1109/FDTC.2009.40 (cf. p. 25).

SERE, Ahmadou A., IGUCHI-CARTIGNY, Julien et LANET, Jean-louis (2009). « Automatic detection of fault attack and countermeasures ». In : *Proceedings of the 4th Workshop on Embedded Systems Security - WESS '09*. New York, New York, USA : ACM Press, p. 1–7. DOI : 10.1145/1631716.1631723 (cf. p. 22).

SERE, Ahmadou Al Khary, IGUCHI-CARTIGNY, Julien et LANET, Jean-Louis (2011). « Evaluation of Countermeasures Against Fault Attacks on Smart Cards ». In : *International Journal of Security and Its Applications* 5.2, p. 49–60 (cf. p. 26).

SHAMIR, Adi (1999). *Method and apparatus for protecting public key schemes from timing and fault attacks* (cf. p. 90).

SHANNON, Claude (1949). « Communication Theory of Secrecy Systems ». In : *Bell System Technical Journal* 28.4, p. 656–715. DOI : 10.1002/j.1538-7305.1949.tb00928.x (cf. p. 2).

SKOROBOGATOV, Sergei (2009). « Local heating attacks on Flash memory devices ». In : *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE, p. 1–6. DOI : 10.1109/HST.2009.5225028 (cf. p. 19).

SKOROBOGATOV, Sergei P et ANDERSON, Ross J (2003). « Optical Fault Induction Attacks ». In : *CHES 2003*. Sous la dir. de B S KALISKI JR., Ç K KOÇ et C PAAR. Lecture Notes in Computer Science. Springer. Chap. 2 (cf. p. 18).

SPRUYT, Albert (2012). *Building fault models for microcontrollers*. Rapp. tech. Amsterdam : University of Amsterdam (cf. p. 61).

STERN, Jacques (1998). *La science du secret*. Odile Jacob (cf. p. 2).

TIRI, Kris et VERBAUWHEDE, Ingrid (2004). « A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation ». In : *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. T. 00. DDL. IEEE Comput. Soc, p. 246–251. DOI : 10.1109/DATE.2004.1268856 (cf. p. 16).

TRICHINA, Elena et KORKIKYAN, Roman (2010). « Multi Fault Laser Attacks on Protected CRT-RSA ». In : *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 75–86. DOI : 10.1109/FDTC.2010.14 (cf. p. 19, 22).

VERBAUWHEDE, Ingrid, KARAKLAJIC, Dusko et SCHMIDT, Jorn-Marc (2011). « The Fault Attack Jungle - A Classification Model to Guide You ». In : *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 3–8. DOI : 10.1109/FDTC.2011.13 (cf. p. 21).

VIGILANT, David (2008). « RSA with CRT : A new cost-effective solution to thwart fault attacks ». In : *Cryptographic Hardware and Embedded Systems – CHES 2008*. T. 5154 LNCS. Springer Berlin Heidelberg, p. 130–145. DOI : 10.1007/978-3-540-85053-3_9 (cf. p. 25, 90).

WITTEMAN, Marc (2008). *Secure Application Programming in the Presence of Side Channel Attacks*. Rapp. tech. (cf. p. 25).

WOUDENBERG, Jasper G.J. van, WITTEMAN, Marc F. et MENARINI, Federico (2011). « Practical Optical Fault Injection on Secure Microcontrollers ». In : *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, p. 91–99. DOI : 10.1109/FDTC.2011.12 (cf. p. 23).

YIU, Joseph (2009). *The Definitive Guide To The ARM Cortex-M3*, p. 479 (cf. p. 30).

ZUSSA, Loic, DEHBAOUI, Amine, TOBICH, Karim, DUTERTRE, Jean-Max, MAURINE, Philippe, GUILLAUME-SAGE, Ludovic, CLEDIERE, Jessy et TRIA, Assia (2014). « Efficiency of a glitch detector against electromagnetic fault injection ». In : *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. New Jersey : IEEE Conference Publications, p. 1–6. DOI : 10.7873/DATE.2014.216 (cf. p. 24).

ZUSSA, Loic, DUTERTRE, Jean-max, CLÉDIÈRE, Jessy, ROBISSON, Bruno et TRIA, Assia (2012). « Investigation of timing constraints violation as a fault injection means ». In : *27th Conference on Design of Circuits and Integrated Systems (DCIS)*. Avignon, France (cf. p. 20).

Bibliographie personnelle

DEHBAOUI, Amine, MIRBAHA, Amir-Pasha, MORO, Nicolas, DUTERTRE, Jean-Max et TRIA, Assia (2013). « Electromagnetic glitch on the AES round counter ». In : *4th International conference on Constructive Side-Channel Analysis and Secure Design*. Paris, France, p. 17–31 (cf. p. xiv, 23, 29, 47).

HEYDEMANN, Karine, MORO, Nicolas, ENCRENAZ, Emmanuelle et ROBISSON, Bruno (2013). « Formal verification of a software countermeasure against instruction skip attacks ». In : *2nd Workshop on Security Proofs for Embedded Systems (PROOFS)*. Santa Barbara, California, USA (cf. p. 77).

MORO, Nicolas, DEHBAOUI, Amine, HEYDEMANN, Karine, ROBISSON, Bruno et ENCRENAZ, Emmanuelle (2013). « Electromagnetic Fault Injection : Towards a Fault Model on a 32-bit Microcontroller ». In : *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. Santa Barbara, California, USA : IEEE, p. 77–88. DOI : 10.1109/FDTC.2013.9 (cf. p. xiv, 53).

MORO, Nicolas, HEYDEMANN, Karine, DEHBAOUI, Amine, ROBISSON, Bruno et ENCRENAZ, Emmanuelle (2014). « Experimental evaluation of two software countermeasures against fault attacks ». In : *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. Arlington, Virginia, USA : IEEE, p. 112–117. DOI : 10.1109/HST.2014.6855580 (cf. p. xv, 53, 109).

MORO, Nicolas, HEYDEMANN, Karine, ENCRENAZ, Emmanuelle et ROBISSON, Bruno (2014). « Formal verification of a software countermeasure against instruction skip attacks ». In : *Journal of Cryptographic Engineering* 4.3, p. 145–156. DOI : 10.1007/s13389-014-0077-7 (cf. p. xv, 77).

Table des figures

Liste des tableaux

Liste des acronymes

ABI
Application Binary Interface. 33, 80

AES
Advanced Encryption Standard. 17, 23, 24, 26, 43, 44, 48, 50, 106, 107, 110, 123, 131, 139

AHB
Advanced High-performance Bus. 64, 65

BDD
Binary Decision Diagram. 89, 91, 102

CCD
Charge-Coupled Device. 8

CMOS
Complementary Metal-Oxyde Semiconductor. 5, 6, 8, 17, 30

CPA
Correlation Power Analysis. 131

CTL
Computation Tree Logic. 88, 97, 98, 100

DES
Data Encryption Standard. 14, 24, 44

DFA
Differential Fault Analysis. 23

DPA
Differential Power Analysis. 11, 12, 17, 23

IFA
Ineffective Fault Analysis. 23, 24

IP
Intra-Procedure call scratch register. 33

JTAG

Joint Test Action Group. 39–41, 43

LTL

Linear Temporal Logic. 89

MPU

Memory Protection Unit. 117

MSP

Main Stack Pointer. 33

NIST

National Institute of Standards and Technology. 44

PSP

Process Stack Pointer. 33

RSA

Rivest Shamir Adleman. 8, 11, 19, 25, 26

RTL

Register-Transfer Level. x, 53, 54, 64, 67, 73, 74, 91

SET

Single Event Transient. 18

SMT

Satisfiability Modulo Theories. 89, 102

SPA

Simple Power Analysis. 10, 14

SRAM

Static Random Access Memory. 18, 19, 24, 36, 58, 62, 64