Compilation pour l'application de contre-mesures contre les attaques par canal auxiliaire

Nicolas Belleville

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Nicolas BELLEVILLE

Thèse dirigée par **Henri-Pierre CHARLES**, Directeur de Recherche, CEA
et codirigée par **Karine HEYDEMANN**, Maître de conférences, Sorbonne Université - Laboratoire d'Informatique de Paris 6

préparée au sein du **Laboratoire Infrastructure Atelier Logiciel pour Puce, CEA Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Compilation pour l'application de contre-mesures contre les attaques par canal auxiliaire

Thèse soutenue publiquement le **21 novembre 2019**,
devant le jury composé de :

Monsieur Lilian BOSSUET
Professeur, Laboratoire Hubert Curien, Président
Monsieur Erven ROHOU
Directeur de Recherche, INRIA, Rapporteur
Monsieur Louis GOUBIN
Professeur, Université de Versailles-St-Quentin-en-Yvelines, Rapporteur
Monsieur Emmanuel PROUFF
Expert sécurité des systèmes embarqués, ANSSI - Agence nationale de la sécurité des systèmes d'information, Invité
Monsieur Benoît GÉRARD
Chercheur associé, DGA / IRISA, Invité
Monsieur Henri-Pierre CHARLES
Directeur de Recherche, CEA, Directeur de thèse
Madame Karine HEYDEMANN
Maître de Conférences, Sorbonne Université - Laboratoire d'Informatique de Paris 6, Co-directrice de thèse
Monsieur Damien COUROUSSÉ
Ingénieur de Recherche, CEA, Encadrant de thèse

Remerciements

Je tiens tout d'abord à remercier Messieurs Erven Rohou et Louis Goubin d'avoir accepté d'être rapporteurs de cette thèse. Je souhaite également remercier Messieurs Emmanuel Prouff et Lilian Bossuet d'avoir accepté d'être membres de mon jury. Je remercie également Monsieur Benoît Gérard d'avoir accepté d'évaluer ma thèse à mi-parcours et pour les conseils qu'il m'a alors donné, puis d'avoir accepté de faire parti de mon jury.

Je tiens à remercier Damien, Karine et Henri-Pierre pour m'avoir encadré tout au long de ma thèse, et leurs précieux conseils et encouragements. Je souhaite également remercier tous les collègues que j'ai eu au LIALP, j'ai énormément apprécié la bonne ambiance du labo et tous nos échanges. J'aimerais remercier en particulier Thierno Barry pour toute l'aide qu'il m'a apporté lors de mon début de thèse.

Enfin, je souhaite remercier mes amis, ma famille et Marie pour leur soutien tout au long de cette thèse et pour tous les bons moments passés ensemble.

Résumé

Les systèmes embarqués et objets connectés sont aujourd'hui de plus en plus répandus. Contrairement à d'autres systèmes accessibles uniquement par le réseau, les systèmes embarqués sont accessibles physiquement par un attaquant. Celui-ci peut alors exploiter cette proximité physique pour monter des attaques par canal auxiliaire afin de compromettre ces systèmes ou leurs données. Ces attaques non intrusives ont ainsi montré une grande efficacité pour récupérer les clés cryptographiques utilisées dans de tels systèmes. Il est alors primordial de protéger les systèmes embarqués contre cette menace sérieuse. Les contre-mesures logicielles sont la plupart du temps appliquées manuellement par des experts. Dans cette thèse, nous proposons d'appliquer automatiquement ces contre-mesures au sein du processus de compilation. Nous proposons deux approches, l'une pour appliquer une contre-mesure de masquage booléen de premier ordre, l'autre pour appliquer une contre-mesure de polymorphisme de code. Nous apportons des réponses à plusieurs problèmes liés à la génération dynamique de code pour permettre l'utilisation du polymorphisme de code sur des systèmes contraints. Enfin, nous adaptons les contre-mesures choisies afin d'obtenir de meilleurs compromis entre performances et sécurité.

Abstract

Embedded systems and connected objects are increasingly used nowadays. Unlike some other systems accessible only through the network, embedded systems are physically accessible by an attacker. The latter can then exploit this physical proximity to mount side-channel attacks to compromise these systems or their data. These non-intrusive attacks have shown great effectiveness in recovering cryptographic keys used in such systems. Embedded systems must therefore be secured against this severe threat. Software countermeasures are most often applied manually by experts. In this thesis, we propose to automatically apply these countermeasures within the compilation process. We propose two approaches, one to apply a first-order Boolean masking countermeasure, the other to apply a code polymorphism countermeasure. We address several problems related to dynamic code generation to enable the use of code polymorphism on constrained systems. Finally, we adapt the chosen countermeasures to obtain a better trade-off between performance and security.

Table des matières

Table des figures

Liste des tableaux

Liste des Listings

Liste des Algorithmes

Chapitre 1

Introduction

Sommaire

1.1 Introduction

Aujourd'hui, les systèmes embarqués font partie intégrante de notre vie quotidienne et constituent le plus grand segment du marché de l'électronique grand public. Le nombre de systèmes embarqués qu'une personne manipule chaque jour devrait augmenter massivement avec le développement de l'Internet des Objets. En 2008, ce nombre était déjà élevé puisqu'une personne utilisait environ 230 puces embarquées chaque jour en moyenne [Sifakis, 2011].

Les systèmes embarqués et objets connectés constituent des cibles intéressantes pour un attaquant pour plusieurs raisons. D'abord, ces systèmes manipulent souvent des données sensibles. Par exemple, des données privées critiques sont traitées quotidiennement par les cartes de paiement, les cartes de transport, les smartphones, les GPS, etc. Ensuite, de plus en plus de ces systèmes sont connectés à internet. Un attaquant réussissant à compromettre un grand nombre de ces systèmes peut alors les utiliser pour mener des attaques par déni de service (DDoS) [INRIA, 2019], comme cela a été le cas par exemple dans une université dont le réseau a été rendu inopérant par une attaque utilisant plus de 5000 objets connectés [Famien, 2017].

De telles attaques peuvent avoir des conséquences de bien plus grande ampleur. Ainsi, une attaque par déni de service massive sur des infrastructures DNS a paralysé toute une partie d'internet pendant plusieurs heures en 2016 [Josset, 2016].

Par conséquent, la sécurité des systèmes embarqués et objets connectés se révèle être une préoccupation majeure tant pour les industriels que pour les organismes publics.

Les dispositifs sécurisés ont recours à la cryptographie pour protéger les données sensibles, ou pour authentifier des mises à jour du micrologiciel afin de garantir que ces mises à jour proviennent du fabricant et non pas d'un attaquant. Bien que ces systèmes utilisent des algorithmes cryptographiques robustes contre la cryptanalyse, les attaquants peuvent exploiter un accès physique à un dispositif soit pour extraire des données sensibles telles qu'une clé cryptographique, soit pour contourner une authentification, soit dans certains cas pour rétro-ingéniérier des propriétés intellectuelles. Ces attaques, connues sous le nom d'attaques physiques, se divisent en deux catégories :

1. Les attaques par canal auxiliaire, introduites par [Kocher, 1996], exploitent la corrélation entre les données traitées à l'intérieur de la puce et des grandeurs physiques pouvant être mesurées de l'extérieur. Ces grandeurs physiques comprennent notamment la consommation d'énergie de l'appareil [Brier et al., 2004, Kocher et al., 1999, Mangard et al., 2007, VanLaven et al., 2005, Peeters, 2013], le rayonnement électromagnétique [Agrawal et al., 2003, Gandolfi et al., 2001], les émissions acoustiques [Genkin et al., 2017], le temps d'exécution [Dhem et al., 2000, Kocher, 1996], etc.
2. Les attaques par injection de fautes, introduites par [Boneh et al., 1997], exploitent l'effet d'une perturbation délibérée sur un système pendant son fonctionnement. Les attaques par injection de fautes peuvent être effectuées au moyen d'un faisceau lumineux ou laser [Dutertre et al., 2014, Skorobogatov et Anderson, 2003], d'une injection électromagnétique [Quisquater et Samyde, 2001, Ordas et al., 2015b, Dehbaoui et al., 2012], d'une variation de la tension d'alimentation [Aumüller et al.,

2003, Carpi et al., 2013], d'une perturbation du signal d'horloge [Agoyan et al., 2010], d'un changement de température [Skorobogatov, 2009, Hutter et Schmidt, 2014], etc.

De nombreuses protections pour contrer les attaques physiques ont été proposées, à la fois logicielles [Barenghi et al., 2010, De Keulenaer et al., 2016, Agosta et al., 2015a, Couroussé et al., 2016, Mangard et al., 2007] et matérielles [Avirneni et Somani, 2014, Güneysu et Moradi, 2011, Singh et al., 2018, Sasdrich et al., 2017, Wang et al., 2017]. Il existe également des approches mixtes matériel-logiciel [Danger et al., 2014, Bayrak et al., 2012, Ambrose et al., 2007].

Par le passé, ces contre-mesures ont principalement été appliquées sur des dispositifs nécessitant une certification *critères communs* comme les cartes bancaires [com]. Les dispositifs embarqués ou objets connectés n'ont, pour la plupart, pas de certification de niveau de sécurité à passer avant d'être commercialisés, bien que de nouveaux processus de certifications nationaux visant ces plate-formes aient été mis en place dans plusieurs pays récemment, comme la CSPN (Certification de Sécurité de Premier Niveau) en France [ANSSI, 2008], la BSZ en Allemagne [Federal Office for Information Security, 2019] ou encore la LINCE en Espagne [CCN, 2018].

1.2 Problématique

Il y a un besoin fort d'appliquer des contre-mesures contre les attaques physiques sur les systèmes embarqués et objets connectés susceptibles d'être la cible de ces attaques. Ce besoin se heurte à plusieurs contraintes. Premièrement, les solutions matérielles peuvent s'avérer trop coûteuses pour ces dispositifs qui sont soumis à de fortes exigences en matière de coûts. Aussi, l'application de solutions logicielles est le plus souvent manuelle, demande un haut niveau d'expertise pour adapter la contre-mesure à un algorithme, et doit le plus souvent être faite sur le code assembleur ou en écrivant le code source sous une forme complexe pour que la protection soit bien présente dans le code final. En effet, le processus de compilation peut altérer voire éliminer les protections appliquées.

Ainsi, l'intégration des contre-mesures est longue et coûteuse. Pour répondre à cette difficulté d'application des contre-mesures, il est nécessaire de développer des approches automatisées. Celles-ci doivent prendre en compte l'étape de compilation des programmes afin d'éviter que les transformations de code interne au compilateur n'affecte les protections, mais peuvent aussi chercher à tirer profit du compilateur pour obtenir de meilleurs compromis sécurité / performance.

1.3 Contributions

Dans ce manuscrit, nous proposons d'étudier comment des contre-mesures logicielles existantes contre les attaques par canal auxiliaire peuvent être intégrées au processus de compilation, afin de tirer profit du compilateur pour les optimisations en performance, tout en réduisant les risques que les contre-mesures soient altérées par les différentes transformations appliquées par le compilateur.

Nous faisons d'abord une analyse de l'état de l'art sur les techniques d'automatisation d'application de contre-mesures contre les attaques physiques. Nous apportons ensuite des contributions au problème d'automatisation pour les deux grands principes de contre-mesures contre les attaques par canal auxiliaire : le masquage et la dissimulation. Nous montrons pour ces deux principes que le compilateur peut être un outil adapté pour leur application automatisée : se placer au sein du compilateur permet d'effectuer à la fois des transformations de haut et de bas niveau, et de choisir le placement de l'application de la contre-mesure par rapport aux diverses optimisations du compilateur.

La première contribution majeure concerne la compilation d'une contre-mesure de masquage et le développement d'optimisations pour celle-ci. Nous montrons comment appliquer une contre-mesure de masquage booléen du premier ordre au sein du compilateur. Nous choisissons d'appliquer une telle contre-mesure à la fin de la partie du compilateur indépendante de la cible, le *middle-end*, pour pouvoir supporter diverses architectures, et éliminer les risques liés aux nombreuses optimisations qui ont lieu en amont. Nous proposons une implémentation dans une passe ajoutée à LLVM. Nous utilisons une technique de masquage pour les tables constantes par évaluation de polynômes interpolateurs, encore jamais automatisée à la compilation. De plus, nous apportons des optimisations pour cette évaluation permettant de réduire le coût, souvent très élevé, du masquage de ces tables. Enfin, nous mettons en évidence les effets de la partie du compilateur dépendant de la cible (le *back-end*) et de la micro-architecture de la plate-forme cible sur la contre-mesure.

La seconde contribution majeure concerne la compilation d'une contre-mesure de dissimulation et l'amélioration de la contre-mesure. Nous choisissons une contre-mesure de dissimulation fortement configurable : le polymorphisme de code. Sa généricité permet de cibler tout type de codes. Nous montrons comment le back-end peut être modifiée pour appliquer cette contre-mesure. Nous expliquons comment profiter des optimisations statiques du compilateur ainsi que de ses capacités d'analyse de code pour obtenir une génération dynamique de code polymorphe efficace. Nous proposons également une transformation de code appelée bruit dynamique afin de permettre de meilleurs compromis entre sécurité et performance. Nous montrons que cette nouvelle transformation permet, pour un niveau de sécurité souhaité, de réduire beaucoup plus la fréquence à laquelle un code différent doit être généré. Enfin, nous observons l'impact de la contre-mesure en sécurité et en performance sur des codes variés, et constatons que la contre-mesure permet d'obtenir de hauts niveaux de sécurité tout en conservant des surcoûts en performance contenus.

1.4 Plan du manuscrit

La suite de ce manuscrit est organisée comme suit.

Le chapitre 2 "Attaques physiques et principes des contre-mesures" est une introduction aux attaques physiques ainsi qu'aux grands principes de contre-mesures logicielles existant.

Le chapitre 3 "État de l'art sur l'application automatisée de contre-mesures" présente un état de l'art sur l'application automatisée de contre-mesures contre les attaques physiques. Il propose une classification des différentes approches de l'état de l'art en fonction du niveau - code source, assembleur ou compilateur - auquel la contre-mesure est appliquée. Les avantages et inconvénients de chaque niveau sont exposés. Nous expliquons alors pourquoi nous pensons que le niveau du compilateur est le plus adapté pour l'application automatisée de contre-mesures contre les attaques physiques.

Le chapitre 4 "Application du masquage" présente une approche d'application automatisée d'une contre-mesure de masquage à la compilation. Nous proposons un algorithme d'application de la contre-mesure destiné à travailler sur une forme SSA, et plusieurs optimisations en performance. La mise en œuvre de cet algorithme dans LLVM est ensuite détaillée. Nous évaluons cette approche en performance et en sécurité.

Le chapitre 5 "Application du polymorphisme de code" présente une approche d'application automatisée d'une contre-mesure de polymorphisme de code. Nous proposons d'utiliser un compilateur statique pour générer automatiquement des générateurs de code spécialisés, qui pourront faire varier le code lors de l'exécution. Nous introduisons une nouvelle transformation dynamique de code permettant d'obtenir de meilleurs compromis sécurité / performance. Nous décrivons un mécanisme de changement de permissions mémoire s'appuyant sur le caractère spécialisé de nos générateurs de code afin qu'eux seuls puissent émettre du code dans une zone mémoire qui sera ensuite exécutée. Nous montrons également comment allouer la mémoire de manière réaliste tout en évitant les dépassements de tampon. Enfin, nous évaluons en performance et en sécurité plusieurs codes sécurisés avec diverses configurations.

Le chapitre 6 "Conclusion et perspectives" effectue un bilan sur les contributions apportées et donne différentes pistes de recherche, à court et moyen terme.

Chapitre 2

Attaques physiques et principes des contre-mesures logicielles associées

Sommaire

2.1 Introduction

Ce chapitre vise à présenter les attaques physiques, ainsi que les grands principes des contre-mesures logicielles déployées contre ces attaques. Il s'étend plus amplement sur les attaques par canal auxiliaire, qui forment le sujet principal de cette thèse. Néanmoins, les attaques par injections de fautes et leur principes de protections sont également introduits de manière à pouvoir aussi analyser les approches automatisées d'application de telles contre-mesures.

2.2 Attaques physiques

Cette section présente les attaques par canal auxiliaire et les attaques en fautes.

2.2.1 Attaques par canal auxiliaire

Comme déjà mentionné dans le chapitre précédent (section 1.1), les attaques par canal auxiliaire exploitent le lien entre les données et instructions manipulées par un processeur et des grandeurs physiques mesurables pour retrouver des données manipulées par le programme ou retrouver la suite d'instructions exécutées.

Lors d'une attaque, l'attaquant mesure une grandeur physique pendant que le processeur exécute le programme ciblé. Il déduit ensuite les données ou instructions manipulées par le processeur à partir de ces mesures, en comparant les mesures avec un modèle comportemental. Dans le cadre des travaux menés dans cette thèse, nous nous sommes concentrés sur les attaques par canal auxiliaire qui exploitent la consommation d'énergie ou les émissions électromagnétiques.

Dans le cas d'une analyse de consommation par corrélation (CPA) [Mangard et al., 2007], l'attaquant choisit les données qu'il fournit en entrée au programme (ou recueille les données de sortie du programme). Pour trouver la clé de chiffrement d'un AES (Advanced Encryption Standard [Daemen et Rijmen, 2002]), il procède un octet de clé à la fois. Chaque octet est trouvé comme suit : l'attaquant place une sonde électromagnétique sur le processeur, ou mesure directement sa consommation électrique avec un oscilloscope. Il effectue des mesures lors de plusieurs exécutions AES. Pour chaque nouvelle exécution, il donne un texte clair aléatoire au programme. Il calcule les consommations théoriques pour chaque valeur de l'octet de clé recherché en utilisant un modèle de consommation (par exemple le poids de Hamming de la valeur retournée par la SBox de la première ronde). Il compare alors les mesures obtenues sur plusieurs exécutions avec les consommations théoriques en utilisant un opérateur statistique, par exemple la corrélation de Pearson. L'hypothèse d'octet qui donne la corrélation la plus forte entre les consommations théoriques et celles mesurées correspond à la valeur réelle de l'octet de clé. Selon les cibles, le nombre de mesures nécessaires pour réussir à récupérer la clé varie.

2.2.2 Attaques en fautes

Comme également mentionné dans le chapitre précédent (section 1.1), les processeurs peuvent également subir des attaques par injection de fautes.

Les effets des fautes sont multiples et dépendent de la cible, du code visé, ainsi que des moyens d'injection [Yuce et al., 2018]. Les effets observés et rapportés dans la littérature comportent notamment :

- le changement de la valeur d'un bit dans un registre ou une cellule de mémoire [Biehl et al., 2000, Boneh et al., 2001, Bar-El et al., 2006, Ordas et al., 2015b, Dusart et al., 2003, Ordas et al., 2015a],
- la modification aléatoire d'une valeur lors de son transfert entre le CPU et la mémoire dynamique ou la mémoire non volatile [Moro et al., 2013, Dureuil et al., 2015],
- le remplacement d'instruction lorsqu'un chargement d'instruction (`fetch`) est corrompu [Moro et al., 2013, Colombier et al., 2019].

Les attaques par injection de fautes permettent de détourner le flot d'exécution d'un programme [Dureuil, 2016] (par exemple pour contourner une vérification de mot de passe d'un VerifyPIN), ou de récupérer des informations sur des données manipulées par le programme (par exemple une clé cryptographique [Dehbaoui et al., 2013]). Pour récupérer une donnée secrète, l'attaquant peut analyser la sortie erronée qui résulte de ces fautes. Il utilise pour cela un modèle d'attaque en faute qui relie la sortie prévue et les sorties possibles en présence des fautes. Il peut aussi exploiter une absence d'erreur sur la sortie [Sung-Ming Yen et Joye, 2000].

2.2.3 Attaques combinées

Les attaques combinées sont des attaques physiques qui associent l'analyse des canaux auxiliaires et l'injection de fautes. Actuellement, toutes les attaques en faute sont combinées en pratique avec une observation de canal auxiliaire. Cela permet de superviser l'injection de la faute, notamment pour contrôler précisément le moment où la faute est injectée [Timmers et Spruyt, 2016]. Aussi, certaines attaques utilisent l'analyse des canaux auxiliaires et l'attaque par injection de fautes comme étapes d'une attaque de plus large envergure [Amiel et al., 2007]. Plusieurs approches ont montré que ces attaques peuvent être efficaces sur des implémentations qui sont protégées à la fois contre les attaques par canal auxiliaire et les attaques par injection de fautes, par exemple sur un AES masqué ayant une contre-mesure contre les attaques par injection de fautes par redondance d'opérations [Roche et al., 2011, Dassance et Venelli, 2012], ou encore sur une implémentation ECC comportant des protections variées [Fan et al., 2011].

2.3 Contre-mesures

Pour faire face aux attaques par canal auxiliaire et aux attaques en fautes, de nombreuses contre-mesures ont été proposées. Cette section présente les principales catégories de

contre-mesures contre les attaques par canal auxiliaire et contre les attaques par injection de fautes.

Pour les attaques par canal auxiliaire, nous nous concentrons sur les canaux auxiliaires liés à la consommation d'énergie ou aux émissions électromagnétiques.

2.3.1 Dissimulation

Une contre-mesure de dissimulation vise à rendre les mesures de l'attaquant trop bruitées pour être exploitables [Clavier et al., 2000].

De nombreuses techniques de dissimulations logicielles ont été proposées :
- du mélange d'instructions ou d'opérations indépendantes [Luo et al., 2015, Rivain et al., 2009, Agosta et al., 2013b, Couroussé et al., 2016],
- de l'insertion de délais aléatoires [Tunstall et Benoit, 2007, Coron et Kizhvatov, 2009, Coron et Kizhvatov, 2010, Couroussé et al., 2016],
- du préchargement aléatoire [Bayrak et al., 2011, Bayrak et al., 2015b],
- un choix aléatoire entre plusieurs variantes sémantiquement équivalentes [Agosta et al., 2012, Agosta et al., 2015b, Couroussé et al., 2016].

Ces techniques sont génériques et peuvent s'appliquer sur tout type de programme. Cependant, leur efficacité peut dépendre du programme à renforcer : par exemple l'efficacité d'un mélange d'instructions indépendantes dépend des permutations possibles entre les instructions et donc de leur degré de dépendance. Il est possible toutefois d'augmenter les possibilités par exemple en insérant des opérations inutiles comme proposé par [Luo et al., 2015].

De même, l'utilisation de variantes sémantiques pour remplacer des instructions a une efficacité variable suivant les programmes et les variantes sémantiques connues : si des règles de remplacement sont définies uniquement pour quelques opérations, un programme qui ne contient aucune de ces opérations ne pourra pas être renforcé.

De plus, l'insertion de délais aléatoires par l'insertion d'instructions `NOP` ou par l'utilisation de boucles factices ayant un nombre d'itérations aléatoire sont des techniques limitées car le comportement des instructions introduites diffère trop du comportement des instructions originales du programme, ce qui permet de construire des attaques capables de discriminer les portions utiles des portions inutiles [Durvaux et al., 2013].

Pour ces raisons, il est préférable de combiner plusieurs techniques pour mieux s'adapter aux différents programmes et pour offrir une meilleure résistance aux différentes attaques. À titre d'exemple, [Agosta et al., 2012] utilisent à la fois des variantes sémantiques, du mélange d'instructions et des permutations d'accès aux tableaux, et [Couroussé et al., 2016] utilisent à la fois de mélange de registres et d'instructions, des variantes sémantiques et l'insertion d'instructions de bruit. La combinaison de plusieurs techniques de dissimulation permet d'obtenir des approches plus configurables, et il est alors possible d'adapter les protections déployées en fonction des contraintes du système.

L'application de contre-mesures de dissimulation peut se faire de manière statique (le code exécutable contient différents chemins d'exécution) [Bayrak et al., 2011, Bayrak et al., 2015a, Agosta et al., 2015b] ou dynamique (le code exécuté varie par génération

ou modification du code à l'exécution) [Agosta et al., 2012, Couroussé et al., 2016]. Les approches statiques sont souvent coûteuses en terme de taille de code. Les approches dynamiques, quant à elles, nécessitent d'accéder à une zone mémoire avec les droits d'écriture et d'exécution à la fois, ce qui peut amoindrir la sécurité du système.

Il est à noter que la dissimulation a également été utilisée en dehors du cadre des canaux auxiliaires de consommation électrique et d'émission électromagnétique, notamment pour renforcer des programmes contre les attaques exploitant le cache [Crane et al., 2015, Rane et al., 2015, Homescu et al., 2013].

2.3.2 Masquage

Le masquage est une contre-mesure destinée à supprimer le lien direct entre les mesures et les données secrètes manipulées par le processeur [Goubin et Patarin, 1999, Mangard et al., 2007]. Pour ce faire, l'algorithme du programme cible est modifié de sorte que tous les résultats intermédiaires qui dépendent des données secrètes soient séparés en plusieurs parties (appelées *shares* dans la suite de ce manuscrit), qui sont toutes requises pour recomposer les résultats, et sont toutes statistiquement indépendantes des résultats. Un masquage est dit d'ordre d s'il décompose un secret en $d+1$ shares. On appelle *masques* les nombres aléatoires utilisés pour faire cette décomposition.

Il existe différentes manières de séparer un secret en shares, ce qui donne différents types de masquage. Le masquage booléen utilise le OU EXCLUSIF (xor) pour effectuer cette séparation [Ishai et al., 2003] : on tire d masques $\{m_1, ..., m_d\}$, on calcule $s_{d+1} = v \oplus m_1 \oplus m_2 \oplus ... \oplus m_d$ où v est la variable à masquer, et on obtient ainsi $d+1$ shares (les d masques et s_{d+1}). Le masquage arithmétique utilise l'addition modulaire de la même manière que le OU EXCLUSIF est utilisé pour le masquage booléen. Le masquage multiplicatif repose sur la multiplication dans un corps fini, mais son principe est légèrement plus compliqué que les deux schémas précédents, puisque la valeur 0 resterait constamment égale à 0, quels que soient les nombres aléatoires utilisés, si la multiplication était utilisée telle quelle pour séparer le secret en shares. Le masquage multiplicatif combine donc la multiplication avec une fonction Dirac pour pouvoir masquer la valeur 0 [Genelle et al., 2010, Genelle et al., 2011].

Par exemple, pour le masquage booléen à l'ordre 1 et un secret s, il y a deux shares s_0 et s_1 avec $s_0 = s \oplus m$ et $s_1 = m$ où m est un nombre aléatoire. Les calculs sont ensuite effectués avec les deux shares pour pouvoir reconstruire la valeur finale. Le masque est défini aléatoirement à chaque exécution, de sorte que les valeurs des shares changent de façon aléatoire d'une exécution à l'autre. Ainsi, en effectuant tous les calculs en utilisant les shares séparément, les observations d'un attaquant via un canal auxiliaire ne lui donneront plus directement d'information sur le secret.

Les masques sont souvent réutilisés pour masquer plusieurs variables différentes pour limiter le coût des appels au générateur de nombre aléatoire. Si deux variables masquées avec les mêmes masques sont combinées, il peut alors y avoir une fuite dépendant des secrets, on dit qu'il y a un démasquage. Pour résoudre ce problème, un remasquage peut être effectué : un nouveau masque est tiré et les shares d'une des variables sont alors combinés avec ce nouveau masque.

Le type de masquage est choisi en fonction du code à sécuriser de manière à utiliser le type permettant le coût le plus faible pour le code considéré. Ainsi, si le code contient uniquement des opérations booléennes, le masquage booléen est privilégié, s'il contient uniquement des opérations arithmétiques, le masquage arithmétique est privilégié, et s'il contient uniquement des multiplications ou des exponentiations dans des corps finis, le masquage multiplicatif est privilégié.

Comme les codes à masquer peuvent contenir des opérations dans plusieurs de ces catégories, des algorithmes ont été développés pour pouvoir passer d'un type de masquage à un autre au cours d'un algorithme sans effectuer de démasquage [Goubin, 2001, Coron et al., 2014a, Genelle et al., 2010]. Il est également possible d'effectuer une addition arithmétique au sein d'un code masqué avec un masquage booléen sans effectuer une telle conversion [Karroumi et al., 2014]. L'utilisation de tels algorithmes permet notamment de masquer des fonctions de chiffrement ARX (`add`, `rotate`, `xor`) qui mélangent des additions arithmétiques avec des opérations booléennes, comme SPARX [Dinu et al., 2016] par exemple.

Pour masquer des opérations qui n'entrent pas dans ces catégories, comme les opérations réalisées avec des tables associatives (*lookup tables*), et en particulier les SBoxes tabulées, deux approches générales sont souvent utilisées [Coron, 2014, Carlet et al., 2012, Coron et al., 2018, Tang et al., 2017, Coron et al., 2014c, Roy et Vivek, 2013] :

— effectuer des accès à des tables masquées [Mangard et al., 2007, Coron et al., 2018, Tang et al., 2017],
— remplacer la table associative par l'évaluation d'un polynôme interpolateur et masquer le code permettant cette évaluation [Carlet et al., 2012, Coron et al., 2014c, Roy et Vivek, 2013].

À l'ordre 1, la première solution consiste à choisir 2 masques, un pour masquer les valeurs d'entrées de la table associative, et un pour masquer les valeurs de sortie de la table. Il est alors nécessaire de recalculer la table en tenant compte de ces masques. Cependant, le rafraîchissement des masques impose de recalculer la table masquée après chaque accès à la table, et un attaquant est susceptible de déterminer les masques utilisés lors d'un tel remasquage [Tunstall et al., 2014].

La deuxième solution consiste à trouver un polynôme interpolateur de la table, puis de masquer le code d'évaluation de ce polynôme comme on masquerait n'importe quelle fonction. Le polynôme est défini dans un corps fini où l'addition est un OU EXCLUSIF, pour être sous une forme favorable au masquage booléen. Cette solution est plus complexe à mettre en place que la solution utilisant une table masquée à l'ordre 1, mais évite le problème du remasquage de la table. De plus, son coût en performance est intéressant à la fois à l'ordre 1 et aux ordres supérieurs [Coron et al., 2014c].

D'autres approches existent pour masquer certaines tables associatives particulières, comme l'approche proposée par [Rivain et Prouff, 2010] pour la SBox de l'AES.

Le masquage s'appuie sur un modèle de fuite : en valeur et/ou en transition. Un modèle de fuite en valeur suppose que les valeurs manipulées par les instructions transparaissent directement par canal auxiliaire. Un modèle de fuite en transition suppose que deux variables utilisant successivement la même ressource physique vont créer une fuite dépendant de la valeur de ces deux variables, par exemple une fuite correspondant au OU EXCLUSIF de ces variables.

Le masquage oblige l'attaquant à combiner plusieurs instants des mesures pour obtenir une information dépendant des $d+1$ shares à la fois, afin d'extraire une information sur le secret. Cependant, il est complexe de savoir quels instants recombiner. Une approche force brute consiste alors à effectuer toutes les combinaisons possibles de $d+1$ échantillons des mesures, ce qui augmente drastiquement le nombre d'échantillons à traiter, et donc la complexité de l'attaque. Les attaques recombinant ainsi les mesures sont dites d'ordre supérieur, leur ordre étant déterminé par le nombre d'échantillons à recombiner. En théorie, l'ordre minimal nécessaire pour effectuer une attaque sur une implémentation masquée est directement lié au nombre de shares utilisés. En pratique, cela dépend des différences entre le modèle de fuite théorique et les caractéristiques des fuites réelles, et des éventuelles transformations de code qui auraient pu atténuer la contre-mesure [Balasch et al., 2015].

2.3.3 Détection des fautes

Une contre-mesure de détection des fautes a pour objectif de détecter une faute, ce qui permet ensuite de moduler la réponse en conséquence (par exemple, en détruisant le système).

Le type de fautes détectées varie d'une contre-mesure à l'autre. L'objectif d'une contre-mesure peut être par exemple de :

— détecter les fautes sur des données,
— détecter le saut d'une instruction,
— détecter un détournement du flot de contrôle.

Les contre-mesures contre les fautes reposent très souvent sur de la redondance de données et d'instructions.

Les données peuvent être encodées en mémoire en utilisant de la redondance d'information [Bringer et al., 2014]. Par exemple, on peut accoler aux données un bit de parité : la valeur de ce bit est obtenu en effectuant le OU EXCLUSIF de tous les autres bits de la donnée. Lors de l'utilisation de la donnée, on peut alors vérifier que la valeur de ce bit est toujours cohérente. Si le OU EXCLUSIF de tous les autres bits de la donnée n'est pas égal à la valeur du bit de parité, une faute est détectée.

Les instructions peuvent aussi être dupliquées, en dupliquant les données dans des registres différents, afin d'effectuer deux fois le même calcul et de comparer ensuite le résultat [Barenghi et al., 2010]. Si le résultat des deux instructions diffère, une faute est détectée.

Enfin, certaines contre-mesures ajoutent du code permettant d'effectuer des contrôles de l'intégrité du flot de contrôle et de l'exécution [de Clercq et Verbauwhede, 2017].

2.3.4 Tolérance aux fautes

Une contre-mesure de tolérance aux fautes vise à s'assurer qu'une faute n'altère pas le bon fonctionnement d'un programme.

Les principes employés pour obtenir de la tolérance aux fautes sont similaires à ceux utilisés pour la détection des fautes. Par exemple, des codes correcteurs d'erreurs peuvent permettre de corriger des fautes sur les données, et ainsi d'obtenir une tolérance aux fautes sur les données [Barenghi et al., 2010]. La duplication d'instructions peut également être implémentée de manière à obtenir une tolérance au saut d'une instruction plutôt qu'une détection d'erreur [Moro et al., 2014].

2.4 Compilation et application des contre-mesures

Nous avons passé en revue différent types de protections. Celles-ci peuvent être déployées sur différentes représentations du code, notamment sur du code source, du code assembleur, ou une représentation intermédiaire telle que celles utilisées par un compilateur. Pour comprendre les différences entre ces représentations et les différentes techniques d'application, nous présentons ici le processus de compilation. Cette présentation sera également l'occasion de présenter des notions utiles pour la suite de ce manuscrit.

2.4.1 Compilation

La compilation est le processus de traduction d'un code source en un programme binaire pour une architecture cible [Muchnick, 1997, Appel et Ginsburg, 2004, Srikant et Shankar, 2007].

Les compilateurs sont généralement divisés en 3 parties.

- La partie avant, le *front-end*, est chargée de parser le code source et de générer une représentation intermédiaire (IR).
- La partie intermédiaire, le *middle-end*, est responsable des optimisations indépendantes de l'architecture cible. Elle est composée d'une séquence de passes d'analyse et de transformation qui visent à optimiser le code IR (réduction du nombre de calculs, élimination de code inutile,...).
- La partie arrière, le *back-end*, est en charge de générer du code adapté à une architecture cible. En particulier, elle est responsable de la sélection des instructions, de l'allocation des registres, et de l'ordonnancement des instructions, mais effectue également d'autres optimisations.

Le middle-end et le back-end du compilateur comportent un enchaînement d'analyses et de transformations de code. Chaque analyse ou transformation est appelée une *passe*.

Les notions suivantes seront utiles pour la suite de ce manuscrit :

Définition 1. On appelle *bloc de base* une suite maximale d'instructions qui comporte un seul point d'entrée et un seul point de sortie.

Définition 2. On appelle *forme SSA* (*single static assignment*) une représentation de code dans laquelle chaque variable n'est assignée qu'une seule fois.

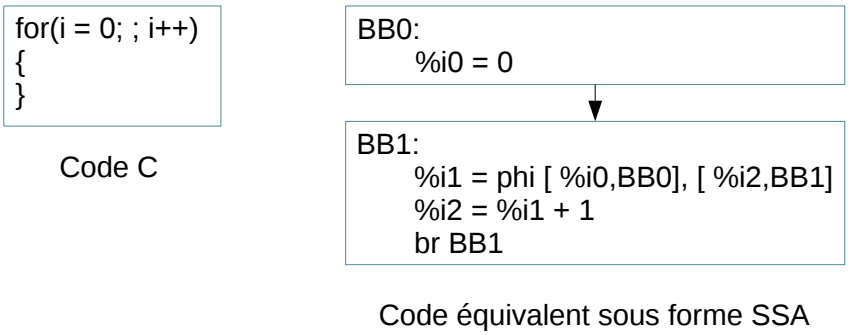

FIGURE 2.1 – Exemple d'utilisation d'un nœud ϕ pour représenter l'incrément d'un compteur de boucle sous forme SSA

Comme il peut y avoir plusieurs affectations d'une variable sur des chemins différents dans un code source, il est nécessaire sous forme SSA de pouvoir choisir la bonne valeur, et d'avoir un moyen de l'exprimer dans le code : c'est le rôle des nœuds ϕ.

Définition 3. Dans une forme SSA, on appelle *nœud* ϕ une instruction permettant d'assigner une variable à une valeur dépendant du chemin emprunté.

La figure 2.1 montre un exemple d'utilisation du nœud ϕ. Dans le code C, on trouve une boucle infinie qui incrémente la variable `i` en commençant à 0. Le code équivalent sous forme SSA comporte un premier bloc de base `BB0` pour l'initialisation de la variable, puis un bloc de base `BB1` qui correspond au corps de boucle. Dans `BB1`, `i` est représenté par les variables `%i1` et `%i2`. Le nœud ϕ affecte `%i0` à `%i1` si l'exécution vient de `BB0`, et affecte `%i2` à `%i1` si l'exécution vient de `BB1`. Ainsi, la variable `%i1` vaut 0 lors de la première itération de la boucle, puis est incrémentée à chaque nouvelle itération.

2.4.2 Application des contre-mesures

L'application des contre-mesures est souvent effectuée sur le code source ou sur le code assembleur.

L'application au niveau code source est semée d'embûches, puisque le code protégé doit être transformé par le compilateur pour produire le code binaire : plusieurs travaux ont montré que le compilateur peut altérer les contre-mesures contre les attaques par canal auxiliaire et par injection de fautes lorsque ces contre-mesures sont appliquées sur le code source [Balasch et al., 2015,Barbosa et al., 2009,Seuschek et al., 2017]. Le problème réside dans le fait que les modifications du code source visant à donner des propriétés de sécurité ne sont pas fonctionnelles, alors que le compilateur cherche uniquement à conserver les propriétés fonctionnelles du programme.

De nombreuses passes peuvent altérer les protections. Par exemple, dans le cas du masquage, les passes qui simplifient les opérations arithmétiques, l'ordonnancement des instructions et les passes d'allocation des registres peuvent altérer la contre-mesure. Le compilateur peut inverser l'ordre de deux OU EXCLUSIFS, ce qui peut dans certains cas révéler une donnée secrète. En cas d'ajout d'instructions de bruit ou de redondance d'instructions, les passes qui suppriment le code mort ou redondant peuvent constituer une

menace pour la contre-mesure. Dans le cas d'un mélange d'instructions, la passe d'ordonnancement des instructions peut également altérer la contre-mesure. Ces exemples sont loin d'être une liste exhaustive des passes qui peuvent altérer les contre-mesures. Une telle liste dépend du compilateur, de sa version, de l'architecture cible, du code, du niveau d'optimisation etc.

Pour contourner ce problème, il est possible de recourir à divers procédés :

- Le code peut être compilé en utilisant le niveau d'optimisation `-O0` pour que peu d'optimisations restent activées. Cependant, le processus de compilation reste risqué : le code passe toujours par les passes de sélection des instructions et d'allocation des registres (entre autres), et chacune de ces passes peut modifier certaines contre-mesures. De plus, compiler en `-O0` augmente la surface de code utilisable pour une attaque, et il y a beaucoup de sauvegardes et de restaurations des registres sur la pile, ce qui peut augmenter la quantité d'information disponible par un canal auxiliaire.
- Il est possible de désactiver certaines passes spécifiques en utilisant les options de ligne de commande du compilateur, mais cela ne résout pas le problème lié aux passes incontournables comme la sélection d'instructions et l'allocation des registres.
- En C/C++, on peut utiliser le mot-clé `volatile` pour forcer le compilateur à ne pas effectuer d'optimisation d'accès mémoire sur certaines variables sélectionnées. Cela est utile pour insérer des barrières ou éviter l'élimination de variables redondantes.
- On peut incorporer du code assembleur directement dans le code source. Cependant, cette solution complexifie le développement, car les développeurs doivent faire le lien entre les variables C/C++ et les registres physiques. De plus, le code source n'est plus portable et devient plus difficile à maintenir.

L'application de la contre-mesure au niveau assembleur permet d'éviter ces problèmes causés par la compilation, mais reste fastidieuse.

Une autre option est d'appliquer les protections à la compilation en insérant de nouvelles passes et en modifiant les passes existantes, ce qui nécessite des approches automatisées. Cela est l'objet du prochain chapitre, qui compare cette méthode avec les approches automatisées modifiant directement le code source ou le code assembleur. Avant d'y arriver, nous présentons les méthodes d'évaluation sécuritaire pour les attaques par canal auxiliaire.

2.5 Méthodes d'évaluation pour les attaques par canal auxiliaire

Dans cette section, nous présentons deux méthodes d'évaluation largement utilisées dans le cadre de l'évaluation de contre-mesures contre les attaques par canal auxiliaire : la CPA et le t-test non spécifique.

2.5.1 Évaluation en CPA

Le première approche consiste en la réalisation d'attaques. La CPA, présentée en section 2.2.1, est une attaque visant à récupérer un secret. À des fins d'évaluation sécuritaire, on peut prendre le rôle d'un attaquant pour évaluer la résistance du système.

Le principe est de regarder l'évolution des valeurs de corrélations pour les différentes hypothèses de clés à mesure que le nombre de traces augmente. Cela permet de déterminer le nombre de traces nécessaires à un attaquant pour retrouver l'information secrète.

Il est important de répéter plusieurs fois le processus, car le nombre de mesures nécessaires peut varier à cause du bruit de mesure par exemple. En répétant suffisamment de fois l'attaque, il est alors possible de calculer un taux de succès, c'est à dire la proportion d'attaques concluantes pour un nombre défini de mesures. Cela permet ensuite de déterminer le nombre minimal de mesures pour avoir une probabilité de succès supérieure à un seuil. Le seuil de 80% est souvent considéré. Un attaquant collectant le nombre de mesures correspondant à ce seuil aura alors 80% de parvenir à récupérer l'information secrète.

Cette méthode d'évaluation est liée à l'attaque en CPA, au modèle de consommation utilisé et à la variable secrète ciblée. Elle a l'avantage de fournir une estimation fidèle du temps nécessaire à un attaquant pour retrouver une information sensible du système, sous condition que l'attaquant déploie la même attaque.

2.5.2 Évaluation en t-test non spécifique

Une autre méthode d'évaluation largement répandue s'appuie sur une méthode statistique : le t-test. Cette méthode permet de mettre en évidence la présence d'une fuite d'information [Goodwill et al., 2011, Schneider et Moradi, 2015].

Pour effectuer un t-test non spécifique, deux ensembles de traces sont collectées :

- le premier ensemble de traces est collecté en faisant varier aléatoirement un ou plusieurs paramètres de la fonction ciblée afin que ses variables intermédiaires sensibles soient aléatoires,
- le second ensemble de traces est collecté en fixant tous les paramètres de la fonction ciblée afin que ses variables intermédiaires soient constantes.

Les deux ensembles de traces sont collectés de manière entrelacée pour éviter des faux positifs dus à l'état interne de la plate-forme [Schneider et Moradi, 2015].

Le t-test cherche ensuite à dissocier statistiquement les deux ensembles de traces. Dans le cas d'attaques par canal auxiliaire, la démonstration de la dissociabilité de mesures effectuées par canal auxiliaire révèle la présence d'une fuite d'information, qui pourrait potentiellement (mais pas toujours) être exploitée pour réussir une attaque, par exemple au moyen d'une CPA.

En pratique, pour un t-test non spécifique effectué sur une fonction de chiffrement, un ensemble de mesures est effectué avec un texte clair fixe, tandis que l'autre ensemble est effectué avec des textes clairs aléatoires. Pour différencier les deux ensembles de mesures,

le t-test calcule des valeurs appelées t-statistiques : si l'une de ces valeurs sort de l'intervalle $]-4{,}5\,;4{,}5[$, les deux ensembles sont considérés comme statistiquement dissociables avec un degré de fiabilité de 99,999% [Goodwill et al., 2011, Schneider et Moradi, 2015].

Le t-test non-spécifique a l'avantage d'être indépendant de tout modèle de fuite de consommation, de toute variable intermédiaire, et de pouvoir être calculé avec des ensembles de traces de taille raisonnable.

2.6 Conclusion

Dans ce chapitre nous avons présenté les principes des attaques physiques ainsi que ceux des contre-mesures logicielles associées. Les contre-mesures contre les attaques par injections de fautes visent souvent à obtenir de la tolérance aux fautes ou de la détection des fautes, tandis que les contre-mesures contre les attaques par canal auxiliaire reposent souvent sur des principes de masquage ou de dissimulation. Nous avons aussi présenté les méthodes d'évaluations sécuritaires les plus courantes pour les contre-mesures contre les attaques par canal auxiliaire.

Nous avons également introduit le processus de compilation et les risques liés à la compilation pour l'application des contre-mesures au niveau du code source. Nous avons notamment détaillé quelques techniques utilisées pour appliquer les contre-mesures à ce niveau et tenter d'éviter leur dégradation par la compilation. Ces techniques demandent à la fois du temps et de l'expertise pour être déployées, tout comme le déploiement de protections au niveau assembleur, ce qui motive l'emploi d'approches d'application automatisée de contre-mesure.

Le prochain chapitre présente et analyse les approches de l'état de l'art pour appliquer automatiquement de telles contre-mesures.

Chapitre 3

État de l'art sur l'application automatisée de contre-mesures contre les attaques physiques

Sommaire

3.1 Introduction

Ce chapitre vise à présenter les approches existantes pour appliquer automatiquement des contre-mesures contre les attaques physiques. Nous nous intéressons ici à la fois aux attaques par canal auxiliaire et aux attaques en fautes. En effet, bien que nous ne considérons que l'application de contre-mesures contre les attaques par canal auxiliaire dans cette thèse, l'étude des approches d'application automatisée de contre-mesures contre les attaques en fautes est intéressante sur plusieurs aspects. D'abord, les contre-mesures contre les deux types d'attaques peuvent être altérées par le compilateur. Il est donc pertinent d'analyser le niveau d'application de ces contre-mesures pour guider les choix de cette thèse. De plus, le fait de considérer les approches contre les attaques en fautes permet également d'obtenir des données de comparaison en termes de performance entre l'application d'une même contre-mesure à plusieurs niveaux, puisque certaines approches contre ces attaques ont ciblé l'application de la même contre-mesure, sur un code et une plate-forme identique.

Les approches sont présentées selon le niveau auquel la contre-mesure est appliquée, et selon le type de contre-mesure appliquée. La table 3.1 résume les différentes approches et leurs niveaux d'application.

3.2 Application automatisée de contre-mesures au niveau code source

Plusieurs approches ont proposé d'appliquer automatiquement des contre-mesures au niveau du code source. Ce niveau permet d'avoir accès à des informations de haut niveau, comme les types et les noms des variables.

3.2.1 Contre-mesures contre les attaques par canal auxiliaire

Luo et al. ont proposé une contre-mesure de dissimulation automatisée dont le principe est de mélanger les opérations C (*statements*) indépendantes [Luo et al., 2015]. L'outil associé prend du code C en entrée. Il regroupe les déclarations par groupe de déclarations indépendantes, et mélange chaque groupe à l'exécution. Il ajoute des déclarations factices lorsque trop peu de déclarations indépendantes ont été trouvées pour un groupe afin d'augmenter l'effet de mélange. En présence de boucles, les opérations internes à la boucles peuvent être mélangés. Cette approche suppose que le code est sous forme SSA.

Comme une opération C résulte souvent en plusieurs instructions assembleur, se placer au niveau code source permet d'obtenir une granularité d'un niveau équivalent à quelques instructions assembleur. L'indépendance des opérations est une propriété variable d'un algorithme à un autre, ou même au sein d'un algorithme, et l'idée de combler ce problème par ajout de déclarations factices est une idée importante pour l'applicabilité large de l'approche. Le fait d'être à haut niveau offre a priori plus d'opportunités de mélange

TABLE 3.1 – Vue générale des approches automatisées existantes pour les attaques par canaux auxiliaire et les attaques par injection de fautes, organisées par le niveau d'application de la contre-mesure.

		Principe de la contre-mesure	Contraintes
Niveau code source	[Luo et al., 2015]	multiversionnement statique (dissimulation)	code linéaire
	[Couroussé et al., 2016]	polymorphisme de code dynamique (dissimulation)	langage spécifique
	[Eldib et Wang, 2014]	masquage	flot de contrôle indépendant des données d'entrée
	[Lalande et al., 2014]	intégrité du flot de contrôle	-
	[Akkar et al., 2003]	intégrité du flot de contrôle	-
Niveau compilation	[Malagón et al., 2012]	multiversionnement statique (dissimulation)	-
	[Agosta et al., 2012]	polymorphisme de code dynamique (dissimulation)	-
	[Agosta et al., 2015b]	multiversionnement statique (dissimulation) et masquage partiel	-
	[Agosta et al., 2015a]	autre	-
	[Moss et al., 2012]	masquage	langage spécifique
	[Agosta et al., 2013a]	masquage	-
	[Bayrak et al., 2015b]	préchargement aléatoire et masquage	mesures à fournir (optionnel)
	[Luo et al., 2017]	implémentation à seuil (masquage)	-
	[Barry et al., 2016]	duplication d'instructions (tolérance aux fautes)	-
	[Reis et al., 2005]	duplication d'instructions (détection des fautes) et intégrité du flot de contrôle	-
	[Proy et al., 2017]	duplication d'instructions sur les sorties de boucles (détection des fautes)	-
	[Chen et al., 2017]	redondance d'instructions et des données (détection des fautes)	la plate-forme doit supporter les instructions SIMD
Niveau assembleur	[Bayrak et al., 2011]	préchargement aléatoire	mesures pour configurer la protection
	[Rauzy et al., 2016]	double rail et logique de préchargement	code d'entrée bitslicé
	[Moro et al., 2014]	duplication d'instructions (tolérance aux fautes)	-
	[De Keulenaer et al., 2016]	plusieurs contre-mesures de détection et de tolérance aux fautes	-

d'opérations par rapport à un bas niveau comme le niveau assembleur : au niveau source il n'y a pas de fausses dépendances telles que celles introduites par l'allocateur de registres.

Couroussé et al. ont proposé une approche pour déployer une contre-mesure de dissimulation basée sur la génération dynamique de codes polymorphes [Couroussé et al., 2016]. Leur approche nécessite l'utilisation d'un langage dédié (DSL). Le code écrit dans ce DSL sert à générer le code C d'un générateur de code polymorphe spécialisé. À l'exécution, le générateur génère régulièrement de nouvelles versions du code machine en utilisant des variantes sémantiques pour remplacer des instructions, en mélangeant des instructions, en changeant les registres alloués, et en insérant des instructions de bruit pour faire varier le code généré.

Cette approche reposant sur la génération dynamique de code, elle ne requiert pas de précaution particulière vis à vis du compilateur, puisque celui-ci optimise le générateur de code et non pas le code à générer. Les transformations de code utilisées correspondent à des transformations de code de niveau assembleur. Elles sont variées, ce qui peut apporter de la flexibilité à l'utilisateur. Cependant, l'utilisation d'un DSL dédié peut freiner l'adoption d'une telle approche.

Eldib et al. ont proposé une approche pour appliquer automatiquement un schéma de masquage, en utilisant un solveur SMT [Eldib et Wang, 2014]. Le programme source est parsé par `clang` pour être transformé en une représentation intermédiaire (IR) de LLVM. Le code au format IR est ensuite transformé en programme booléen. Ensuite, chaque opération du programme est masquée, soit directement s'il s'agit d'une opération linéaire, soit en déterminant avec un solveur SMT une séquence d'instructions masquées équivalente. Ensuite, le code sécurisé est émis sous forme de code C++ et compilé en `-O0`[1], pour éviter que le compilateur n'altère la contre-mesure.

Le masquage est une contre-mesure difficile à appliquer de manière générique, de part les modifications engendrées sur les calculs effectués. Ici, bien que l'approche émette du code source, le compilateur est utilisé pour transformer le code en IR de manière à ensuite obtenir un programme booléen. L'utilisation combinée du compilateur et d'un solveur SMT apporte une réponse pertinente au problème d'application automatisée du masquage, puisque le SMT se charge de trouver comment masquer les instructions pour lesquelles il n'y a pas de manière simple d'appliquer la contre-mesure. L'obligation de compiler le code sans optimisation résulte du fait que l'approche n'a pas été intégrée pleinement au sein du compilateur en adaptant les passes d'optimisations. Cela rend l'approche plus difficile à utiliser en pratique, puisque le code produit sera alors beaucoup plus lent et plus volumineux.

3.2.2 Contre-mesures contre les attaques en fautes

Lalande et al. ont proposé d'appliquer au niveau du code source une contre-mesure d'intégrité du flot de contrôle basée sur des compteurs et des variables supplémentaires

1. Cette information provient d'une discussion avec les auteurs

[Lalande et al., 2014]. Cette contre-mesure, vérifiée formellement, permet de sécuriser un code contre toute corruption s'assimilant à un saut de plus de deux lignes de code source. La contre-mesure est appliquée en 2 phases ; d'abord, toutes les vulnérabilités du code d'origine sont recherchées en simulant des détournements du flot de contrôle au niveau du code source, ce qui permet d'appliquer ensuite la contre-mesure uniquement sur les régions sensibles. Par construction, les attaques par saut de plus de deux instructions C sont systématiquement détectées. Cependant, les fautes ayant un impact plus petit, par exemple le saut d'une ligne ou d'une instruction assembleur, ne sont pas toujours détectées.

Ces résultats montrent le décalage qu'il peut y avoir entre un modèle de fautes de haut niveau et la réalité matérielle : le modèle de fautes au niveau code source ne capture pas toutes les fautes possibles, il n'y a pas toujours de correspondance entre un saut d'instruction à haut et bas niveau. Les instructions C résultant souvent en plusieurs instructions assembleur, il est logique que certaines attaques ne soient pas détectées si celles-ci ont une granularité plus fine que la contre-mesure. Enfin, cet effet est amplifié si le code est compilé sans optimisation, puisqu'on a alors plus d'instructions assembleur par déclaration C en moyenne. D'autre part, le flot de calcul étant indépendant des calculs de code insérés pour vérifier l'intégrité, il est nécessaire de compiler le code en `-O0` pour éviter que le compilateur ne puisse modifier trop fortement la contre-mesure.

Akkar et al. ont également présenté une application automatisée d'une contre-mesure d'intégrité de flot de contrôle [Akkar et al., 2003]. Le développeur doit au préalable annoter son code à l'aide de pragmas pour indiquer les zones à sécuriser. L'application est réalisée par un outil qui se présente sous la forme d'un préprocesseur.

Cette approche est proche de l'approche précédente [Lalande et al., 2014]. L'approche est configurable par le développeur qui peut définir précisément les zones d'application. Ici aussi, les auteurs constatent que 20% des fautes ne sont pas détectées. Les risques de l'approche sont identiques à ceux de l'approche précédente.

3.2.3 Discussion

Appliquer des contre-mesures au niveau du code source a l'avantage d'être compatible avec l'utilisation de compilateurs propriétaires, et même de permettre l'utilisation de plusieurs chaînes d'outils différentes sans aucun problème de compatibilité. Ce niveau d'application permet également d'être indépendant de l'architecture cible.

En outre, une quantité substantielle d'informations est disponible à ce niveau, telles que les informations de type des variables.

Toutefois, les contre-mesures peuvent être altérées par la compilation. Ce n'est pas toujours le cas, par exemple dans l'approche COGITO [Couroussé et al., 2016], la contre-mesure est appliquée à l'exécution par un générateur dédié et il n'y a donc aucun risque qu'elle soit altérée lors de la compilation. Les approches proposées par [Lalande et al., 2014] et [Eldib et Wang, 2014] nécessitent quant à elles de compiler les parties sécurisées sans optimisation du compilateur pour contourner ce problème. Cependant, comme déjà

TABLE 3.2 – Niveau d'application et passes modifiées dans le compilateur pour les approches de niveau compilation

Approche	Niveau d'application	Passes modifiées
[Malagón et al., 2012]	Middle-end	Passe de déroulage de boucle
[Agosta et al., 2012]	Non indiqué	-
[Agosta et al., 2015b]	Middle-end et back-end	Plusieurs passes dans le middle-end et le back-end
[Agosta et al., 2015a]	Middle-end et back-end	Sélection d'instructions
[Moss et al., 2012]	Middle-end	-
[Agosta et al., 2013a]	Middle-end	-
[Bayrak et al., 2015b]	Middle-end et back-end	-
[Eldib et Wang, 2014]	Middle-end	-
[Barry et al., 2016]	Back-end	Sélection d'instructions et allocation de registres
[Reis et al., 2005]	Non indiqué	-
[Proy et al., 2017]	Middle-end et back-end	Pliage des branches (Branch folding) et allocation de registres
[Chen et al., 2017]	Middle-end	-
[Luo et al., 2017]	Middle-end	-

évoqué, cela n'élimine pas complètement le risque : le code passe encore par la sélection des instructions, l'allocation des registres et l'ordonnancement des instructions (entre autres), chacune de ces passes pouvant modifier certaines contre-mesures. Ainsi, même en utilisant des techniques de contournement comme présenté en section 2.4.2, les développeurs devront vérifier pour chaque application automatiquement durcie au niveau du code source que les contre-mesures sont toujours présentes et correctes après compilation. Cela implique généralement de vérifier le code assembleur produit par le compilateur, ce qui est une tâche fastidieuse et sujette aux erreurs.

Enfin, le niveau code source est parfois de trop haut niveau pour que l'application de la contre-mesure s'adapte correctement les phénomènes matériels en jeu. Le cas des attaques en fautes illustre le problème de la sécurisation à un trop haut niveau d'abstraction de code ou de modèle de faute.

3.3 Application automatisée de contre-mesures dans le compilateur

Plusieurs approches ont proposé d'appliquer des contre-mesures durant la compilation.

Le tableau 3.2 résume, pour chaque approche, le niveau d'application à l'intérieur du compilateur et les passes qui ont été modifiées. Appliquer une contre-mesure au sein du compilateur permet de jouer sur plusieurs niveaux, puisque le middle-end fournit une représentation d'assez haut niveau, avec des informations de type, tandis que dans le back-end, le code est proche du code final émis, et s'en rapproche de plus en plus.

3.3.1 Contre-mesures contre les attaques par canal auxiliaire

Malagón et al. ont proposé de déployer une contre-mesure de dissimulation utilisant la génération statique de plusieurs variantes d'une fonction [Malagón et al., 2012]. Cette contre-mesure consiste à choisir au hasard entre différentes versions d'un même code à chaque exécution de la fonction. Le code source doit être annoté à l'aide de pragmas par le développeur pour indiquer les fonctions où des données sensibles sont manipulées. Le compilateur génère ensuite plusieurs versions différentes du code de la fonction en modifiant les configurations des optimisations, par exemple en utilisant la passe de déroulage de boucle. Il insère également le code qui est chargé de sélectionner aléatoirement au moment de l'exécution la version du code à exécuter.

Cette approche montre une utilisation originale du compilateur et des optimisations disponibles. Elle repose sur les capacités des optimisations du compilateur à modifier fortement le code, pour obtenir plusieurs versions ayant des comportements observables différents. Bien que les optimisations n'aient pas été conçues pour cet usage, cette approche illustre que des moyens de diversifications divers peuvent être utilisés pour augmenter la résistance du code face aux attaques par canal auxiliaire. En particulier, il est possible d'utiliser des transformations à granularité variable : fonction, boucle, bloc de base, instruction.

Agosta et al. ont proposé une contre-mesure de dissimulation basée sur la modification dynamique du code [Agosta et al., 2012]. Le code est modifié à l'exécution en utilisant des équivalences sémantiques au niveau des instructions, en rendant aléatoire les accès aux tables, et en mélangeant les instructions. La contre-mesure est automatiquement appliquée par un compilateur : des passes de transformation ont été ajoutées dans LLVM afin de préparer statiquement les transformations effectuées à l'exécution par un moteur de polymorphisme (fonction permettant de modifier le code à l'exécution).

Nous retrouvons dans cette approche une combinaison de plusieurs transformations visant à faire varier le code qui s'exécute. Le code est modifié à l'exécution, les performances sont donc fortement impactées par la modification du code. Les auteurs proposent de modifier le code après plusieurs exécutions afin d'amortir le coût de la modification. L'approche est étudiée sur une plate-forme qui, de part son degré de résistance aux attaques par canal auxiliaire, permet d'avoir une fréquence de modification assez faible. Sur une plate-forme plus vulnérable, le coup de la modification de code pour rendre le code suffisamment robuste pourrait toutefois devenir rédhibitoire.

Agosta et al. ont également proposé une contre-mesure de dissimulation basée sur la génération statique de plusieurs variantes [Agosta et al., 2015b]. Les auteurs proposent de générer automatiquement un code contenant plusieurs chemins d'exécution équivalents et comportant des instructions différentes, et de choisir aléatoirement entre les différents chemins à l'exécution. Cette approche intègre également certains éléments de masquage, puisque les accès aux tables constantes (comme la SBox dans le cas d'un AES) sont masqués. De plus, la sauvegarde du contenu des registres sur la pile (*spill*) est protégée. Un registre réservé pour contenir une valeur aléatoire sert à masquer toute valeur de registre stockée sur la pile. Lorsque le contenu d'un registre est restauré, il est démasqué pour récupérer sa valeur. Toutes ces transformations sont gérées par de nouvelles passes

de transformation dans LLVM. Certaines passes existantes ont également été modifiées : ces modifications visent notamment à faire en sorte qu'une instruction qui se trouve dans une zone à protéger ne puisse quitter cette zone. Le développeur doit fournir un fichier C annoté afin de spécifier les régions de code à protéger et les tables constantes à masquer. En plus du fichier source, le compilateur prend un fichier d'entrée qui spécifie les équivalences d'instructions à utiliser pour la génération de chemins équivalents.

Cette approche présente de nombreux intérêts. Tout d'abord, l'application de la contre-mesure est statique et permet d'éviter le problème d'avoir une zone mémoire disponible en écriture et en exécution simultanément pour modifier dynamiquement le code. D'autre part, l'approche intègre des éléments de masquage des accès mémoire, qui sont souvent des vecteurs de fuite d'information. L'approche combine donc un masquage partiel avec de la dissimulation. L'évaluation en sécurité, très poussée, montre que le niveau de sécurité atteint est très bon : aucune fuite d'information n'est détectée par t-test non spécifique (méthode d'évaluation présentée en section 2.5.2).

Agosta et al. ont également proposé une nouvelle contre-mesure contre les attaques par canal auxiliaire qui vise à faire ressortir plusieurs hypothèses de clés au lieu d'une seule pendant une attaque afin que l'attaquant ne sache pas quelle est la bonne hypothèse [Agosta et al., 2015a]. Cette contre-mesure est entièrement appliquée lors de la compilation, en plusieurs étapes. Plusieurs passes ont été ajoutées au middle-end et au back-end, la passe de sélection des instructions a également été modifiée. Le compilateur prend un fichier d'entrée annoté par le développeur qui spécifie les parties du code à protéger.

L'idée de cette contre-mesure est intéressante : nous cherchons habituellement à ne pas donner d'information à un attaquant, et ici plutôt que de brouiller l'information, ou de la subdiviser pour qu'elle soit plus dure à trouver, il s'agit de donner de l'information faussée à l'attaquant. Celui-ci ne peut alors pas distinguer le vrai du faux, ce qui complique l'attaque. Cette approche illustre la variété des contre-mesures pouvant être appliquées contre les attaques par canal auxiliaire, et en particulier à la compilation.

Moss et al. ont proposé d'appliquer automatiquement une contre-mesure de masquage booléen lors de la compilation [Moss et al., 2012]. Le développeur doit écrire son programme dans un langage dédié (DSL). Ce DSL permet d'exprimer avec des types prédéfinis le niveau de confidentialité des variables, en particulier il permet d'indiquer qu'une variable est secrète. Le compilateur utilise ensuite ces informations pour déterminer quelles valeurs intermédiaires doivent être masquées, et masque ces valeurs.

L'approche utilise du filtrage par motif (*pattern matching*) pour propager le niveau de confidentialité des variables. L'approche procède de manière itérative : à chaque itération le code est analysé jusqu'à trouver un problème de masquage, qui est corrigé en suivant une règle de correction. La contre-mesure est appliquée en suivant un modèle de fuite en valeur. Les performances mesurées s'approchent des performances obtenues pour un code masqué manuellement lorsque les boucles sont déroulées. L'approche permet donc d'appliquer efficacement la contre-mesure.

Agosta et al. ont également proposé une approche pour l'application d'une contre-mesure de masquage. Leur approche permet de générer un code masqué à l'ordre supérieur [Agosta et al., 2013a]. Le compilateur calcule pour chaque valeur dépendant de la clé le nombre de bits de la clé dont dépend la valeur. Cette analyse permet d'appliquer la contre-mesure uniquement aux valeurs intermédiaires qui dépendent d'un petit nombre de bits de clé, celles qui permettent à un attaquant d'effectuer une attaque en suivant le principe de diviser pour mieux régner. Ainsi, les valeurs intermédiaires dépendantes de tous les bits de la clé ne sont pas masquées.

Le compilateur, au delà d'être un outil d'application efficace, permet ici d'effectuer des analyses pour appliquer la contre-mesure de manière ciblée. Le compromis entre sécurité et performance est alors fortement amélioré. L'approche permet également une certaine configurabilité de la contre-mesure de masquage booléen, qui n'est que peu configurable autrement. Un utilisateur peut ainsi définir un seuil en terme de nombre de bits de la clé à partir duquel une variable n'est plus à masquer. Cet aspect rend la contre-mesure plus adaptable aux diverses contraintes rencontrées par l'utilisateur, comme des contraintes en termes de temps d'exécution.

Bayrak et al. ont également proposé une approche automatisée à la compilation permettant d'appliquer le masquage booléen à un programme [Bayrak et al., 2015b]. Une différence importante avec les autres approches est que le compilateur est utilisé pour décompiler un programme binaire et le représenter à un niveau plus élevé, puis pour recompiler le programme tout en appliquant la protection. Pour savoir où appliquer la contre-mesure, ils suggèrent de commencer par identifier les instructions qui peuvent révéler des données sensibles par un canal auxiliaire. Cette analyse s'effectue soit à l'aide de mesures fournies par l'utilisateur, soit par analyse statique. Par analyse statique, toutes les instructions manipulant des données dépendant de la clé sont jugées sensibles. Lorsque des mesures sont fournies, la mesure de sensibilité est effectuée en utilisant une métrique d'information mutuelle normalisée, et les instructions ayant une sensibilité supérieure à un seuil défini par l'utilisateur sont déclarées sensibles. La contre-mesure est ensuite appliquée à toutes les instructions sensibles. Ceci permet d'appliquer partiellement la contre-mesure et donc de réduire les surcoûts en performance. En outre, le compilateur peut également appliquer une contre-mesure de précharge aléatoire.

Cette approche se distingue par sa proposition de partir de mesures par canal auxiliaire pour appliquer la contre-mesure à des endroits ciblés. Elle effectue aussi un lien fort entre le comportement du matériel et l'application de la contre-mesure. C'est aussi une approche qui ne nécessite pas le code source mais uniquement le code binaire.

Luo et al. ont proposé une approche similaire pour générer automatiquement une implémentation par seuil (*threshold implementation*) sur la représentation intermédiaire de LLVM [Luo et al., 2017]. L'implémentation par seuil est une contre-mesure proche de la contre-mesure de masquage, puisque le secret est divisé en shares. Mais, dans l'implémentation par seuil, l'évaluation de chaque fonction est remplacée par l'évaluation de plusieurs fonctions, toutes indépendantes d'au moins un des shares, ce qui n'est pas le cas pour le masquage. Luo et al. utilisent un solveur SAT ainsi qu'une étape de transformation de code afin de trouver une solution appropriée.

L'usage combiné du compilateur et d'un solveur SAT permet ici d'obtenir une solution à la fois générique et efficace au problème d'application d'une contre-mesure d'implémentation par seuil. Une telle contre-mesure est moins sensible aux effets de la micro-architecture que la contre-mesure de masquage, mais elle est plus difficile à appliquer, l'apport du solveur SAT est donc important pour obtenir une approche générique.

3.3.2 Contre-mesures contre les attaques en faute

Barry et al. ont utilisé le compilateur pour appliquer automatiquement une contre-mesure de tolérance aux fautes [Barry et al., 2016] dont le principe est de dupliquer toutes les instructions assembleur pour tolérer le saut d'une instruction. À cette fin, plusieurs passes ont été ajoutées à LLVM et les passes de sélection des instructions et d'allocation des registres ont été modifiées pour générer des instructions moins coûteuses à dupliquer. Les passes d'optimisations classiques améliorent ensuite la performance du code produit. Les surcoûts obtenus sont inférieurs à ceux obtenus en appliquant cette même contre-mesure directement sur le code assembleur. Barry et al. ont de plus généralisé cette contre-mesure pour tolérer n fautes pouvant chacune sauter plusieurs instructions [Barry, 2017].

Cette approche montre l'intérêt de placer l'application de contre-mesure dans le compilateur. Ce niveau d'application offre une grande flexibilité puisque différentes étapes de la compilation, comme la sélection d'instructions et l'allocation de registres par exemple, peuvent être adaptées précisément pour une contre-mesure et permettre de réduire l'impact sur les performances.

Reis et al. ont proposé de déployer une contre-mesure de détection des fautes lors de la compilation [Reis et al., 2005]. Les instructions et leurs opérandes sont dupliqués afin que leurs résultats soient comparés pour détecter les fautes. En outre, des vérifications supplémentaires sont ajoutées pour s'assurer que le flot de contrôle n'est pas détourné. Les auteurs indiquent que l'approche pourrait facilement être étendue pour de la tolérance aux fautes.

L'application de la contre-mesure est ici aussi réfléchie de manière à limiter l'impact sur les performance. Les auteurs choisissent ainsi d'appliquer la duplication avant la passe d'ordonnancement d'instructions (*instruction scheduling*) et celle d'allocation de registres. Le compilateur peut alors choisir un ordre d'exécution des instructions permettant de masquer les latences de certaines instructions. Ces techniques permettent d'obtenir des surcoûts inférieurs à $\times 2$, malgré la duplication d'instructions. Toutefois, la cible est un processeur haute performance qui permet d'absorber en partie le coût des calculs redondants et conditionnels.

Proy et al. ont proposé d'utiliser le compilateur pour appliquer une contre-mesure de sécurisation des boucles contre les attaques par injection de fautes [Proy et al., 2017]. Les instructions impliquées dans le calcul des conditions de sortie des boucles sont dupliquées pour ajouter des blocs de contrôle chargés de détecter si la sortie est anticipée ou retardée. Cette transformation est appliquée au niveau IR. Les auteurs expliquent que certaines

passes du compilateur comme la sélection d'instructions et l'allocation de registres doivent être modifiées pour que la contre-mesure reste correctement appliquée jusqu'à ce que le code soit émis.

L'application ciblée d'une contre-mesure est intéressante en termes de performances : les surcoûts observés en temps d'exécution et en taille de code sont en moyenne inférieurs à 20%.

Enfin, Chen et al. ont proposé de réaliser de la redondance des opérations en utilisant les instructions SIMD [Chen et al., 2017]. Leur compilateur vectorise certaines instructions afin d'obtenir de la redondance des données et des opérations, et ajoute des codes de vérification d'erreurs. Toutes les transformations de code sont effectuées au niveau IR et l'approche est indépendante de l'architecture. Il faut simplement que l'architecture cible prenne en charge les instructions SIMD. L'utilisation des instructions SIMD permet d'obtenir un coût en performance inférieur à celui des approches classiques de duplication d'instructions.

L'approche tire partie de ressources matériels non exploitées. Elle montre que certaines capacités du processeur offrent une belle opportunité d'optimisation pour améliorer les compromis entre sécurité et performance. En particulier, le coût en terme de taille de code se trouve ici fortement réduit.

3.3.3 Discussion

Le niveau du compilateur est intéressant si plusieurs langages sources doivent être supportés, car le front-end supporte généralement plusieurs langages.

L'application de contre-mesure à la compilation requiert de choisir comment intercaler l'application de la contre-mesure avec les différentes passes d'optimisations. Appliquer la contre-mesure au niveau IR permet de supporter plus facilement plusieurs architectures cibles, et d'avoir des informations de haut niveau comme les types des variables. Appliquer la contre-mesure dans un back-end permet d'être plus proche du code émis, et donc de réduire le nombre de transformations effectuées entre l'application de la contre-mesure et l'émission du code. De manière générale, l'application de la contre-mesure devrait être effectuée après les passes de transformation pouvant mettre à mal la contre-mesure, et avant les passes permettant d'optimiser la performance du code sans menacer la contre-mesure. Il n'est pas toujours faisable de remplir cette condition. Les passes situées après la passe d'application de la contre-mesure doivent donc être adaptées ou désactivées pour prendre en compte au mieux la contre-mesure [Proy et al., 2017]. D'autre part, les passes en amont de l'application de la contre-mesure peuvent être adaptées pour faciliter l'application de la contre-mesure [Barry et al., 2016].

L'effort d'ingénierie déployé pour mettre en œuvre de telles approches est important, néanmoins, le contrôle offert par ce niveau d'application permet d'obtenir de bonnes performances, et d'adapter les passes de transformation pour mieux s'adapter à une contre-mesure. Cependant, il reste encore nécessaire de vérifier le code assembleur généré

pour être sûr que les passes en aval de la passe d'application de la contre-mesure ne l'ont pas affectée.

Enfin, ce niveau d'application nécessite d'avoir accès au code source du compilateur. Si le développeur utilise un compilateur sans avoir accès à son code source, il devra utiliser un autre compilateur dont code source est ouvert pour désassembler un fichier, reconstruire une représentation intermédiaire, appliquer la contre-mesure au code et le recompiler, un processus qui est difficile.

3.4 Application automatisée de contre-mesures à l'édition des liens ou au niveau assembleur

Cette section présente les approches qui appliquent des contre-mesures directement sur un code assembleur, pendant ou avant la phase d'édition des liens.

3.4.1 Contre-mesures contre les attaques par canal auxiliaire

Bayrak et al. ont proposé d'appliquer automatiquement une contre-mesure de préchargement aléatoire au niveau de l'assembleur [Bayrak et al., 2011]. L'application de cette contre-mesure est tout à fait naturelle à ce niveau, puisque l'allocation des registres a déjà été effectuée. Des mesures empiriques effectuées sur des codes non sécurisés sont utilisées pour déterminer les instructions à sécuriser.

De la même manière que pour leur approche à la compilation [Bayrak et al., 2015b], la force de cette approche repose sur le lien effectué entre les mesures du comportement du processeur et l'application de la contre-mesure. Une telle correspondance ne peut se faire qu'à des niveaux d'application suffisamment bas, comme dans le back-end du compilateur ou ici sur un code assembleur.

Rauzy et al. ont également mis en place une contre-mesure contre les attaques par canal auxiliaire au niveau de l'assembleur : le double rail avec logique de préchargement [Rauzy et al., 2016]. Leur approche exige que le code ait été préalablement bitslicé. Leur approche permet également de prouver l'absence de fuite en consommation sur le code assembleur obtenu et de prouver que ce code reste sémantiquement correct.

L'aspect preuve de cette méthode est pertinent : la sémantique du programme doit être conservée. La contrainte imposée d'avoir un code bitslicé fait perdre un des avantages de ce niveau, puisqu'un programme binaire propriétaire ne pourra pas être sécurisé.

3.4.2 Contre-mesures contre les attaques en fautes

Moro et al. ont proposé une contre-mesure reposant sur la duplication des instructions pour obtenir de la tolérance aux fautes [Moro, 2014]. Cette contre-mesure est destinée

à tolérer le saut d'une instruction. Pour cela, chaque instruction est remplacée par une séquence d'instructions idempotentes qui est sémantiquement équivalente à l'instruction originale et qui, une fois dupliquée, permet de tolérer un saut d'instruction. Comme cette contre-mesure nécessite des registres supplémentaires, il est parfois nécessaire de sauvegarder certains registres sur la pile. De plus, certaines instructions (par exemple les lectures mémoire volatiles) ne peuvent pas être remplacées par une séquence tolérante aux fautes. Il s'agit de la contre-mesure automatisée par [Barry et al., 2016] au niveau de la compilation.

Le niveau d'application sur le code assembleur est particulièrement adapté à la contre-mesure. Bien que le niveau d'application soit adapté, le fait que la compilation du programme se fasse sans prendre en compte la contre-mesure complique son application : il faut refaire l'analyse de vivacité des registres.

De Keulenaer et al. ont montré comment déployer automatiquement diverses contre-mesures contre les attaques de fautes au niveau binaire en utilisant de la réécriture à l'édition des liens [De Keulenaer et al., 2016]. Leur outil, basé sur Diablo [Van Put et al., 2005], combine à la fois des contre-mesures de tolérance aux fautes et des contre-mesures de détection des fautes : duplication des sauts conditionnels, intégrité du graphe d'appel, vérification des écritures mémoire, duplication des compteurs de boucle.

Cette approche se rapproche en partie des approches de niveau compilation puisque dans les deux cas la contre-mesure est appliquée au sein d'un outil d'optimisation qui repose sur des analyses de codes proches de celles effectuées par le compilateur. La variété des contre-mesures appliquées montre que même à bas niveau il est possible d'appliquer diverses contre-mesures.

3.4.3 Discussion

Ce niveau est surtout utilisé pour appliquer des contre-mesures demandant des analyses ou transformations de niveau assembleur. L'application de contre-mesures nécessite cependant de reconstruire des informations qui étaient disponibles lors du processus de compilation. Par exemple, les contre-mesures nécessitent souvent l'utilisation de registres supplémentaires, ce qui nécessite soit de sauvegarder des registres sur la pile, soit de faire une réallocation complète des registres, et dans tous les cas de faire une analyse de vivacité.

Ainsi, lors du développement d'une approche automatique à ce niveau, un effort d'ingénierie important est nécessaire pour réobtenir des informations disponibles à la compilation, ou pour refaire des traitements qui avaient été effectués par le compilateur d'une manière non optimale par rapport à la contre-mesure à appliquer.

Toutefois, l'application de contre-mesures à ce niveau évite d'avoir à vérifier manuellement si la contre-mesure est toujours présente dans le code final, puisque le processus de compilation a lieu entièrement avant l'application des contre-mesures. Cela permet l'utilisation d'un tel outil par un développeur non expert en sécurité. De plus, ce niveau d'application permet d'être indépendant du langage du code source, ce qui est intéressant

si plusieurs langages sources doivent être pris en charge. Il permet de sécuriser le code après les optimisations d'édition des liens, et de sécuriser potentiellement les bibliothèques binaires : l'accès au code source d'une application n'est pas nécessaire pour la sécuriser. Ainsi, ce niveau d'application est compatible avec l'utilisation d'outils propriétaires au code source fermé.

3.5 Discussion

Dans cette section, nous comparons les niveaux susmentionnés d'application automatique des contre-mesures du point de vue de l'effort de développement d'un outil et des conséquences pour l'utilisation.

Le premier aspect à considérer est le temps nécessaire pour développer un outil automatisé. Cet aspect dépend de la contre-mesure à appliquer. Une contre-mesure de masquage est plus facile à appliquer au niveau du code source qu'au niveau de l'assembleur car elle nécessite une modification importante des calculs effectués. Le compilateur est un endroit où différentes contre-mesures peuvent être appliquées, car pendant la compilation, le compilateur manipule à la fois des représentations de haut niveau (par exemple des variables typées) et des représentations de bas niveau (par exemple des instructions en pseudo assembleur).

Le développement d'un outil automatisé implique de parser et d'émettre le code dans les langages ciblés. Les compilateurs ont déjà l'infrastructure nécessaire pour cela (parseur et/ou émetteur de code), et généralement le développeur n'a qu'à ajouter le support d'un pragma pour délimiter les zones de code à sécuriser. Pour les approches de niveaux code source et assembleur, les développeurs doivent souvent implémenter ou réutiliser un parseur et/ou un émetteur de code pour les langages ciblés.

Les coûts d'ingénierie liés à l'utilisation des outils doivent également être pris en compte. Comme ces outils sont automatiques, le coût de génération du code sécurisé est proche de zéro, mais le développement des outils demande beaucoup de travail. Lorsque l'outil applique la contre-mesure au niveau du code source, la revue de code est facilitée, mais l'utilisateur doit vérifier que la contre-mesure est toujours correcte au niveau assembleur. Cette tâche fastidieuse est l'un des principaux inconvénients de l'approche au niveau du code source. L'approche de niveau assembleur ne souffre pas de cet inconvénient : l'application d'une contre-mesure au niveau assembleur évite toute altération lors de la compilation. L'application d'une contre-mesure lors de la compilation permet parfois de vérifier que la contre-mesure est toujours valide juste avant l'émission du code assembleur/binaire si le développeur parvient à propager les informations nécessaires au sein du compilateur. Si la vérification de la contre-mesure avant l'émission du code n'est pas possible, le code assembleur devra être vérifié manuellement. Des travaux de recherche sur la vérification des contre-mesures sont cependant en cours [Ben El Ouahma et al., 2017, Bréjon et al., 2019].

Lorsqu'une contre-mesure est appliquée dans le compilateur, elle peut bénéficier d'optimisations qui devraient être réimplémentées si la contre-mesure était appliquée en dehors du processus de compilation. Plusieurs approches qui utilisent des compilateurs modifient

certaines passes du compilateur pour réduire le coût des contre-mesures. Les passes qui appliquent la contre-mesure peuvent être finement entrelacées avec les passes des compilateurs pour profiter de ces passes sans risquer que la contre-mesure soit altérée par des optimisations [Barry et al., 2016]. Aux autres niveaux, il peut être plus difficile d'optimiser les transformations pour la performance. Par exemple, au niveau de l'assembleur, il peut être nécessaire de sauvegarder des registres sur la pile voire de recommencer l'allocation des registres pour pouvoir utiliser des registres pour la contre-mesure. À titre de comparaison, [Moro et al., 2014] et [Barry et al., 2016] ont mis en œuvre la même contre-mesure au niveau assembleur et compilateur respectivement, et [Barry et al., 2016] ont obtenu des surcoûts en temps d'exécution et en taille inférieurs à ceux de [Moro et al., 2014]. L'utilisation du compilateur a également permis de généraliser la protection pour supporter différentes largeur de fautes et nombre de fautes [Barry, 2017].

Il en ressort que le niveau du compilateur parait le plus adapté en général pour l'application automatisée de contre-mesure.

3.6 Conclusion et positionnement de la thèse par rapport à l'état de l'art

Ce chapitre montre que de nombreux travaux ont proposé d'automatiser l'application de contre-mesures. Nous avons montré, au travers de leur synthèse, les avantages et inconvénients des différents niveaux possibles et mis en évidence les nombreux avantages de l'application de contre-mesures à la compilation.

Bien que différents travaux aient déjà proposé des approches d'insertion de contre-mesures contre les attaques par canal auxiliaire à la compilation, nous pensons qu'il est nécessaire d'étudier encore ce sujet et de proposer des solutions automatisées visant des systèmes contraints et dépourvus de sécurisation contre les attaques par canal auxiliaire. L'objectif est alors de permettre à ces systèmes d'acquérir un premier niveau de sécurisation.

En cohérence avec cet objectif, nous considérons pour nos évaluations un attaquant capable d'effectuer uniquement des attaques simples comme une CPA, et disposant de peu d'expertise pour monter des attaques plus compliquées (attaques d'ordre supérieur, attaques par template, attaques par apprentissage machine, attaques avec prétraitement des traces...).

De plus, nous voulons pouvoir appliquer au sein d'un même flot des protections de masquage ou de dissimulation. Nous choisissons une contre-mesure de masquage booléen d'ordre 1 et une contre-mesure de polymorphisme de code. Alors qu'une contre-mesure de masquage induit des modifications importantes des calculs effectués, une contre-mesure de polymorphisme de code peut s'appuyer sur des transformations de bas niveau. Le niveau de compilation nous parait le plus flexible pour satisfaire ces deux usages, en plus des avantages précédemment mentionnés.

Ensuite, nous explorons des aspects non considérés jusqu'ici dans les approches automatisées de l'état de l'art, dans l'objectif d'améliorer les compromis performance / sécurité. D'une part, pour l'application d'une contre-mesure de masquage, nous avons dé-

cidé de suivre une approche par interpolation polynomiale pour le masquage des tables constantes, et proposons de nouvelles optimisations dans le but d'améliorer la performance. D'autre part, pour l'application du polymorphisme de code, nous apportons des réponses aux problèmes d'allocation mémoire et de gestion des droits pour permettre la génération dynamique de code, et nous étendons les transformations de code utilisées afin de permettre de meilleurs compromis.

Nos deux approches sont présentées dans les deux chapitres suivants : le chapitre 4 présente l'application automatisée d'une contre-mesure de masquage booléen d'ordre 1 et le chapitre 5 présente l'application automatisée d'une contre-mesure de polymorphisme de code.

Chapitre 4

Application automatique de masquage à la compilation

Sommaire

4.1 Introduction

Ce chapitre traite du problème de l'application automatisée d'une contre-mesure de masquage à la compilation. Cette technique de protection contre les attaques par canal auxiliaire est probablement la plus répandue [Belaïd, 2015]. Comme déjà évoqué en section 2.4.2, l'application du masquage est en pratique effectuée manuellement par des experts sur du code assembleur, notamment pour éviter les risques liés à la compilation [Balasch et al., 2015].

Le processus d'application de la contre-mesure est donc coûteux, puisqu'il demande à la fois du temps de développement et de l'expertise.

Nous proposons dans ce chapitre un algorithme pour automatiser l'application d'une contre-mesure de masquage au sein d'un compilateur, ainsi que des optimisations de performance. Nous implémentons cet algorithme dans LLVM, puis nous l'évaluons.

Ce chapitre commence par présenter le type de contre-mesure choisi et les hypothèses associées (section 4.2). Une fois la contre-mesure définie, un exemple préliminaire sera présenté (section 4.4), puis le principe général d'application de la contre-mesure sera détaillé (section 4.5), ainsi que les optimisations en performance que nous avons déployées (section 4.5.7.6). La suite du chapitre présentera la mise en œuvre de ce principe dans un compilateur (section 4.6), ainsi que l'évaluation en sécurité et en performance de la contre-mesure ainsi appliquée (section 4.7). La fin du chapitre développe ensuite une discute des problèmes mis en évidence (section 4.9) et une conclusion (section 4.10).

4.2 Choix pour notre approche

Les principes de masquage ont été présentés en section 2.3.2.

Le masquage booléen est le plus couramment utilisé pour AES, qui comporte beaucoup de OU EXCLUSIFS. C'est ce schéma qui est utilisé dans les approches d'application de masquage automatisée par la compilation [Moss et al., 2012, Agosta et al., 2013a, Bayrak et al., 2015b]. Cependant, aucune de ces approches n'a considéré l'utilisation d'approche par interpolation polynomiale pour le masquage des SBoxes, malgré les avantages d'une telle approche (déjà évoqués en section 2.3.2), qui est utilisable quel que soit l'ordre de masquage, fournit un coût en performance intéressant, et évite le problème du remasquage de la table.

Nous choisissons d'appliquer également un masquage booléen à la compilation, mais d'utiliser la méthode proposée dans [Coron et al., 2014c] pour le masquage des SBoxes par interpolation polynomiale. Les algorithmes proposés pourraient être étendus facilement pour du masquage arithmétique ou multiplicatif, ou pour gérer les changements de type de masquage. Nous nous sommes restreins au modèle de fuite par valeur, qui est un modèle couramment utilisé pour l'application du masquage et qui nous permet de simplifier l'application de la contre-mesure. Un modèle de fuite en transition pourrait être supporté en combinant ces travaux avec l'approche proposée dans [Wang et al., 2019]. Enfin, nous décrivons ici une implémentation de masquage à l'ordre 1. Nous en évaluons les

surcoûts en performance. Dans l'objectif de fournir des surcoûts en performance adaptés aux systèmes embarqués, nous proposons plusieurs optimisations pour réduire le coût de l'évaluation des polynômes interpolateurs de SBox.

4.3 Prérequis

Cette section présente la notion de fonction linéaire dans le cadre du masquage.

Définition 4. On dit qu'une fonction f est linéaire par rapport à un opérateur $\oplus$ si elle vérifie la condition suivante : $\forall a, b \in F_{2^n}, f(a \oplus b) = f(a) \oplus f(b)$.

Ici, le OU booléen entre deux variables masquées n'est pas linéaire par rapport au OU EXCLUSIF, tandis que le ET booléen avec une variable publique l'est. Plus généralement, les fonctions suivantes sont linéaires par rapport au OU EXCLUSIF :

- la multiplication $\otimes$ par un élément : $\forall a, b,c \in F_{2^n}, (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
- l'élévation au carré, et par induction l'élévation une puissance qui est elle même une puissance de 2 :
  $\forall a, b \in F_{2^n}, (a \oplus b)^2 = (a \oplus b) \otimes (a \oplus b) = a^2 \oplus a \otimes b \oplus b \otimes a \oplus b^2 = a^2 \oplus ((a \otimes b) \oplus (a \otimes b)) \oplus b^2 = a^2 \oplus b^2$

Par contre, on n'a pas : $\forall a, b,c,d \in F_{2^n}, (a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (b \otimes d)$. Pour cette raison, les multiplications sous cette forme sont appelées multiplications non linéaires dans la suite de ce manuscrit.

Le masquage des fonctions linéaires est simple puisqu'il consiste à effectuer le même traitement sur chacun des 2 shares. Ainsi, au lieu de faire $f(secret)$, on calcule $f(share_0)$ et $f(share_1)$, car $f(secret) = f(share_0 \oplus share_1) = f(share_0) \oplus f(share_1)$.

Le masquage des fonctions non linéaires demande de trouver une manière de calculer 2 nouveaux shares res_0 et res_1 tels que $f(share_0 \oplus share_1) = res_0 \oplus res_1$ sans jamais révéler d'information sur le secret.

Des algorithmes existent dans l'état de l'art pour certains cas particuliers comme le masquage des multiplications non linéaires de corps finis binaires [Rivain et Prouff, 2010] (l'algorithme en question est dénommé *secMult*), ou comme le OU booléen [Baek et Noh, 2005].

4.4 Exemple préliminaire

Cette section déroule un exemple d'application du masquage avec un modèle de fuite en valeur sur un exemple simple.

Le listing 4.1 présente la fonction que nous souhaitons masquer. Cette fonction prend en arguments deux variables non secrètes (nous utiliserons par la suite le terme *publique*) `a` et `b` et une variable secrète `s`. Cette fonction commence par un simple OU EXCLUSIF entre une variable secrète et une variable publique (ligne 4 du listing 4.1). Pour cette opération, le masquage ne coûte aucune opération supplémentaire, il suffit d'effectuer

l'opération sur l'un des deux shares (ligne 4 du listing 4.2) Ensuite, la fonction comporte un OU booléen entre deux variables masquées (ligne 5 du listing 4.1), et un ET booléen entre une variable masquée et une variable publique (ligne 6 du listing 4.1)

Pour masquer le ET booléen du listing 4.1 entre une variable publique et une variable secrète, qui est linéaire par rapport au OU EXCLUSIF, on applique le ET avec la variable publique sur chacun des deux shares (lignes 12 et 13 du listing 4.2).

Pour cet exemple, nous appliquons l'algorithme de [Baek et Noh, 2005] pour le OU booléen, et nous obtenons ainsi les lignes 9 et 10 du listing 4.2.

Cependant, la fonction du listing 4.2 n'est pas correctement masquée. Pour s'en rendre compte, plaçons nous dans le cas $a = 0$ et $s0 = s \oplus r$ et $s1 = r$, où r est un masque. Analysons par exemple les valeurs que prend $c0$ & $s1$ (ligne 9 du listing 4.2). On a : $c0$ & $s1 = (a \hat{} s0)$ & $s1 = s0$ & $s1$. La table de vérité de $c0$ & $s1$ en fonction de s et r est présentée en table 4.1. Nous pouvons constater que $(c0$ & $s1 = 1) \Rightarrow (s = 0)$. Ainsi, la variable intermédiaire $c0$ & $s1$ n'est pas statistiquement indépendante du secret, on dit que le masquage n'est pas *parfait* [Eldib et Wang, 2014].

TABLE 4.1 – Table de vérité de $c0$ & $s1$ en fonction de s et r, lorsque $a = 0$

s	$s1 = r$	$s0 = s \hat{} r$	$c0$ & $s1 = s0$ & $s1$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Ici, le problème vient du fait que les variables c et s ont été masquées en utilisant le même masque. Il faut donc remasquer l'une des deux variables avec un nouveau masque avant de les combiner. Le listing 4.3 présente le code obtenu en ajoutant un remasquage. Un nouveau masque est tout d'abord choisi (appel à `rand` de la ligne 9 du listing 4.3). Ensuite, les deux shares de l'une des deux variables sont mis à jour avec ce nouveau masque (lignes 10 et 11 du listing 4.3) pour conserver la propriété $c = c0 \oplus c1$.

Cet exemple montre que l'on peut appliquer cette contre-mesure de masquage en déterminant les opérations qui dépendent d'un secret puis en les masquant une à une, et en remasquant si nécessaire. Ce principe d'application de la contre-mesure peut être mis en œuvre automatiquement au sein du compilateur.

4.5 Application itérative du masquage sur une forme SSA

Cette section présente notre principe d'application de la contre-mesure de masquage, qui permet d'appliquer itérativement la contre-mesure sur une fonction sous forme SSA.

Listing 4.1 – Fonction à masquer, écrite en C

```
1 /* a et b sont publiques, et s est secret */
2 bool f(bool a, b, bool s)
3 {
4         bool c = a ^ s; /* OU EXCLUSIF par une variable publique */
5         bool d = c | s; /* OU entre deux secret : non lineaire */
6         bool e = d & b; /* ET entre une variable publique et une variable
              secrete : lineaire */
7         return e;
8 }
```

Listing 4.2 – Application d'un schéma de masquage booléen d'ordre 1 à la fonction du listing 4.1 sans prendre en compte le besoin de remasquer

```
1  /* a et b sont publiques, et s0 et s1 sont deux shares */
2  bool f(bool a, b, bool s0,s1)
3  {
4         bool c0 = a ^ s0;
5         bool c1 = s1;
6         /* c0 ⊕ c1 = a ⊕ s0 ⊕ s1
7            = a ⊕ s = c */
8         /* masquage du OU avec la formule de [Baek et Noh, 2005] */
9         bool d0 = (c0 | s0) ^ (c0 & s1); /* fuite d'ordre 1 */
10        bool d1 = (c1 & s1) ^ (c1 | s0);
11        /* d0 ⊕ d1 = d */
12        bool e0 = d0 & b;
13        bool e1 = d1 & b;
14        /* e0 ⊕ e1 = (d0 ∧ b) ⊕ (d1 ∧ b)
15           = (d0 ⊕ d1) ∧ b */
16        return e0 ^ e1;
17 }
```

Listing 4.3 – Application d'un schéma de masquage booléen d'ordre 1 à la fonction du listing 4.1 en prennant en compte le besoin de remasquer

```
1  /* a et b sont publiques, et s0 et s1 sont deux shares */
2  bool f(bool a, b, bool s0,s1)
3  {
4          bool c0 = a ^ s0;
5          bool c1 = s1;
6          /* c0 ⊕ c1 = a ⊕ s0 ⊕ s1
7             = a ⊕ s = c */
8          /* remasquage de c0 et c1 */
9          bool rand = rand();
10         c0 = c0 ^ rand;
11         c1 = c1 ^ rand;
12         /* masquage du OU avec la formule de [Baek et Noh, 2005] */
13         bool d0 = (c0 | s0) ^ (c0 & s1);
14         bool d1 = (c1 & s1) ^ (c1 | s0);
15         /* d0 ⊕ d1 = d */
16         bool e0 = d0 & b;
17         bool e1 = d1 & b;
18         /* e0 ⊕ e1 = (d0 ∧ b) ⊕ (d1 ∧ b)
19            = (d0 ⊕ d1) ∧ b */
20         return e0 ^ e1;
21 }
```

4.5.1 Algorithme général pour une fonction

L'algorithme 1 présente une vue générale du principe d'application d'un schéma de masquage. Celle-ci se déroule en 3 grandes étapes : retrouver les variables indiquées comme secrètes par l'utilisateur (`getUserSpecifiedSecretVariables(`f`)`), propager l'information de confidentialité sur les variables pour obtenir la liste des instructions à transformer (`getSecretInstsLists(`$ListOfSecretVars$`,` f`)`), puis transformer ces instructions pour obtenir un code masqué (`TransformInst1SecretOperand` et `TransformInst2SecretOperands`).

Nous détaillons dans les sections suivantes les étapes du processus d'application du masquage : la recherche des variables indiquées comme secrètes est présentée en section 4.5.2, la propagation de l'information de confidentialité est présentée dans la section 4.5.3, et la transformation des instructions est présentée dans les sections 4.5.4 et 4.5.5. Nous considérons pour ces étapes que l'utilisateur utilise le premier share à la place de la variable secrète lors de ses appels de la fonction. Typiquement, si une instruction utilisait le secret s, elle utilisera à la place le share s_0.

4.5.2 Recherche des variables spécifiées comme secrètes

La première étape de l'application de la contre-mesure consiste à retrouver dans la fonction sous forme SSA les variables ou emplacements mémoire contenant directement ou

Algorithme 1 : Masquage d'une fonction en place

```
f ← processFunction(f)
    Input : Function f
    Output : Masked function f
1   ListOfSecretVars = getUserSpecifiedSecretVariables(f)
2   ListOfInstsToTransform = getSecretInstsList(ListOfSecretVars, f)
3   foreach instruction I of ListOfInstsToTransform do
4       nbSecretOperands = getNbSecretOperands(I)
5       switch nbSecretOperands do
6           case 1
7               TransformInst1SecretOperand(I, f)
8           case 2
9               TransformInst2SecretOperands(I, f)
```

indirectement l'un des secrets spécifiés par l'utilisateur. Elle correspond à l'appel à l'appel à `getUserSpecifiedSecretVariables(`f`)` de l'algorithme 1. Pour cela, l'algorithme itère sur tous les opérandes de toutes les instructions ainsi que sur les arguments de la fonction en testant pour chaque variable si elle correspond à l'une des variables spécifiées par l'utilisateur.

4.5.3 Constitution de la liste des instructions à transformer

Une fois la correspondance effectuée entre les variables indiquées par l'utilisateur et les variables visibles sous forme SSA, il s'agit de déterminer l'ensemble des instructions à transformer pour obtenir un code masqué. C'est la charge de la fonction `getSecretInstsLists(SecretVars)`, dont le fonctionnement est présenté dans l'algorithme 2. Cet algorithme effectue une analyse visant à retrouver toutes les instructions dépendant d'une des variables secrètes.

Cette analyse repose sur un système d'annotations pour différencier les variables selon leur niveau de confidentialité. Ce système d'annotations permet qu'une variable secrète sauvegardée en mémoire puis rechargée dans une autre variable reste considérée comme secrète.

Les variables sont annotées avec un niveau de confidentialité pouvant être l'un des niveaux suivants :

- P ou publique - la variable est publique, et n'est pas séparée en shares
- S ou secrète - la variable ne doit pas être divulguée, et elle est ou devra être séparée en shares
- AS ou adresse d'un secret - la variable elle même n'est pas sensible, mais elle permet d'accéder à une variable secrète stockée en mémoire.

Pour les variables de niveau AS, on note dans le type le nombre de déréférencements nécessaires pour récupérer un secret à partir de la variable. Ainsi, une variable de niveau AS_1 est un pointeur vers un secret. Une variable de niveau AS_2 est un pointeur vers une

Algorithme 2 : Analyse de dépendances pour trouver toutes les instructions à transformer

```
ListOfInstsToTransform ← getSecretInstsList(ListOfSecretVars, f)
    Input : User specified secret variables list ListOfSecretVars, Function f
    Output : Instruction list ListOfInstsToTransform
1   ListOfInstsToTransform = List()
2   Confidentiality = Map() // Map variables and insts to their confidentiality
    // Infer confidentiality of the secret variables specified by the user
3   foreach variable v of ListOfSecretVars do
4       Confidentiality[v] = inferConfidentialityFromType(v)
5   ListOfInstsWithUnknownConfidentiality=getInstructions(f)
6   while isNotEmpty(ListOfInstsWithUnknownConfidentiality) do
7       I = getFirstInst(ListOfInstsWithUnknownConfidentiality)
8       removeFromList(ListOfInstsWithUnknownConfidentiality, I)
        // Propagate confidentiality and put results in Confidentiality
9       success = propagateConfidentiality(I, Confidentiality)
10      if not(success) then
            // An operand has an unknown confidentiality
            // Put back the instruction in the list to try again later
11          append(ListOfInstsWithUnknownConfidentiality, I)
12      else
13          if instructionManipulatesSecretData(I) then
14              append(ListOfInstsToTransform, I)
15          else
                // Nothing to do
16  return ListOfInstsToTransform
```

variable de type AS_1. Les niveaux AS_0 et S sont équivalents et désignent tous deux un secret.

Les variables spécifiées comme secrètes par l'utilisateur sont tout d'abord annotées. Chacune de ces variables est annotée par le niveau S si elle est de type entier, et est annotée par le niveau AS si elle correspond à une adresse d'un entier. Nous supposons ici que les secrets sont des nombres entiers, ce qui est le cas en ce qui concerne les fonctions de chiffrement visées, mais une option pourrait être ajoutée pour que l'utilisateur indique lui même le type de chaque variable en entrée de la fonction.

Une fois que les variables spécifiées comme secrètes par l'utilisateur ont été annotées, une analyse de flot de données est utilisée pour inférer le niveau de confidentialité de toutes les autres variables de la fonction. Pour cela, l'analyse de flot de données utilise les règles de propagation suivantes :

— Le résultat de toute opération de calcul utilisant un opérande de type AS_t est de type AS_t.
— L'adresse mémoire utilisée pour stocker en mémoire un élément de type AS_t est de type AS_{t+1}.

— Le résultat d'une lecture mémoire dont l'adresse est de type AS_t est de type $AS_{max\{t-1;0\}}$. Ainsi, le résultat d'une lecture mémoire indexée par un secret est lui aussi secret.
— Toute variable non prise en compte par les règles précédentes est de type P.

Cette propagation a lieu dans l'appel à la fonction `propagateConfidentiality` de l'algorithme 2. Cette fonction renvoie un booléen de succès, qui est faux si l'un des opérandes de l'instruction nécessaire à la propagation est de confidentialité inconnue, et qui est vrai si la propagation est effectuée avec succès.

Toutes les instructions manipulant un opérande de type S ou ayant un résultat de type S sont ensuite ajoutées à la liste des instructions à masquer.

Bien que le typage des adresses mémoire permette de propager un type secret à travers les accès mémoire, l'analyse reste imprécise et demande qu'une adresse ne serve pas à calculer à la fois un pointeur vers des secrets et un pointeur vers des variables non confidentielles.

De plus, ces règles de propagations sont syntaxiques, ainsi, le résultat d'un OU EXCLUSIF entre un secret et lui même est considéré comme de type S, bien que ce résultat soit constant égal à 0.

4.5.4 Transformation d'une instruction ayant un seul opérande secret

Cette section présente la transformation des instructions ayant un seul opérande secret, correspondant à l'appel à la fonction `TransformInst1SecretOperand(`I`,`f`)` de l'algorithme 1. L'algorithme 3 présente cette transformation. Les fonctions préfixées par `create` insèrent la nouvelle instruction après la dernière instruction insérée, ou à défaut, après l'instruction I à transformer. De plus, ces opérations renvoient la variable résultat de la nouvelle instruction, qui peut ainsi éventuellement être utilisée pour des instructions subséquentes.

Cette étape est effectuée en place dans f. Elle exploite le fait que l'utilisateur remplace dans son code chaque variable secrète par le premier share associé à cette variable. Ainsi, une instruction qui manipulait un secret manipule alors le premier share. L'instruction est donc laissée telle quelle lors de la transformation si elle utile au calcul masqué, ou supprimée si elle ne l'est pas.

Pour les instructions ayant un seul opérande secret, l'algorithme applique des méthodes de transformation pour chaque type d'instructions.

L'algorithme gère tout d'abord les opérations linéaires par rapport au OU EXCLUSIF : les décalages, rotations, extensions signées ou non signées, troncatures, et le ET booléen. Le ET booléen est ici une opération linéaire par rapport au OU EXCLUSIF parce qu'il s'agit d'un ET booléen ayant un seul opérande secret : $a \wedge (s_0 \oplus s_1) = (a \wedge s_0) \oplus (a \wedge s_1)$. Pour toutes ces instructions, il suffit d'ajouter une nouvelle instruction identique qui effectuera l'opération sur le second share, c'est le rôle de l'appel de fonction `createInstWithSameOpcodeAs(`I`,`s_1`)` de l'algorithme 3. Les résultats des deux instructions forment deux nouveaux shares.

Algorithme 3 : Transformation d'une instruction ayant un opérande secret

```
TransformInst1SecretOperand(I, f)
    Input : Instruction I, Function f
1   instOpcode = getOpcode(I)
2   s0 = getSecretOperand(I)
3   s1 = getSecondShare(s0)
4   switch instOpcode do
5       case shift, rotate, zext, sext, trunc, and
            // the operation must be applied to each share
6           createInstWithSameOpcodeAs(I, s1)
7       case xor
            // nothing to do: the xor must be done on one share only
8       case or
            // the following property is used:
            // a ∨ (s0 ⊕ s1) = (a ∨ s0) ⊕ (¬a ∧ s1)
            // the inst I already does (a ∨ s0), the second part must be added:
9           a = getPublicOperand(I)
10          createAnd(createNot(a), s1)
11      case load
12          if isAddressOfLoadPublic(I) then
13              createLoad(getAddressOfShare(s1))
14          else
                // load with an adress which depends on a secret
15              if isLoadOfConstTable(I) then
16                  replaceInstByPolynomialEvaluation(I)
17              else error
18      case store
19          if isAddressOfStorePublic(I) then
20              createStore(getAddressOfShare(s1), s1)
21          else error
22      case phi node
23          createPhiNode(s1, 0)
```

Ensuite, le OU EXCLUSIF ne nécessite aucune transformation : $a \oplus (s_0 \oplus s_1) = (a \oplus s_0) \oplus s_1$. Le second share du résultat du OU EXCLUSIF correspond au second share de l'opérande secret.

Pour le OU booléen, l'instruction originale calcule correctement un premier share, et l'algorithme créé deux nouvelles instructions pour calculer un second share en utilisant la relation $a \vee (s_0 \oplus s_1) = (a \vee s_0) \oplus (\neg a \wedge s_1)$.

Les lectures mémoire ayant une adresse publique ($AS1$) sont simplement dupliquées de manière à récupérer chaque share. Lorsque l'adresse est secrète, l'algorithme commence par vérifier que l'instruction de lecture mémoire fait un accès à une table constante. Si l'instruction correspond bien à un accès à une table constante, alors l'instruction est remplacée par une séquence d'instructions masquée permettant de faire une évaluation d'un

polynôme interpolateur de la table constante. La génération de ce polynôme est détaillée en section 4.5.7. Dans le cas contraire, l'instruction n'est pas gérée et la compilation s'arrête sur une erreur.

Les écritures mémoire sont gérées de manière similaire, à ceci près que les écritures mémoire indexées par un secret donnent systématiquement lieu à une erreur.

Enfin, pour les nœuds ϕ (présentés en section 2.4.1) ayant un opérande secret et un opérande publique, l'algorithme duplique le nœud ϕ en remplaçant l'opérande secret par le second share qui lui est associé, et l'opérande publique par la constante 0. Ainsi, dans les deux cas, les résultats des deux nœuds ϕ forment deux shares dont le OU EXCLUSIF donne bien la variable d'origine.

4.5.5 Transformation d'une instruction ayant deux opérandes secrets

Cette section présente la transformation des instructions ayant deux opérandes secrets, correspondant à l'appel à la fonction `TransformInst2SecretOperands(`I`, `f`)` de l'algorithme 1. L'algorithme 4 présente cette transformation. De la même manière que la transformation d'une instruction ayant un seul opérande secret, cette étape est effectuée en place.

Algorithme 4 : Transformation d'une instruction ayant 2 opérandes secrets

```
    TransformInst2SecretOperands(I, f)
        Input : Instruction I, Function f
1       instOpcode = getOpcode(I)
2       a_0 = getFirstSecretOperand(I)
3       a_1 = getSecondShare(a_0)
4       b_0 = getSecondSecretOperand(I)
5       b_1 = getSecondShare(b_0)
6       if shouldRemask(a_0,b_0) then
7           [a'_0, a'_1] = remaskOperand(a_0, a_1)
8           replaceSubsequentUseBy(a_0,a'_0)
9           replaceSubsequentUseBy(a_1,a'_1)
10      switch instOpcode do
11          case xor
                // keep the xor between a_0 and b_0 and add a new one
12              createXor(a_1, b_1)
13          case or
14              replaceInstByMaskedOr(I)
15          case and
16              replaceInstBySecMult(I)
17          case phi node
18              createPhiNode(a_1, b_1)
```

Pour les instructions ayant deux opérandes secrets, l'algorithme commence par analyser s'il est nécessaire d'effectuer un remasquage (appel à `shouldRemask` de l'algorithme 4). Lorsqu'une variable est remasquée (appel à `remaskOperand` de l'algorithme 4), toutes ses utilisations ultérieures sont remplacées par des utilisations de la variable remasquée (appels à `replaceSubsequentUseBy` dans l'algorithme 4). La variable remasquée dépend d'un nombre de masques plus grand, ce qui peut éviter d'autres remasquages.

Pour tester si un remasquage doit être effectué, il est nécessaire d'analyser la dépendance statistique des variables aux secrets : si le résultat de la combinaison des deux variables est statistiquement indépendant des secrets, alors il n'est pas nécessaire de remasquer. Notre mise en œuvre se base sur une approche syntaxique, moins fiable qu'une véritable analyse de dépendance statistique. Ce point est détaillé en section 4.5.6 et pourra évoluer au cours de travaux futurs.

Le remasquage est effectué en créant un appel à un générateur de nombre aléatoire pour obtenir un nouveau masque, et en effectuant le OU EXCLUSIF de ce masque avec chacun des deux shares de la variables à remasquer [Rivain et Prouff, 2010].

L'algorithme 4 traite ensuite les instructions en fonction de leur nature :

- Pour les OU EXCLUSIFS, un nouveau OU EXCLUSIF est créé entre les second shares des opérandes secrets.
- Pour les OU booléens, l'algorithme utilise la formule décrite dans [Baek et Noh, 2005]. L'application de cette formule est présentée dans l'algorithme 5.
- Pour les ET booléens, l'algorithme utilise la formule du secMult [Rivain et Prouff, 2010]. L'algorithme 6 présente l'application de cette formule, pour un masquage d'ordre 1.
- Enfin, pour les nœuds ϕ à deux opérandes tous deux secrets, l'algorithme ajoute un nouveau nœud ϕ ayant comme opérandes les seconds shares.

Algorithme 5 : Transformation d'un OU ayant deux opérandes secrets

```
   replaceInstByMaskedOr(I)
       Input : Instruction I
1      a_0 = getFirstSecretOperand(I)
2      a_1 = getSecondShare(a_0)
3      b_0 = getSecondSecretOperand(I)
4      b_1 = getSecondShare(b_0)
       // the instruction I computes a_0 ∨ b_0
       // the following instructions must be added to compute the first share:
5      and_0 = createAnd(a_0, b_1)
6      res_0 = createXor(and_0, I)
       // the following instructions must be added to compute the second share:
7      and_1 = createAnd(a_1, b_1)
8      or_1 = createOr(a_1,b_0)
9      res_1 = createXor(or_1, and_1)
```

4.5.6 Gestion des masques lors de la phase d'application

Pour gérer les remasquages, une liste des masques de chaque variable secrète est maintenue. L'utilisateur a la charge du masquage initial des variables qu'il déclare comme

Algorithme 6 : Transformation d'un AND ayant deux opérandes secrets

```
replaceInstBySecMult(I)
    Input : Instruction I
1   a_0 = getFirstSecretOperand(I)
2   a_1 = getSecondShare(a_0)
3   b_0 = getSecondSecretOperand(I)
4   b_1 = getSecondShare(b_0)
5   r_01 = createCall(RNG)
    // the instruction I computes a_0 ∧ b_0
    // the following instruction must be added to compute the first share:
6   res_0 = createXor(I, r_01)
    // the following instructions must be added to compute the second share:
7   and_01 = createAnd(a_0, b_1)
8   and_10 = createAnd(a_1, b_0)
9   xor = createXor(and_01, r_01)
10  r_10 = createXor(xor, and_10)
11  and_11 = createAnd(a_1, b_1)
12  res_1 = createXor(and_11, r_10)
```

secrètes, et pour chacune de ces variables, le second share est considéré comme étant un masque. L'algorithme applique des règles de propagation des masques afin de déterminer pour chaque instruction transformée la liste des masques du résultat. L'analyse de masque est basée uniquement sur une dépendance syntaxique à un masque, de la même manière que [Moss et al., 2012]. Comme nous allons le voir après, cette analyse n'est pas sure à 100% comme montré par [Eldib et Wang, 2014] et peut donner à la fois des faux positifs et des faux négatifs.

Instructions à un opérande secret. La variable résultat d'une instruction qui n'a qu'un seul opérande secret hérite de la liste de masques de l'opérande secret.

Instructions à deux opérandes secrets. La liste de masques du résultat d'une instruction à deux opérandes secrets est calculée à partir des listes de masques des deux opérandes. Nous calculons la liste Mr des masques du résultat à partir des deux listes Ma et Mb des opérandes ainsi : $Mr = (Ma \cup Mb) \setminus (Ma \cap Mb)$. Si $Mr = \emptyset$, l'un des opérandes est remasqué : on effectue un OU EXCLUSIF de chaque share de l'opérande avec un nouveau masque. L'ensemble de masques du résultat peut alors être recalculé.

Cas des nœuds phi. Dans le cas d'un nœud ϕ, la liste des masques de la variable résultat correspond à celle d'un de ses opérandes. Les opérandes secrets source du nœud ϕ peuvent avoir des ensembles de masques différents. Dans ce cas, on ne peut pas définir de manière unique l'ensemble des masques du résultat du nœud ϕ. La gestion des masques doit donc être adaptée.

L'ensemble des masques d'une variable résultat d'un nœud ϕ n'est pas déterminable statiquement à la compilation, puisqu'on ne sait pas le chemin d'exécution qui a mené à ce nœud ϕ. On considère alors l'ensemble des possibilités : la variable résultat peut

hériter son ensemble de masques soit du premier opérande, soit du deuxième. Au lieu d'associer un ensemble de masques à la variable résultat, on lui associe alors l'ensemble des ensembles de masques possibles. Donc la variable résultat portera les deux ensembles associés à chacun des opérandes du nœud ϕ.

Toutes les opérations sur les masques sont adaptées afin de travailler avec des ensembles d'ensembles de manière à explorer toutes les possibilités. En particulier, lorsqu'une instruction combine deux opérandes secrets qui ont chacun plusieurs ensembles de masques possibles, le calcul de Mr est fait sur toutes les combinaisons possibles d'ensembles de masques. Un remasquage est réalisé dès lors qu'une des combinaisons résulte en un ensemble vide. Le remasquage ajoute alors un masque à tous les ensembles de masques associés à l'opérande qui est remasqué.

Des dépendances cycliques peuvent exister, si le nœud ϕ dépend d'une variable qui dépend elle-même du résultat du nœud ϕ. On considère alors que le nœud ϕ a un opérande d'initialisation qui lui donne sa première valeur, et un opérande cyclique qui met sa valeur à jour, à chaque itération par exemple dans le cas d'une boucle. Dans ce cas, l'analyse des masques essaie de déterminer un ensemble d'ensembles de masques pour cette variable en suivant l'algorithme 7. L'idée générale de cet algorithme est de simuler la propagation des masques, comme si on masquait les instructions mais sans modifier la fonction, en bouclant tant que l'ensemble des ensembles de masques possibles du nœud ϕ s'agrandit. Si la propagation ne donne pas lieu à l'ajout de masques dans cet ensemble au bout d'un certain nombre d'itérations, l'algorithme a trouvé l'ensemble de tous les ensembles de masques possibles pour le nœud ϕ. Cet algorithme ne converge pas toujours, et le nombre d'itérations de l'algorithme est donc borné par l'utilisateur : si l'algorithme dépasse cette limite alors il renvoie un ensemble vide, et le résultat du nœud ϕ sera alors remasqué, ce qui changera le masque du résultat du nœud ϕ à toutes les itérations et permettra de continuer l'application de la contre-mesure.

Une fois l'ensemble d'ensemble de masques trouvé, le nœud ϕ et son bloc de base sont masqués normalement.

Limitations de la méthode d'analyse des masques utilisée. La méthode que nous utilisons pour déterminer si un remasquage est nécessaire a l'avantage d'être simple, mais elle peut donner lieu à des remasquages inutiles, voire à une absence de remasquage à un endroit où il serait nécessaire d'en avoir un dans certains cas, car l'analyse de dépendance à un masque n'est pas une condition suffisante.

Si la fonction effectue un OU EXCLUSIF entre un secret et lui même, bien que le résultat soit nul et donc indépendant de la valeur du secret, cette méthode indiquera qu'un remasquage est nécessaire : nous sommes dans le cas d'un remasquage inutile.

Les listings 4.4 et 4.5 montrent une fonction pour laquelle un remasquage devrait être effectué mais n'est pas détecté par l'analyse. L'analyse est trompée lorsqu'une combinaison de deux secrets juxtapose des bits au lieu de les combiner. Les commentaires du listing 4.5 indiquent les masques déterminés par la règle que nous utilisons. Le code de la fonction commence par effectuer des décalages à gauche et à droite sur les shares des deux secrets, de manière à ce que les shares de `k` aient leurs 16 bits de poids faible à 0 et que les shares de `n` aient leurs 16 bits de poids fort à 0. Ensuite, un OU EXCLUSIF est effectué

Listing 4.4 – Fonction à masquer

```
1  #pragma maskara secret k
2  #pragma maskara secret n
3  uint32_t f(uint32_t k[2], uint32_t n[2])
4  {
5          a = k[0] << 16;
6          b = n[0] >> 16;
7          c = a ^ b;
8          d = c & a;
9          return d;
10 }
```

Listing 4.5 – Fonction mal masquée

```
1  #pragma maskara secret k
2  #pragma maskara secret n
3  /* soit m1 le masque de k[0] et k[1], et m2 celui de n[0] et n[1]*/
4  uint32_t shares f(uint32_t k[2], uint32_t n[2])
5  {
6                           /* liste de masques issue de l'analyse de masques */
7          a_0 = k[0] << 16; /* masque : m1 */
8          a_1 = k[1] << 16; /* masque : m1 */
9          b_0 = n[0] >> 16; /* masque : m2 */
10         b_1 = n[1] >> 16; /* masque : m2 */
11         c_0 = a_0 ^ b_0;  /* masques : m1,m2 */   ← la partie haute de c_0
12                                                     n'est masquée que par m1
13                                                     et la partie basse n'est
14                                                     masquée que par m2
15         c_1 = a_1 ^ b_1;     /* masques : m1,m2 */  ← même constat que c_0
16         r01 = rand();        /* debut du secmult */
17         and00 = c_0 & a_0;   /* masque : m2 */
18         res_0 = r01 ^ and00; /* masques : m2,r01 */
19         and01 = c_0 & a_1;   /* masque : m2 */      ← démasquage ici
20         and10 = c_1 & a_0;   /* masque : m2 */      ← démasquage ici
21         xor = and10 ^ r01;   /* masques : m2,r01 */
22         r10 = xor ^ and10;   /* masque : r01 */
23         and11 = c_1 & a_1;   /* masque : m2 */
24         res_1 = and11 ^ r10; /* masque : r01 */
25         return {res_0,res_1};
26 }
```

Algorithme 7 : Constitution de l'ensemble d'ensemble de masques pour un nœud ϕ à deux opérandes

```
   L ← getPhiNodeMaskList(I, timeout)
       Input : Instruction I, integer timeout
       Output : List of Mask Lists L
       // get the operand which initialize the phi node value
1      iv = getInitVariable(I)
       // get the operand which exhibits cyclic dependency
2      cv = getCyclicVariable(I)
3      addToList(L, getMaskLists(iv))
4      for i=0 ; i < timeout ; i++ do
           // determine the mask lists of all instructions of the basic
           // block assuming L is the phi node's mask lists
           // in particular, this determines the mask lists of cv
           // it does not affect L
5          propagateMaskListToInstructionsOfCurrentBasicBlock()
           // test if mask lists from this iteration are already in our list
6          if isSubGroupOf(L, getCurrentIterationMaskLists(cv)) then
7              return L
8      return emptyList()
```

entre les shares de `k` et les shares de `n` (lignes 11 et 15). À cet endroit, le résultat de l'analyse des masques n'est plus correct. En effet, les variables résultant du OU EXCLUSIF contiennent les bits des shares de `k` en poids forts, et les bits des shares de `n` en poids faibles, et chaque bit n'est masqué que par un seul des deux masques, et non pas par les deux masques comme l'analyse le considère. L'application de la formule du secMult pour masquer le ET booléen va ensuite combiner `c_0` et `a_1`, ce qui va effectuer, sur les poids forts, un OU EXCLUSIF entre `a_0` et `a_1` (ligne 19), ce qui révèle $a_0 \oplus a_1 = k << 16$. De même, la combinaison de `c_1` et `a_0` va également effectuer un OU EXCLUSIF entre `a_1` et `a_0` (ligne 20). L'analyse utilisée ne permet donc pas de détecter tous les démasquages potentiels.

Des analyses plus poussées existent [Eldib et Wang, 2014, Ben El Ouahma et al., 2017, Wang et al., 2019] pour déterminer plus finement si un remasquage est nécessaire et avoir l'assurance que chaque bit est correctement masqué, et pourraient être implémentées également lors de travaux futurs.

De part les limitations de la méthode que nous utilisons actuellement, le code masqué produit par l'algorithme doit être vérifié pour garantir l'absence de démasquage. Les transformations de code du back-end obligent elles aussi à effectuer une telle vérification, comme nous le verrons dans la section 4.6.3.

4.5.7 Transformation des accès aux tables constantes

Cette section présente comment les instructions de lecture mémoire accédant à des tables constantes sont transformées pour que le code soit masqué. En particulier, ces instructions

incluent les accès à des tableaux de substitution, appelés *SBoxes* dans les primitives cryptographiques et dont la structure est un réseau de substitution/permutation, qui associent à chaque entier d'un ensemble de départ un autre entier d'un ensemble d'arrivé. Par exemple, dans le cas de l'AES, la SBox est une bijection de l'ensemble des entiers de l'intervalle [0,255] dans lui même.

Nous désignerons abusivement toutes les tables constantes par le terme de SBox dans la suite, même si certaines tables constantes pourraient être utilisées sans but de substitution.

4.5.7.1 Vue d'ensemble

L'utilisation de SBoxes est détectée par la présence d'une lecture mémoire indexée par un secret, et dont l'adresse de base est celle d'une table constante.

Nous avons choisi de masquer les SBoxes en les remplaçant par des évaluations de polynômes interpolateurs. Nous suivons la méthode décrite dans [Coron et al., 2014c] pour déterminer un polynôme interpolateur sous une forme permettant un calcul efficace. De plus, nous avons apporté ensuite plusieurs optimisations afin d'accélérer l'évaluation du polynôme, qui seront présentées en section 4.5.7.6. Le code permettant l'évaluation masquée du polynôme est émis en décomposant l'évaluation du polynôme en évaluations masquées des monômes et en combinaisons par multiplications non linéaires et de OU EXCLUSIFS des résultats intermédiaires. Il est à noter que le code masqué est produit directement à partir de l'expression analytique du polynôme. Ainsi, nous ne passons pas par une étape intermédiaire d'émission d'un code non masqué, puis de masquage de celui-ci.

Ces polynômes interpolateurs sont définis sur des corps finis binaires, ce qui demande de présenter quelques prérequis mathématiques pour expliquer l'approche.

4.5.7.2 Notions mathématiques : corps finis binaires

Cette sous-section aborde des notions relatives au corps finis, et vise à introduire la représentation des entiers par des polynômes dans ces corps, et à faire la distinction entre l'utilisation des polynômes comme représentation d'entiers de ces corps et l'utilisation des polynômes comme fonctions polynomiales définies sur ces corps.

Définition 5. Un corps (C,$\oplus$,$\otimes$) est un ensemble d'éléments muni d'une addition $\oplus$ et d'une multiplication $\otimes$. Ces 2 opérations vérifient les axiomes suivants :

- associativité de l'addition et de la multiplication :
  $\forall x,y,z \in C$, $\forall$ op $\in \{\oplus, \otimes\}$, $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$
- commutativité de l'addition et de la multiplication :
  $\forall x,y \in C$, $\forall$ op $\in \{\oplus, \otimes\}$, $x \text{ op } y = y \text{ op } x$
- existence d'un élément neutre pour l'addition (noté 0) et pour la multiplication (noté 1) :
  $\exists 0 \in C$, $\forall x \in C$, $x \oplus 0 = 0 \oplus x = x$
  $\exists 1 \in C$, $\forall x \in C$, $x \otimes 1 = 1 \otimes x = x$

TABLE 4.2 – Exemples de quelques éléments d'un corps fini écrits sous diverses représentations

Polynôme en α	Encodage binaire	Entier associé à l'encodage binaire
$\alpha^2 + \alpha^0$	101	5
$\alpha^4 + \alpha^3 + \alpha^1$	11010	26
α^1	10	2

— existence d'un inverse additif et d'un inverse multiplicatif (sauf pour 0)
$\forall x \in C, \exists y \in C, x \oplus y = 0$
$\forall x \in C \setminus \{0\}, \exists y \in C, x \otimes y = 1$
— distributivité de la multiplication sur l'addition :
$\forall x,y,z \in C, x \otimes (y \oplus z) = x \otimes y \oplus x \otimes z$

Nous considérerons ici des corps finis binaires, c'est à dire des corps finis tels que $\forall a \in C$, $a \oplus a = 0$. Les corps qui vérifient cette propriété sont dits de *caractéristique 2*. On les note F_{2^n}, où 2^n est le nombre d'éléments du corps. Les éléments de F_{2^n} sont souvent représentés à l'aide de polynômes. La multiplication correspond à la multiplication polynomiale modulo un polynôme irréductible fixé noté PI, qui peut être choisi parmi des listes connues de polynômes irréductibles (comme par exemple la liste [pri] extraite de [Moon, 2005]). On suppose que ce polynôme irréductible PI admet une racine α, ce qui donne l'équation : $PI[\alpha] = 0$. Tous les éléments du corps sont alors représentés sous forme de polynômes en α, par exemple $\alpha^2 + \alpha^0$.

Pour encoder ces éléments en entiers, on remplace simplement α par le chiffre 2. Par exemple, $\alpha^2 + \alpha^0$ s'encode comme $2^2 + 2^0$, donc comme l'entier 5. Par la suite, nous considérerons dans nos notations que $\alpha = 2$, afin que le lien entre les polynômes et leurs encodages en champs de bits soit direct. Le tableau 4.2 donne quelques exemples d'éléments d'un corps fini représentés sous forme polynomiale, binaire, et entière.

L'équation induite par le polynôme irréductible donne donc un modulo [1]. Par exemple, dans F_{2^2} (qui comporte uniquement les éléments {0,1,2,3})), le polynôme $P = 1 + X + X^2$ est irréductible, et on peut donc l'utiliser pour définir la multiplication. On obtient l'équation $1+\alpha+\alpha^2 = 0$, ce qui sous forme entière donne $1+2+2^2 = 0$, c'est à dire $7 = 0$. Sous cette forme, on voit que cette équation agit comme un modulo 7. Les chiffres plus petits que 7 mais étant hors de l'ensemble, comme 4, peuvent être ramenés dans l'ensemble considéré en écrivant l'équation autrement : $1 + 2 + 2^2 = 0 \Leftrightarrow 2^2 = 1 + 2 \Leftrightarrow 4 = 3$. C'est une différence importante par rapport au modulo usuel de $(\mathbb{N}, +, *)$; ici comme l'addition est le OU EXCLUSIF, il n'y a pas besoin d'atteindre la valeur du modulo pour reboucler. Ainsi, l'équation permet de ramener dans l'ensemble considéré n'importe quel entier : elle permet de simplifier les polynômes ayant un degré supérieur ou égal au degré du polynôme primitif. Pour indiquer que les calculs s'effectuent relativement à l'équation induite par le polynôme PI, nous utiliserons la notation $\underset{\text{PI}}{=}$ à la place du $=$.

1. l'addition et la multiplication étant celles du corps fini, ce modulo ne donne pas les même résultats que le modulo usuel sur $(\mathbb{N}, +, *)$

Définition 6. Dans un corps fini à 2^n éléments, on appelle réduction l'opération qui consiste à trouver, pour un entier i supérieur ou égal à 2^n, l'entier j dans [0,2^n[tel que $i \underset{\text{PI}}{=} j$, où PI est le polynôme irréductible choisi pour ce corps.

Par exemple, considérons le corps F_{2^8} muni du polynôme irréductible $PI[X] = X^8 + X^4 + X^3 + X^2 + 1$, qui induit l'équation $2^8 \underset{\text{PI}}{=} 2^4 + 2^3 + 2^2 + 1$, c'est-à-dire $256 \underset{\text{PI}}{=} 29$. Réduisons l'entier $2^{17} + 2^9 + 2$. Les termes en gras sont les termes qui seront modifiés à la ligne suivante, et ceux soulignés sont ceux qui ont été modifiés à la ligne courante :

$$\mathbf{2^{17}} + 2^9 + 2 \underset{\text{PI}}{=} \underline{\mathbf{2^{8+9}}} + 2^9 + 2 \tag{4.1}$$
$$\underset{\text{PI}}{=} \underline{\mathbf{2^8} * 2^9} + 2^9 + 2 \tag{4.2}$$
$$\underset{\text{PI}}{=} \underline{\mathbf{(2^4 + 2^3 + 2^2 + 1)} * \mathbf{2^9}} + 2^9 + 2 \tag{4.3}$$
$$\underset{\text{PI}}{=} \underline{2^{13} + 2^{12} + 2^{11} + \mathbf{2^9}} + \mathbf{2^9} + 2 \tag{4.4}$$
$$\underset{\text{PI}}{=} \mathbf{2^{13}} + 2^{12} + 2^{11} + 2 \tag{4.5}$$
$$\underset{\text{PI}}{=} \underline{\mathbf{2^8} * 2^5} + 2^{12} + 2^{11} + 2 \tag{4.6}$$
$$\underset{\text{PI}}{=} \underline{\mathbf{(2^4 + 2^3 + 2^2 + 1)} * \mathbf{2^5}} + 2^{12} + 2^{11} + 2 \tag{4.7}$$
$$\underset{\text{PI}}{=} \underline{2^9 + 2^8 + 2^7 + 2^5} + \mathbf{2^{12}} + 2^{11} + 2 \tag{4.8}$$
$$\underset{\text{PI}}{=} 2^9 + 2^8 + 2^7 + 2^5 + \underline{\mathbf{2^8} * 2^4} + 2^{11} + 2 \tag{4.9}$$
$$\underset{\text{PI}}{=} 2^9 + 2^8 + 2^7 + 2^5 + \underline{\mathbf{(2^4 + 2^3 + 2^2 + 1)} * \mathbf{2^4}} + 2^{11} + 2 \tag{4.10}$$
$$\underset{\text{PI}}{=} 2^9 + \mathbf{2^8} + \mathbf{2^7} + 2^5 + \underline{\mathbf{2^8} + \mathbf{2^7} + 2^6 + 2^4} + 2^{11} + 2 \tag{4.11}$$
$$\underset{\text{PI}}{=} 2^9 + 2^6 + 2^5 + 2^4 + \mathbf{2^{11}} + 2 \tag{4.12}$$
$$\underset{\text{PI}}{=} 2^9 + 2^6 + 2^5 + 2^4 + \underline{\mathbf{(2^4 + 2^3 + 2^2 + 1)} * \mathbf{2^3}} + 2 \tag{4.13}$$
$$\underset{\text{PI}}{=} 2^9 + \mathbf{2^6} + \mathbf{2^5} + 2^4 + \underline{2^7 + \mathbf{2^6} + \mathbf{2^5} + 2^3} + 2 \tag{4.14}$$
$$\underset{\text{PI}}{=} \mathbf{2^9} + 2^7 + 2^4 + 2^3 + 2 \tag{4.15}$$
$$\underset{\text{PI}}{=} \underline{\mathbf{(2^4 + 2^3 + 2^2 + 1)} * \mathbf{2}} + 2^7 + 2^4 + 2^3 + 2 \tag{4.16}$$
$$\underset{\text{PI}}{=} \underline{2^5 + \mathbf{2^4 + 2^3 + 2}} + 2^7 + \mathbf{2^4 + 2^3 + 2} \tag{4.17}$$
$$\underset{\text{PI}}{=} 2^7 + 2^5 \tag{4.18}$$
$$\tag{4.19}$$

Nous obtenons ainsi l'entier réduit.

L'opération de réduction étant définie, on s'intéresse maintenant à la multiplication. La multiplication correspond à la multiplication usuelle sur les polynômes, à deux différences près : l'addition est le OU EXCLUSIF ($a \oplus a \neq 2 \otimes a$), et le résultat est réduit petit à petit en remplaçant X^8 par $X^4 + X^3 + X^2 + 1$. Ainsi, pour multiplier 2 entiers a et b, on commence par exprimer chacun d'eux comme somme de puissances de 2. Ensuite, on développe le calcul de multiplication, et on calcule les produits de puissances de 2. Chaque fois qu'un

terme apparaît 2 fois, on l'élimine, car l'addition est le OU EXCLUSIF. On réduit enfin les puissances qui deviennent trop grandes.

Par exemple :

$$
\begin{aligned}
74 * 7 &\underset{\text{PI}}{=} (2^6 + 2^3 + 2) * (2^2 + 2 + 1) && (4.20)\\
&\underset{\text{PI}}{=} 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + \mathbf{2^3 + 2^3} + 2^2 + 1 && (4.21)\\
&\underset{\text{PI}}{=} \mathbf{2^8} + 2^7 + 2^6 + 2^5 + 2^4 + 2^2 + 1 && (4.22)\\
&\underset{\text{PI}}{=} \underline{\mathbf{2^4} + 2^3 + \mathbf{2^2 + 1}} + 2^7 + 2^6 + 2^5 + \mathbf{2^4 + 2^2 + 1} && (4.23)\\
&\underset{\text{PI}}{=} 2^7 + 2^6 + 2^5 + 2^3 && (4.24)\\
&\underset{\text{PI}}{=} 232 && (4.25)
\end{aligned}
$$

Pour effectuer la multiplication rapidement, sans devoir développer le produit des 2 polynômes, il est possible de transformer tout polynôme (sauf l'élément 0) en un monôme équivalent dont l'exposant est compris dans $[0; 255]$. Pour cela, on peut utiliser une table de correspondance (table de `log`), indexée par la représentation entière du polynôme, et renvoyant l'exposant du monôme équivalent [fas]. Pour multiplier les polynômes, on fait alors la somme des 2 exposants obtenus via la table de `log`, pour obtenir un monôme correspondant au résultat du produit. Il faut ensuite faire une réduction de ce monôme pour obtenir un polynôme de degré plus petit que 8. La réduction d'un monôme peut également se faire rapidement en utilisant une table de correspondance (table d'`alog`), permettant d'obtenir à partir de l'exposant du monôme la représentation entière du polynôme associé. Ce processus sera détaillé dans la section 4.5.7.6.

Dans la suite, nous utiliserons également des polynômes pour interpoler une fonction $f : [0 : 2^n - 1] \longrightarrow [0 : 2^n - 1]$. Dans ce cas, on cherche un polynôme Q tel que $\forall x \in [0 : 2^n - 1], Q(x) = f(x)$, les coefficients de Q sont dans $[0 : 2^n - 1]$, et son degré est inférieur ou égal à $2^n - 1$. Un tel polynôme existe toujours puisque l'on peut prendre le polynôme de Lagrange. Ainsi, dans ce cas, Q est un polynôme défini dans le but d'être utilisé comme une fonction, contrairement aux polynômes présentés dans cette section qui servent à la représentation polynomiale d'entiers.

4.5.7.3 Approche

Dans notre approche de masquage, les accès en lecture à une table de substitution, ou SBox, sont remplacés par l'évaluation masquée d'un polynôme interpolateur ($x \mapsto P[x]$). Pour trouver un polynôme adéquat, nous suivons l'approche présentée par [Coron et al., 2014c]. Cette approche permet de trouver une expression polynomiale ayant peu de multiplications non linéaires, afin que l'application du masquage ne soit pas trop coûteuse. En effet, pour effectuer une multiplication non linéaire masquée à l'ordre 1, il est nécessaire d'effectuer un tirage d'un nombre aléatoire, quatre multiplications de corps, et quatre OU EXCLUSIFS. Afin de limiter le nombre de ces multiplications, l'approche de Coron et al. vise à bien exploiter l'élévation au carré, parce qu'elle est linéaire par rapport au OU EXCLUSIF, ce qui rend le masquage de cette opération peu coûteux : $(a_0 + a_1)^2 = a_0^2 + a_0 * a_1 + a_1 * a_0 + a_1^2 = a_0^2 + a_0 * a_1 + a_0 * a_1 + a_1^2 = a_0^2 + a_1^2$. Cette

approche consiste à trouver le polynôme interpolateur sous la forme suivante, où les q_i et les p_i sont des polynômes construits en utilisant principalement l'élévation au carré :

$$P[X] = \sum_{i=1}^{t-1} p_i(X) \cdot q_i(X) + p_t(X). \tag{4.26}$$

Pour comprendre l'intérêt de l'approche proposée par Coron et al., considérons tout d'abord une approche plus directe. Il est facile de trouver un polynôme interpolateur pour une table définie dans un corps fini binaire à 2^n éléments en utilisant par exemple le polynôme interpolateur de Lagrange. On aura alors un polynôme défini par : $P[X] = \sum_{\beta=0}^{2^n-1} c_\beta \cdot X^\beta$. Pour évaluer ce polynôme de manière masquée, on considère les différentes opérations une à une : les OU EXCLUSIFS entre secrets et les multiplications d'un secret par un scalaire sont des fonctions linéaires par rapport au OU EXCLUSIF, et se masquent comme toute fonction linéaire, et l'évaluation des monômes requiert des multiplications non linéaires qui forment alors les opérations les plus coûteuses. Pour calculer X^β, on peut procéder à une exponentiation rapide. Calculons par exemple X^{241} :
$X^{241} = (((((((X^2 * X)^2 * X)^2 * X)^2)^2)^2)^2) * X$
Cette exponentiation rapide nécessite 7 élévations au carré (qui est une opération linéaire) et 4 multiplications non linéaires pour calculer ce monôme. C'est beaucoup, mais on peut remarquer qu'on n'a pas calculé un seul monôme, mais plusieurs : les étapes intermédiaires du calcul ont permis de calculer X^2, X^3, X^6, X^7, X^{14}, X^{15}, X^{30}, X^{60}, X^{120}, X^{240}. De plus, à partir du moment où X^{15} est connu, on a calculé 4 autre monômes (X^{30}, X^{60}, X^{120}, X^{240}) en effectuant uniquement des élévations au carré. Ainsi, à partir de quelques monômes évalués, on peut en calculer rapidement beaucoup d'autres en faisant uniquement des élévations au carré.

Bien qu'élever un monôme au carré permette d'obtenir d'autres monômes, on retombe sur le monôme initial après quelques carrés successifs, de part les propriétés du corps fini. En fait, dans un corps fini binaire à 2^n éléments, on a $\forall \beta \geq 2^n - 1, X^\beta = X^{\beta-(2^n-1)}$.

On appelle classe cyclotomique d'un exposant β l'ensemble des exposants pouvant être obtenus en élevant X^β au carré une ou plusieurs fois. Le nombre d'éléments dans une classe cyclotomique dépend de l'exposant β : la classe cyclotomique de l'exposant 0 ne contient que lui même, au contraire de la classe cyclotomique de 15 qui, dans un corps fini à 2^8 éléments, contient $\{15, 30, 60, 120, 240, 225, 195, 135\}$. Une classe cyclotomique définie par rapport à un corps fini à 2^n éléments ne peut contenir plus de n éléments.

Cependant, même en optimisant le calcul des exposants et en exploitant les classes cyclotomiques pour en générer le maximum par élévation au carré, le nombre de multiplications non linéaires nécessaires pour l'évaluation de tous les monômes reste conséquent. Dans un ensemble à $2^n - 1$ exposants, il y a au moins $\frac{2^n - 2}{n} + 1$ classes cyclotomiques, puisque 0 est toujours seul dans sa classe et que les autres classes ont moins de n éléments. Pour un corps fini binaire à 2^8 éléments comme celui utilisé pour la SBox de l'AES, cela représente 33 classes minimum. Bien que la classe de 0 et la classe de l'exposant 1 ne nécessitent pas de multiplication non linéaire pour le calcul de leurs exposants, il est nécessaire d'effectuer une multiplication non linéaire pour chaque autre classe afin d'obtenir un monôme dont l'exposant appartient à la classe. Le nombre de multiplications non linéaires à effectuer est donc au moins égal à 31, ce qui est coûteux.

Afin de réduire ce coût, l'approche de Coron et al. permet d'obtenir un polynôme interpolateur sous une forme qui ne nécessite pas de calculer tous les monômes. Ainsi, le polynôme interpolateur n'est pas recherché sous la forme $P[X] = \sum_{\beta=0}^{2^n-1} c_\beta \cdot X^\beta$, mais sous la forme donnée dans l'équation 4.26 et rappelée ci dessous :

$$P[X] = \sum_{i=1}^{t-1} p_i(X) \cdot q_i(X) + p_t(X). \tag{4.27}$$

Les q_i et p_i sont des sous polynômes définis sur un ensemble de monômes restreint, choisi de manière à ce que n'importe quel monôme puisse s'exprimer comme le produit de deux autres monômes. La valeur de t est l'un des paramètres de l'approche.

Le fait de restreindre l'ensemble des monômes utilisés permet de diminuer le coût dû aux multiplications non linéaires, même si quelques multiplications non linéaires restent nécessaires pour multiplier les p_i avec les q_i. À titre d'exemple, pour la SBox de l'AES à 256 élements, cette approche permet de passer de 31 multiplications non linéaires minimum à seulement 10 multiplications non linéaires. Le gain est alors d'un facteur 3.

4.5.7.4 Déroulement de l'algorithme pour constituer le polynôme interpolateur

Lors du masquage d'un accès à une table constante, la première étape de l'algorithme de recherche de polynôme interpolateur consiste à retrouver les valeurs de la table constante. Une fois celles-ci connues, on calcule les différents paramètres de l'approche de Coron et al. : le nombre de classes cyclotomiques utilisées l, le nombre de produits $p_i(X) * qi(X)$ appelé $t-1$, et l'entier n tel que 2^n corresponde au nombre d'éléments de la SBox.

On calcule d'abord $t = \lceil \sqrt{\frac{2^n}{n}} \rceil$, puis $l = \lceil \frac{(2^n) + (n-1) * t}{n * t} \rceil$.

Selon Coron et al., ces formules permettent d'approcher un nombre de multiplications non linéaires optimal, tout en remplissant les conditions nécessaires pour qu'un polynôme interpolateur puisse être trouvé.

On se place dans un corps fini à 2^n éléments, en prenant un polynôme primitif PI parmi une liste de polynômes irréductibles connus.

On choisit alors un ensemble de l monômes tels que :

1. le monôme 1 est toujours choisi
2. chaque monôme nécessite une seule multiplication non linéaire pour être formé à partir des monômes précédemment choisis
3. la classe cyclotomique de l'exposant de chaque monôme contient n exposants distincts. Autrement dit, chaque monôme génère n monômes distincts lorsqu'on l'élève au carré plusieurs fois d'affilée.
4. tout monôme hors de l'ensemble peut s'écrire comme produit de 2 monômes de l'ensemble

Listing 4.6 – Table de substitution SBox de PRESENT

```
uint8_t SBox[16] = {0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD,
                    0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2};
```

Les règles 1 et 4 sont primordiales. La règle 3 n'est pas obligatoire, l'algorithme cherche d'abord à la respecter, mais l'enfreint s'il ne trouve pas de solution. La règle 2 est importante puisqu'elle force l'algorithme à choisir un ensemble de monômes pour lequel le nombre de multiplications non linéaires est minimal.

Ensuite, l'algorithme choisit $t-1$ polynômes aléatoires à partir de cet ensemble de monômes : il s'agit des polynômes q_i dans l'équation 4.26. Les polynômes p_i dans l'équation 4.26 sont alors déterminés par la résolution d'un système d'équations linéaires. Pour résoudre ce système, l'algorithme du pivot de Gauss est utilisé en considérant l'arithmétique du corps fini. Ainsi, l'addition est le OU EXCLUSIF, la soustraction est le OU EXCLUSIF, la multiplication est la multiplication du corps fini, et la division est l'inverse multiplicatif dans le corps fini.

Les sous polynômes p_i sont alors tous connus. Cela donne une expression de $P[X]$. Il ne reste plus qu'à générer le code permettant l'évaluation masquée de $P[X]$.

4.5.7.5 Exemple : SBox de PRESENT

Afin de mieux comprendre cette approche, nous allons l'appliquer ici sur un exemple simple : la SBox de PRESENT. La SBox de PRESENT contient 16 valeurs comprises entre 0 et 15. Elle est présentée dans le listing 4.6. Nous avons donc ici $n = 4$, le corps utilisé est $\mathbb{F}_{2^4}$.

On calcule d'abord t : $\lceil\sqrt{\frac{16}{4}}\rceil = 2$.

Ensuite, on détermine la valeur de l : $\lceil\frac{16+3*2}{4*2}\rceil = \lceil\frac{11}{4}\rceil = 3$.

On choisit le polynôme primitif $X^4 + X^1 + X^0 = 2^4 + 2^1 + 2^0 = 19$, qui est irréductible dans $\mathbb{F}_{2^4}$.

On peut ensuite choisir les l monômes. On choisit d'abord X^0. L'ensemble des monômes obtenus par carrés successifs est alors $\{X^0\}$. On choisit ensuite X^1. L'ensemble des monômes obtenus par carrés successifs devient $\{X^0, X^1, X^2, X^4, X^8\}$. Enfin, on choisit un monôme pouvant s'écrire en multipliant l'un de ces monômes par X. Par exemple, on choisit $X^3 = X^2 * X^1$. L'ensemble des monômes obtenus par carrés successifs devient $\{X^0, X^1, X^2, X^4, X^8, X^3, X^6, X^9, X^{12}\}$. X^9 est obtenu car $(((X^3)^2)^2)^2 = X^{24} = X^{15} * X^9 = X^9$ car $15 = 2^n - 1$ et $X^{2^n-1} = X^0$ d'après Coron et al.

On note que l'ensemble de monômes ne contient pas tous les monômes possibles. Notamment, $X^5, X^7, X^{10}, X^{11}, X^{13}, X^{14}$ ne sont pas dans l'ensemble des monômes. Pour que l'ensemble des monômes soit valide, il faut pouvoir écrire tous ces monômes comme produits de 2 monômes présents dans l'ensemble :

$X^5 = X^4 * X$
$X^7 = X^6 * X$
$X^{10} = X^9 * X$
$X^{11} = X^9 * X^2$
$X^{13} = X^{12} * X$
$X^{14} = X^{12} * X^2$
C'est bien le cas ici, l'ensemble de monômes est donc valide.

On peut maintenant former aléatoirement $t - 1 = 1$ polynôme à partir de l'ensemble de monômes : $q_1[X] = 12X^{12} + 14X^9 + 5X^8 + 7X^6 + 3X^4 + 15X^3 + 3X^2 + 8X + 8$.

Il ne reste plus qu'à trouver p_1 et p_2 tels que :

$$\forall x \in [0,15], q_1(x) * p_1(x) + p_2(x) = SBox[x] \tag{4.28}$$

Cette équation est réexprimée comme un système d'équations linéaires, dont la résolution par pivot de Gauss donne la solution suivante :

1. $p_1[X] = 3X^{12} + 15X^9 + 14X^8 + 7X^6 + 8X^4 + 4X^3 + 1X^2 + 11X + 5$
2. $p_2[X] = 0X^{12} + 0X^9 + 9X^8 + 1X^6 + 11X^4 + 15X^3 + 10X^2 + 5X + 2$

On peut vérifier que le polynôme est bien interpolateur en calculant pour chaque valeur de $x \in [0,15]$ le résultat de l'évaluation du polynôme. Par exemple, pour $x = 0$, on a :

$$\begin{aligned}
P(0) &= q_1(0) * p_1(0) + p_2(0) && (4.29)\\
&= 8 * 5 + 2 && (4.30)\\
&= 2^3 * (2^2 + 2^0) + 2 && (4.31)\\
&= 2^5 + 2^3 + 2 && (4.32)\\
&= 2^4 * 2^1 + 2^3 + 2 && (4.33)\\
&= (2^1 + 2^0) * 2^1 + 2^3 + 2 && (4.34)\\
&= 2^2 + 2^1 + 2^3 + 2^1 && (4.35)\\
&= 2^3 + 2^2 && (4.36)\\
&= 12 && (4.37)\\
&= SBox[0] && (4.38)
\end{aligned}$$

Le polynôme trouvé interpole donc bien la SBox de PRESENT en 0.

4.5.7.6 Optimisations

Afin que cette approche soit efficace, il est important d'avoir le moins de multiplications de corps fini possible, et que les multiplications restantes soient le plus optimisées possible. Cette section présente les optimisations effectuées, visant à améliorer le temps d'exécution de l'approche.

Optimisation de l'évaluation du polynôme interpolateur par regroupement d'opérations. Une optimisation consiste à regrouper dans chaque polynôme p_i ou q_i les monômes qui ont leurs exposants dans la même classe cyclotomique, puis à évaluer le

polynôme classe cyclotomique par classe cyclotomique [Coron et al., 2014c, remarque 3]. Plus précisément, soit A l'un des p_i ou q_i. Au lieu d'évaluer A directement, on évalue chaque sous polynôme B_k de A tel que tous les monômes de B_k peuvent être obtenus par élévations au carré successives d'un même monôme X^{α_k}. Comme nous avons choisi l classes cyclotomiques au départ, chaque polynôme p_i ou q_i est décomposé en l sous polynômes. Chaque B_k est une somme de termes sous la forme $\beta(X^{\alpha_k})^{2*\gamma}$. Pour chaque B_k, on évalue d'abord X^{α_k}. Chaque terme, une fois exprimé comme fonction de X^{α_k}, devient linéaire par rapport au OU EXCLUSIF. Soit B_{lin-k} le polynôme tel que $B_{lin-k}[X^{\alpha_k}] = B_k[X]$. Le polynôme B_{lin-k} est linéaire par rapport au OU EXCLUSIF. Ainsi, une fois les deux shares m_0 et m_1 associés à X^{α_k} connus, l'évaluation de B_k revient à calculer $B_{lin-k}[m_0]$ et $B_{lin-k}[m_1]$. B_{lin-k} peut être entièrement tabulé de manière à ce que son évaluation soit efficace. Cette méthode permet alors d'éviter toutes les multiplications par des scalaires, et un grand nombre des élévations au carré.

Par exemple, pour nos q_i et p_i choisis pour PRESENT, nous avons 3 classes cyclotomiques, une pour l'exposant 0, une pour les exposants 1,2,4,8 et une pour les exposants 3,6,9,12. On réécrit les polynômes de la manière suivante :

1. $q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$
2. $p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$
3. $p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$

Pour évaluer $12X^{12} + 14X^9 + 7X^6 + 15X^3$, on évalue d'abord $z = X^3$, puis on évalue $12z^4+14z^8+7z^2+15z$. Ce dernier polynôme ne contient que des élévations à des puissances de 2, des additions et des multiplications par des constantes, il est linéaire par rapport au OU EXCLUSIF. En tabulant toutes les valeurs possibles de z pour ce polynôme, l'évaluation se résume à calculer $z = X^3$, puis à faire un accès tabulé pour chaque share. On procède de la même manière pour évaluer les autres sous-polynômes des q_i et p_i.

Optimisation de l'évaluation des l monômes choisis. La seconde optimisation effectuée est liée à la règle 2 dans le choix des monômes. Initialement, l'exposant 0 et l'exposant 1 sont toujours choisis. Le choix des autres exposants est restreint de manière à pouvoir évaluer efficacement ces exposants : on choisit uniquement des exposants de la forme $\beta = \alpha * (2^k + 1)$ où α est un exposant déjà choisi. Cela permet d'utiliser l'algorithme proposé dans [Coron et al., 2014b, algorithme 5] qui permet d'évaluer directement $(X^\alpha)^{2^k+1}$ sans passer par une exponentiation suivie d'une multiplication non linéaire.

Par exemple, pour PRESENT, après avoir choisi les monômes 1 et X, on ne choisit pas X^{14} parce que 14 ne s'écrit pas sous la forme $\beta = \alpha * (2^k + 1)$ avec α dans $\{0, 1, 2, 4, 8\}$.

L'évaluation des l monômes pourrait être plus optimisée en utilisant l'approche proposée par [Mathieu-Mahias et Quisquater, 2018]. Cette approche donne plus de flexibilité dans le choix des monômes et fournit une méthode d'évaluation plus rapide en passant par un masquage multiplicatif. Nous n'avons pas mis en œuvre cette approche faute de temps.

Optimisation du choix des coefficients des qi Une partie des coefficients des polynômes q_i peuvent être fixés à 0 afin d'éviter des opérations. Pour rappel, l'évaluation des q_i se fait par sous-polynômes dont les monômes ont tous leurs exposants dans une même

Listing 4.7 – Table de log et d'alog pour les entiers de $\mathbb{F}_{2^4}$, avec le polynôme primitif 19 et calcul de la multiplication de corps fini

```
uint8_t log[16] = {0, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[15] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9};

uint8_t mul(uint8_t a, uint8_t b) {
       return ((a != 0) && (b != 0)) * alog[(log[a] + log[b]) %15];
}
```

classe cyclotomique, et ces sous-polynômes sont tabulés. Pour éviter des calculs en fixant des coefficients à 0, il faut alors que tous les coefficients d'un même sous-polynôme d'un q_i soient nuls. Pour chaque q_i, on choisit aléatoirement quelques classes cyclotomiques, et on fixe à 0 tous les coefficients associés aux monômes dont les exposant appartiennent aux classes choisies.

Cette optimisation fait à la fois gagner en temps d'exécution et en consommation mémoire, puisqu'on a alors moins de sous-polynômes à évaluer. L'inconvénient de cette optimisation est que le système d'équations linéaires permettant de trouver les p_i peut devenir insoluble. Il faut alors rechoisir des polynômes q_i avec des coefficients nuls différents, et réessayer de résoudre le système.

Dans le cas de PRESENT, on peut par exemple obtenir :

1. $q_1[X] = \underline{(0X^{12} + 0X^9 + 0X^6 + 0X^3)} + (0X^8 + 0X^4 + 13X^2 + 13X) + (7)$
2. $p_1[X] = (1X^{12} + 13X^9 + 3X^6 + 9X^3) + (15X^8 + 7X^4 + 15X^2 + 14X) + (11)$
3. $p_2[X] = (11X^{12} + 11X^9 + 0X^6 + 0X^3) + (8X^8 + 0X^4 + 0X^2 + 10X) + (8)$

Optimisations de la multiplication de corps fini. L'évaluation de P requiert des multiplications de corps fini. Nous avons choisi d'utiliser des tables de `log` et `alog` pour le code de ces multiplications. Comme présenté en section 4.5.7.2, la table de `log` permet de passer d'un entier non nul sous représentation polynomiale à l'exposant d'un monôme représentant le même entier. La table d'`alog` permet de faire la transformation inverse. La multiplication de deux entiers **non nuls** a et b devient alors : $a * b = alog[(log(a) + log(b)) \ mod \ (2^n - 1)]$. Le listing 4.7 montre les tables de `log` et d'`alog` utilisées pour PRESENT dans $\mathbb{F}_{2^4}$ avec le polynôme primitif 19, ainsi que le code nécessaire pour effectuer la multiplication de deux éléments du corps fini.

Nous avons utilisé 2 optimisations en lien avec ce calcul pour optimiser son temps d'exécution. La première optimisation vise à éliminer le modulo à chaque multiplication, et la deuxième à gérer de manière plus efficace le cas où a ou b vaut 0.

La première optimisation consiste à agrandir la table d'`alog` afin d'éviter le calcul du modulo $(2^n - 1)$ [fas]. Cette opération de modulo peut s'avérer coûteuse sur des architectures qui n'ont pas d'instruction machine dédiée. Pour ce faire, on remarque que comme $log(a) < 2^n - 1$ et $log(b) < 2^n - 1$, on a $log(a) + log(b) < 2^{n+1} - 3$. En calculant une table d'`alog` de taille $2^{n+1} - 3$ au lieu de $2^n - 1$, on peut donc s'affranchir du modulo. La multiplication de deux entiers non nuls a et b devient alors : $a * b = alog[(log(a) + log(b))]$.

Listing 4.8 – Table de log et d'alog pour les entiers de $\mathbb{F}_{2^4}$, avec le polynôme primitif 19, et calcul de la multiplication de corps fini après une optimisation visant à s'affranchir du modulo

```
uint8_t log[16] = {0, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[30] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9,
                    1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9};

uint8_t mul(uint8_t a, uint8_t b) {
       return ((a != 0) && (b != 0)) * alog[log[a] + log[b]];
}
```

Listing 4.9 – Table de log et d'alog pour les entiers de $\mathbb{F}_{2^4}$, avec le polynôme primitif 19, et calcul de la multiplication de corps fini après les deux optimisations qui permettent de s'affranchir du modulo et du code gérant le cas du 0

```
uint8_t log[16] = {31, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[63] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9,
                    1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

uint8_t mul(uint8_t a, uint8_t b) {
       return alog[log[a] + log[b]];
}
```

Le listing 4.8 montre les tables ainsi obtenues, et le code pour effectuer la multiplication de corps fini mis à jour.

La seconde optimisation est, à notre connaissance, une nouvelle optimisation. Elle vise à gérer le cas où a ou b est égal à 0, afin d'éviter de devoir gérer ce cas comme un cas particulier. Typiquement, la multiplication de deux entiers quelconques a et b peut s'écrire $a * b = ((a! = 0)\&\&(b! = 0)) \times alog[(log(a) + log(b))]$, où $\times$ désigne ici la multiplication usuelle sur $\mathbb{N}$. La gestion de ce cas représente alors une partie importante du calcul. Notre optimisation vise à modifier les tables de `log` et d'`alog` pour gérer le cas du 0 de manière identique aux autres cas. Pour cela, on définit $log(0)$ de manière à ce que $log(x) + log(0)$ soit toujours supérieur au maximum qu'il est possible d'obtenir avec des nombres non nuls. Pour tous les nombres qu'il est possible d'obtenir en calculant $log(x) + log(0)$, la table d'`alog` est définie à 0. Comme on a $log(a) + log(b) < 2^{n+1} - 3$, il est possible de prendre $log(0) = 2^{n+1} - 3$. Suivant la valeur de n, il peut être nécessaire de transformer la table de `log` pour qu'elle renvoie des entiers encodés sur plus de bits. La table d'`alog` est ensuite agrandie pour qu'elle renvoie 0 pour toutes les valeurs supérieures ou égales à $log(0) = 2^{n+1} - 3$. Le listing 4.9 montre les tables obtenues après cette optimisation, et le code nécessaire pour effectuer une multiplication de corps fini : le test de nullité a été éliminé.

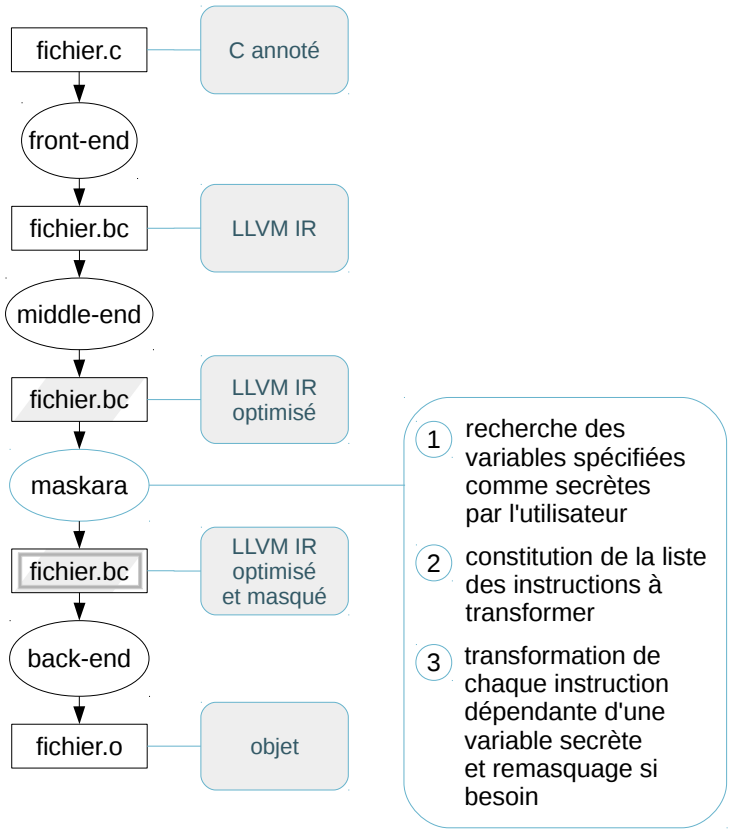

FIGURE 4.1 – Étapes de l'application automatisée du masquage

4.6 Mise en œuvre de l'algorithme dans LLVM

Cette section présente l'implémentation de l'algorithme au sein de LLVM, une infrastructure de compilateur [llv].

4.6.1 Choix du placement de l'application dans le compilateur

Le principe d'application pourrait être mis en place à divers endroits dans le compilateur. Cependant, le masquage est une contre-mesure qui peut souffrir des optimisations du compilateur : celles-ci peuvent par exemple intervertir deux instructions commutatives comme deux OU EXCLUSIFS successifs, ce qui peut produire des démasquages. Nous avons choisi de l'appliquer en toute fin du middle-end, de manière à ce qu'elle soit après les différentes optimisations du middle-end.

4.6.2 Vue générale du flot d'application de la contre-mesure dans le compilateur

L'application de la contre-mesure de masquage d'ordre 1 se déroule au sein d'une passe d'application, appelée Maskara. Elle est appliquée à la toute fin du middle-end, sur le code en représentation intermédiaire de LLVM sous forme SSA.

Listing 4.10 – Fichier C original avant toute modification

```
unsigned char state[16];
unsigned char key[16];
void addRoundKey() {
  for(int i=0; i < 16; i++) {
    state[i] ^= key[i];
  }
}
```

Cette passe applique la contre-mesure à la granularité de la fonction, sur une ou plusieurs fonctions spécifiées par l'utilisateur à l'aide de pragmas. L'utilisateur doit indiquer aussi les variables ou paramètres secrets, comme la clé dans le cas d'une fonction de chiffrement. L'utilisateur est en charge du masquage des données secrètes utilisées par les fonctions, et du démasquage des données produites par celles-ci. La passe se charge de transformer le code des fonctions pour que les calculs soient effectués avec les secrets séparés en shares.

De plus, une convention est établie afin de pouvoir retrouver facilement les 2 shares de chaque secret préalablement masqué par l'utilisateur : l'utilisateur doit déclarer ses variables de manière à toujours placer les shares de manière contiguë en mémoire dans un tableau, et allouer la mémoire pour ceux-ci. Ainsi, à partir de l'adresse d'un share (par exemple un pointeur), on peut calculer l'adresse du second (en incrémentant le pointeur pour pointer sur le second share).

Cet agencement n'est pas une contrainte forte pour notre approche, il serait possible d'utiliser des annotations dans le code source pour que l'utilisateur puisse indiquer où se trouve les shares de chaque variable d'entrée secrète.

La passe Maskara implante les différentes étapes de l'algorithme 1 comme montré sur la figure 4.1.

4.6.2.1 Exemple d'application de la contre-mesure par Maskara

Afin de mieux comprendre les différentes étapes d'application de la contre-mesure, prenons l'exemple d'une fonction `AddRoundKey` et considérons l'implémentation décrite dans le listing 4.10.

Pour pouvoir appliquer la passe de masquage sur le code, il faut d'abord modifier le code de manière à respecter la convention de contiguïté des shares. Dans le listing 4.11, le code est adapté pour pouvoir travailler avec 2 shares pour chaque octet de `state` et de `key`. L'utilisateur se charge d'effectuer la décomposition du secret en shares avant cette fonction, et de démasquer le résultat ensuite. Cependant, le corps de la fonction est réécrit comme si le premier share correspondait à la variable originale, sans utiliser le second share. Après application du masquage sur la fonction, les deux shares seront utilisés dans le code, et l'utilisateur doit donc anticiper cela dans le reste de son code pour décomposer les secrets en shares et recomposer la sortie.

Listing 4.11 – Fichier C original modifié par l'utilisateur. Des annotations sont ajoutées, et les shares sont placés de manière contiguë en mémoire. Seul le premier share est utilisé dans le code avant l'application du masquage. Le code reste fonctionnel si l'utilisateur définit le premier share comme étant le secret et le second share comme étant nul.

```
unsigned char state[16][2];
unsigned char key[16][2];
#pragma maskara secure_function addRoundKey
#pragma maskara secret state
#pragma maskara secret key
void addRoundKey() {
  for(int i=0; i < 16; i++) {
    *(state[i]) ^= *(key[i]);
  }
}
```

Listing 4.12 – Code LLVM IR après les optimisations et avant application du masquage

```
define void @addRoundKey() #0 {
  br label %1
  ; <label>:1 ; preds = %2, %0
  %i.0 = phi i32 [ 0, %0 ], [ %8, %2 ]
  %exitcond = icmp eq i32 %i.0, 16 ; condition de sortie de boucle
  br i1 %exitcond, label %9, label %2
  ; <label>:2 ; preds = %1
  %3 = getelementptr inbounds [16 x [2 x i8]], [16 x [2 x i8]]* @key, i32 0,
      i32 %i.0, i32 0
  %4 = load i8, i8* %3, align 1 ; load key+%3+0
  %5 = getelementptr inbounds [16 x [2 x i8]], [16 x [2 x i8]]* @state, i32
      0, i32 %i.0, i32 0
  %6 = load i8, i8* %5, align 1 ; load state+%3+0
  %7 = xor i8 %6, %4
  store i8 %7, i8* %5, align 1 ; store du resultat dans le state
  %8 = add nuw nsw i32 %i.0, 1 ; increment du compteur de boucle
  br label %1
  ; <label>:9 ; preds = %1
  ret void
}
```

Le code C du listing 4.11 passe ensuite dans le front-end qui génère un code au format LLVM IR. Celui-ci est ensuite optimisé dans le middle-end. On obtient alors le code présenté dans le listing 4.12. On peut reconnaître dans le code IR les variables, ici appelées `@state` et `@key`, et voir les différents éléments de la boucle de la fonction `AddRoundKey`.

La passe de masquage est appliquée sur ce code. Celle-ci commence par rechercher les variables déclarées secrètes par l'utilisateur, ici `state` et `key`, qu'elle retrouve grâce à leurs noms. Une fois ces variables trouvées, une analyse de flot de donnée est effectuée pour identifier le niveau de confidentialité de chaque variable et pour déterminer les instructions dépendantes de variables secrètes. Par exemple, dans le listing 4.12, `%7` est secrète puisqu'il s'agit du résultat d'une opération manipulant des secrets (`%6` et `%4`). Au contraire, `%8` est publique puisqu'elle ne dépend pas de variable secrète, il s'agit de l'incrément du compteur de la boucle. Une fois ces analyses terminées, l'application du masquage peut démarrer. Toutes les instructions manipulant des secrets sont transformées petit à petit pour aboutir au code du listing 4.13. On constate par exemple que les loads ont été dupliqués afin de récupérer les 2 shares de chaque variable, qu'il y a également 2 xors pour obtenir les 2 shares du résultat, et que le store (instruction d'écriture en mémoire) a également été dupliqué pour pouvoir sauvegarder ces 2 shares en mémoire. Les instructions manipulant des variables indépendantes des secrets sont restées inchangées.

4.6.3 Modification de la sélection d'instructions

Nous avons constaté que la sélection d'instructions du back-end ARM peut impacter la contre-mesure : elle rassemble parfois 2 lectures mémoire d'un octet à des adresses contiguës en une seule lecture mémoire de deux octets (un demi-mot). Cela induit une fuite en valeur lorsque les 2 octets à charger sont deux shares d'une valeur secrète. En effet, en considérant que la fuite suit le poids de Hamming de la valeur, et avec $share_0 = s \oplus rand$ et $share_1 = rand$, on a : $HW(share_0||share_1) = HW(share_0) + HW(share_1) = HW(s \oplus rand) + HW(rand)$ où $||$ désigne ici la concaténation. Si $s = 255$, on a $HW(share_0||share_1) = HW(rand) + HW(\neg rand) = HW(rand) + 8 - HW(rand) = 8$. Ainsi, le poids de Hamming du demi-mot a une distribution qui dépend de la valeur du secret, puisque lorsque un poids de Hamming de 2 est observé (ce qui est possible si $s = 0$ par exemple), l'hypothèse $s = 255$ peut être exclue.

Ce problème peut être résolu en forçant la sélection d'instruction à ne pas sélectionner ce motif. Pour cela, on peut modifier le code de la sélection d'instruction, ou changer le niveau d'optimisation particulier pour cette passe. Nous avons opté pour cette deuxième option. Dans les deux cas, seule la sélection d'instructions est impactée, sans affecter le reste des optimisations du back-end, qui restent activées.

4.7 Évaluation expérimentale

Cette section présente l'évaluation expérimentale de l'approche en temps d'exécution, en t-test non spécifique, et par analyse formelle.

Listing 4.13 – Code LLVM IR après application du masquage

```
define void @addRoundKey_secure() #0 {
  br label %1
; <label>:1 ; preds = %2, %0
  %i.0 = phi i32 [ 0, %0 ], [ %5, %2 ]
  %exitcond = icmp eq i32 %i.0, 16 ; condition de sortie de boucle
  br i1 %exitcond, label %6, label %2
; <label>:2 ; preds = %1
  %3 = getelementptr inbounds [16 x [2 x i8]], [16 x [2 x i8]]* @key, i32 0,
      i32 %i.0, i32 0
  %share1_key = load i8, i8* %3, align 1 ; load key+%3+0
  %address_share2_key = getelementptr i8, i8* %3, i32 1
  %share2_key = load i8, i8* %address_share2_ ; load key+%3+1
  %4 = getelementptr inbounds [16 x [2 x i8]], [16 x [2 x i8]]* @state, i32
      0, i32 %i.0, i32 0
  %share1_state = load i8, i8* %4, align 1 ; load state+%3+0
  %address_share2_state = getelementptr i8, i8* %4, i32 1
  %share2_state = load i8, i8* %address_share2_state ; load state+%3+1
  %share1_res = xor i8 %share1_state, %share1_key ; xor entre les 1ers shares
  %share2_res = xor i8 %share2_state, %share2_key ; xor entre les 2nds shares
  store i8 %share1_res, i8* %4, align 1 ; store du 1er share du resultat
  %address_share2_res = getelementptr i8, i8* %4, i32 1
  store i8 %share2_res, i8* %address_share2_res ; store du 2nd share du
      resultat
  %5 = add nuw nsw i32 %i.0, 1 ; increment du compteur de boucle
  br label %1
; <label>:6 ; preds = %1
  ret void
}
```

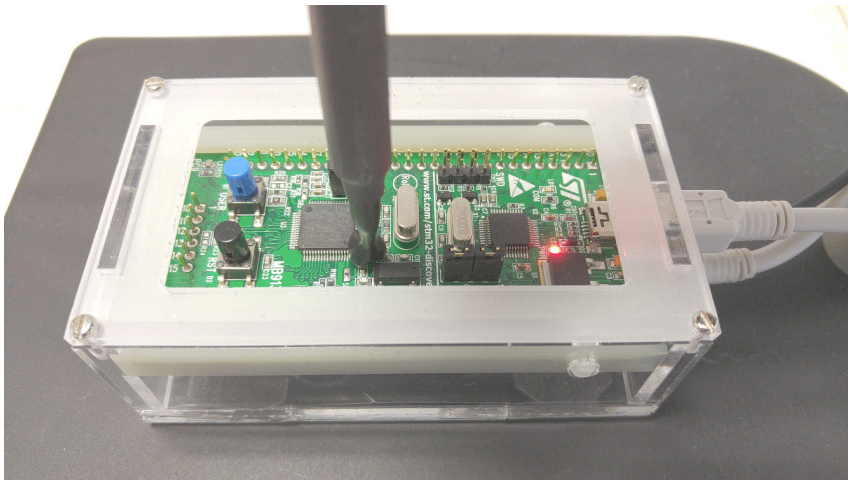

FIGURE 4.2 – Configuration expérimentale

4.7.1 Environnement expérimental

Pour évaluer cette mise en œuvre, nous avons choisi une plate-forme embarquée contrainte : une carte STM32VLDISCOVERY de STMicroelectronics (figure 4.2), équipée d'un cœur Cortex-M3 fonctionnant à 24 MHz, 8 kB de RAM et 128 kB de mémoire flash. Cette plate-forme ne fournit aucun mécanisme de sécurité matérielle contre les attaques par canal auxiliaire.

Nous effectuons des t-tests non spécifique en mesurant les émissions électromagnétiques. La configuration pour les mesures d'émissions électromagnétiques comprend un PicoScope 2208A, une sonde EM RF-U 5-2 de Langer et un préamplificateur PA 303 de Langer. Le picoscope dispose d'une bande passante de 200 MHz et d'une résolution verticale de 8 bits. Le taux d'échantillonnage est de 500 Msample/s (ce qui donne 20,83 échantillons par cycle CPU), et le nombre d'échantillons enregistrés a été adapté pour chaque mesure afin de couvrir la zone d'intérêt. Pour faciliter la synchronisation des mesures pour l'évaluation en t-test, un signal est émis sur une broche GPIO au début de la fonction évaluée. Le picoscope déclenche l'acquisition à l'instant où ce signal est émis. Les mesures acquises sont alors toutes alignées : le n-ième échantillon de chaque mesure correspond systématiquement au même instant de l'exécution du programme.

4.7.2 Évaluation en performance

Nous avons d'abord évalué l'impact sur les performances de l'approche sur des fonctions variées. Nous avons évalué séparément chacune des fonctions d'une implémentation maison d'AES et d'une implémentation de Simon 32/64 tirée de [Khandouzy et Azizzadeh, 2015]. Nous avons choisi ces fonctions car d'une part l'AES est fortement utilisé, et d'autre part le masquage booléen est très adapté aux fonctions de Simon. En effet, Simon s'appuie sur les opérations de ET booléen, rotation, et de OU EXCLUSIF, et ne contient pas de SBox. De plus, nous évaluons ensuite l'impact sur les performances du masquage d'un accès SBox pour des SBoxes de taille variées : celles de l'AES (8 bits), de DES (6 bits) et de PRESENT (4 bits).

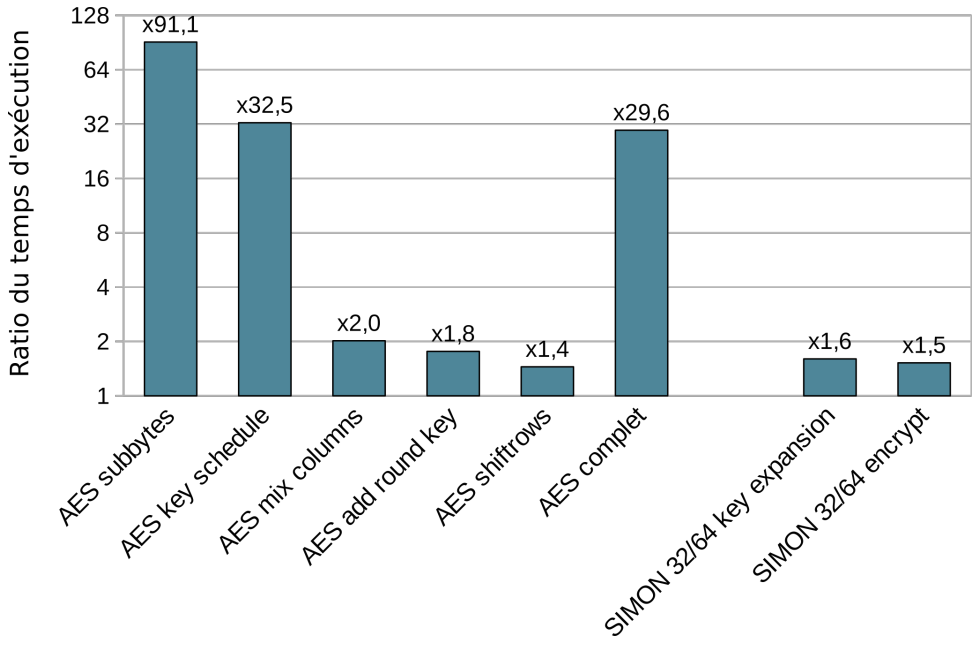

FIGURE 4.3 – Ratio du temps d'exécution pour diverses fonctions masquées automatiquement par Maskara. L'axe des ordonnées est en échelle logarithmique.

L'impact en performance est mesuré en calculant le ratio du nombre de cycles passés dans une fonction masquée divisé par le nombre de cycles utilisés pour exécuter la même fonction non masquée. La figure 4.3 présente le surcoût en performance pour les différentes fonctions. Les résultats montrent que le surcoût en performance varie énormément d'une fonction à l'autre, allant de $\times 1.4$ à $\times 91.1$.

Le surcoût en performance dépend de plusieurs paramètres. Tout d'abord, il dépend de la proportion d'instructions manipulant des variables secrètes. Ensuite, il dépend énormément de la nature des instructions à masquer. Par exemple, masquer un OU EXCLUSIF avec une constante n'implique pas de surcoût. Masquer un OU EXCLUSIF entre 2 secrets, un ET booléen avec une constante, une lecture mémoire ou écriture mémoire d'un secret, ou encore une rotation par une constante revient à dupliquer l'instruction. Masquer un ET booléen entre 2 secrets coûte déjà bien plus cher, puisqu'il est nécessaire d'effectuer un appel au générateur de nombre aléatoire puis d'effectuer 8 opérations. Masquer un accès à une SBox représente un surcoût encore bien plus important.

Le surcoût de masquage de la fonction `SubBytes` de l'AES est ainsi beaucoup plus élevé que celui de la fonction `ShiftRows` qui ne contient que des lectures et écritures mémoire de secrets. La fonction `KeySchedule` de l'AES a, comme la fonction `SubBytes`, des accès à une SBox, et coûte également extrêmement cher. Cependant, l'accès à la SBox constitue la majeure partie des calculs effectués dans la fonction `SubBytes`, ce qui n'est pas le cas dans la fonction `KeySchedule`. Cela explique l'écart des surcoûts de ces deux fonctions.

La contre-mesure de masquage booléen est particulièrement adaptée pour la fonction de chiffrement Simon qui n'utilise que des rotations, des OU EXCLUSIFS et des ET booléens : pour les deux fonctions considérées, le surcoût en performance est très faible ($\times 1{,}6$ et $\times 1{,}5$).

Le surcoût du masquage des SBoxes étant dépendant de la taille des tables, nous avons mesuré les surcoûts d'un accès à une SBox masquée pour différentes tailles de SBox. Nous avons choisi les SBoxes de l'AES, du DES et de PRESENT. La SBox de l'AES comporte 256 éléments, celle du DES 64 éléments, et celle de PRESENT 16 éléments. La figure 4.4 présente les surcoûts en temps d'exécution obtenus. Nous observons que l'impact sur les performances diminue rapidement lorsque la taille de la SBox décroît.

Nous avons également évalué l'impact de l'optimisation consistant à mettre à 0 un certain nombre de coefficients des polynômes q_i (présentée en section 4.5.7.6) pour éviter des calculs, en définissant à 1/2 la probabilité de mettre tous les coefficients associés à une classe d'exposants à 0 pour un polynôme q_i. Cette optimisation permet de gagner jusqu'à 21% pour la SBox de l'AES, 15% pour la SBox du DES, et 5% pour la SBox de PRESENT, comme présenté figure 4.4. Cette optimisation permet donc de diminuer significativement le coût de la contremesure pour les SBoxes assez grandes.

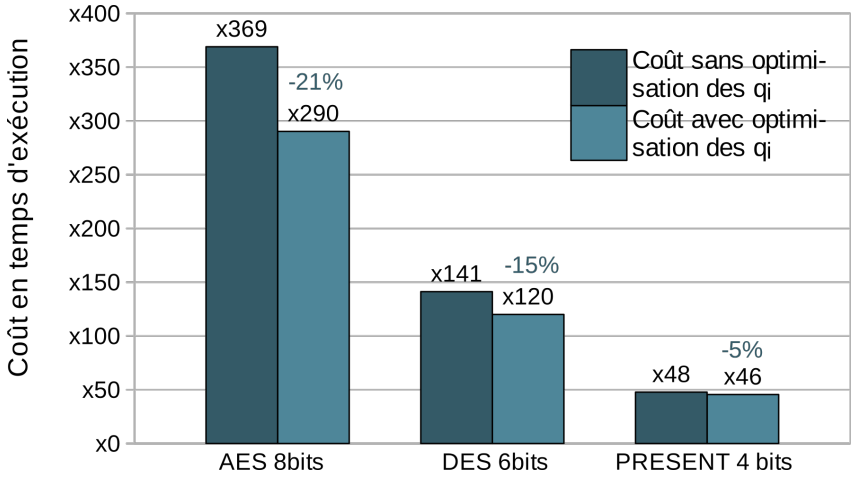

FIGURE 4.4 – Coût en temps d'exécution pour effectuer l'équivalent d'un accès à une SBox pour des SBox de tailles variées avec ou sans notre optimisation des coefficients des q_i.

4.7.3 Évaluation en sécurité

Cette section présente l'évaluation en sécurité de l'approche avec une analyse symbolique et un t-test non spécifique.

4.7.3.1 Analyse formelle

Deux fonctions de l'AES sécurisées par Maskara ont tout d'abord été analysées en utilisant un outil de vérification formelle qui a été développé dans le cadre du projet PROSECCO [pro]. Cet outil a été présenté dans la publication [Ben El Ouahma et al., 2017], il implante une analyse symbolique au niveau binaire permettant de déterminer pour chaque variable dépendant du secret si la distribution statistique de ses valeurs est indépendante du secret. Il permet notamment de vérifier si l'application d'un masquage d'ordre 1 est correcte en sortie du compilateur.

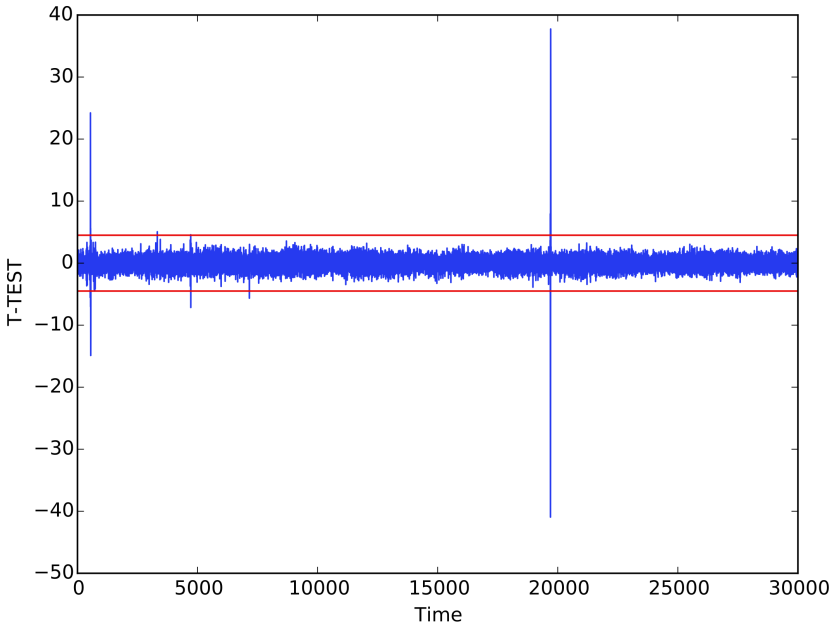

FIGURE 4.5 – t-test non spécifique sur l'équivalent masqué d'un accès à la SBox de l'AES

Nous avons réalisé l'analyse de la fonction `AddRoundKey` de l'AES, ainsi qu'une fonction effectuant un accès SBox de l'AES, toutes deux masquées automatiquement par Maskara.

Aucune fuite en valeur d'ordre 1 n'a été reportée par l'outil. Ce résultat nous permet d'avoir une confiance sur le bon déroulement de l'application de la contre-mesure par Maskara, et indique que pour ces fonctions là le back-end du compilateur n'a pas atténué la contre-mesure, et les limitations de l'implémentation de la gestion des masques n'ont pas donné lieu à de démasquage.

4.7.3.2 Analyse en t-test non spécifique

Pour l'évaluation sécuritaire expérimentale, nous avons choisi d'évaluer une fonction masquée effectuant un accès à la SBox de l'AES. L'évaluation est réalisée avec un t-test non spécifique (méthode d'évaluation présentée en section 2.5.2) en mesurant 2×10000 traces d'émissions électromagnétiques. Le résultat de ce t-test est montré en figure 4.5. La présence de plusieurs pics, plus grands que 4,5 en valeur absolue indique que des émissions électromagnétiques restent dépendantes des données secrètes.

La contre-mesure ne semble pas suffisante pour éliminer toute fuite d'information. Plusieurs hypothèses pourraient expliquer la présence de corrélations aux données. Tout d'abord, la contre-mesure pourrait être mal appliquée (il manque des remasquages par exemple). Ensuite, le back-end du compilateur atténue peut être la contre-mesure. Enfin, l'hypothèse de fuite en valeur ne tient peut être pas pour la plate-forme ciblée. L'analyse formelle effectuée sur le code binaire n'a pas indiqué de fuite en valeur, ce qui nous oriente vers la dernière hypothèse : le modèle de fuites en valeur ne modélise pas correctement le comportement de la plate-forme. D'autre part, une analyse formelle ciblant cette fois-ci les fuites en transitions entre registres et entres accès mémoire a indiqué la présence de

deux transitions registres-registres, et n'a pas pû conclure à l'absence de fuite en transition sur près de 1000 variables. De plus, le pic en début de t-test pourrait correspondre à une fuite en transition entre deux lectures mémoire d'octets visant à récupérer les deux shares d'entrée. Nous faisons donc l'hypothèse que le modèle de fuite en transition doit être considéré pour mieux modéliser le comportement de la plate-forme.

4.8 Modification du back-end ARM et nouvelle évaluation sécuritaire

Afin de mieux comprendre les phénomènes constatés, nous avons ajouté une passe en toute fin du back-end ARM pour tenter de supprimer, ou du moins atténuer les fuites en transition. Cette passe n'a pas pour vocation de produire un code performant, mais uniquement de vérifier l'hypothèse de fuite en transition.

Nous supposons que les fuites pouvaient provenir de transitions lors des accès mémoires lorsque les données transitent sur un bus, ce qui expliquerait le pic constaté en début de t-test, ou de calculs dans l'ALU lorsque les données transitent entre les registres et l'ALU, puisque [Le Corre et al., 2018] ont constaté sur un Cortex-M3 la présence de registres dans la micro-architecture qui peuvent faire apparaître des fuites en transition entre deux utilisations de l'ALU. La nouvelle passe permet d'ajouter une ou plusieurs instructions inutiles avant chaque instruction, en fonction de la nature de cette dernière, dans le but de limiter la réutilisation immédiate d'une même ressource matérielle pour deux donnés partageant le même masque. Nous avons considéré l'ajout des instructions suivantes :

- des lectures mémoires relatives au PC avant chaque lecture et écriture mémoire, pour charger une donnée depuis la mémoire flash,
- un `push` (sauvegarde d'un registre sur la pile) suivi d'un `pop` (chargement dans un registre d'une valeur sur la pile) avant chaque lecture et écriture mémoire,
- une addition entre un registre et un immédiat avant toutes les autres instructions.

Bien que cette passe ajoute des instructions inutiles, celles-ci sont ajoutées statiquement et sont identiques sur toutes les traces.

Il est à noter que des fuites en transition peuvent subsister après l'ajout de ces instructions ; les instructions inutiles ne permettent que de supprimer les transitions dans l'ALU et entre les accès mémoires, mais les transitions entre deux instructions définissant le même registre restent présentes.

Pour pouvoir ajouter ces instructions sans modifier la sémantique du code, le registre `R9` a été réservé et n'est donc pas utilisé pour les instructions utiles. De plus, la passe de création des blocs IT [2] est désactivée, puisqu'il serait complexe d'insérer de telles instructions entre les instructions d'un blocs IT puisque le nombre d'instructions dans le then et le else est limité.

Nous avons effectué des t-tests non spécifiques sur la fonction effectuant un accès à la SBox de l'AES recompilée avec le back-end modifié, en adaptant le nombre d'échantillons

2. Blocs d'instructions conditionnellement exécutés permettant d'effectuer un if then else. La première instruction encode la condition ainsi que le nombre d'instructions dans le then et le nombre d'instructions dans le else.

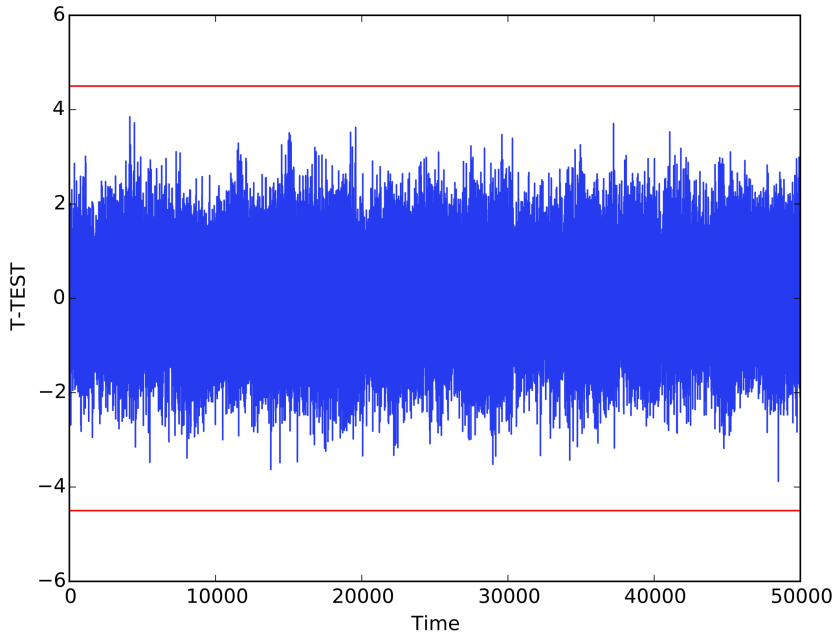

FIGURE 4.6 – t-test obtenu sur une fonction d'accès à la SBox masquée, après modification du back-end ARM

des traces au nombre d'instructions ajoutées, en essayant plusieurs combinaisons d'ajout d'instructions. Comme précédemment, tous les t-tests ont été réalisés en mesurant 2×10000 traces d'émissions électromagnétiques. Nous avons tout d'abord configuré la passe pour ajouter uniquement une addition avant chaque instruction du programme. La fuite constatée est restée présente, mais la valeur `t` maximale obtenue était alors plus faible que sans ces instructions ($\sim$ 9 au lieu de $\sim$ 40). Nous avons ensuite essayé d'ajouter un push/pop avant chaque lecture mémoire et chaque écriture mémoire, et une addition avant les autres instructions. Nous avons constaté que la fuite d'information restait visible, mais que la valeur `t` maximale diminuait pour prendre une valeur proche de 5. Nous avons alors essayé l'ajout d'une lecture mémoire PC relative avant chaque lecture mémoire et chaque écriture mémoire, et une addition avant les autres instructions. Cette fois, nous avons obtenu un t-test réussi. Nous avons alors répété le t-test 5 fois, et obtenu un t-test échoué sur les 5, avec une valeur `t` maximale de 4,72.

Nous avons alors effectué une nouvelle expérience en insérant deux lectures mémoire relatives au PC avant chaque lecture mémoire et chaque écriture mémoire pour augmenter l'écart entre les accès mémoires utiles, et une addition avant les autres instructions. Pour cette configuration d'ajout d'instructions, aucun des 5 t-tests effectués n'a montré de fuite d'information. La figure 4.6 montre le résultat de l'un de ces t-tests.

Nous avons enfin effectué des t-tests en ajoutant deux lectures mémoire PC relatives avant chaque lecture mémoire et chaque écriture mémoire, sans ajouter d'addition, pour vérifier que l'ajout systématique d'une addition a bien un impact. Nous avons alors obtenu 2 t-tests échoués sur 5, avec des valeurs `t` maximales proches de 5. L'ajout d'une addition en plus des lectures mémoire semble donc bien contribuer à la diminution de la fuite d'information.

Cette évaluation avec modification du back-end conforte l'hypothèse que des fuites en transitions liées à la micro-architecture existent sur la plate-forme. Les fuites en transition semblent être liées à plusieurs ressources matérielles sur cette plate-forme, puisqu'il nous a fallu combiner des accès mémoire et des opérations de calcul pour que les t-tests réussissent. Ces analyses ouvrent des pistent d'investigation et d'amélioration discutées ci-dessous.

4.9 Discussion

L'application de masquage à la compilation est pour l'instant limitée par l'absence de connaissance précise de la micro-architecture de la plate-forme cible. Bien que la plate-forme utilisée ici soit simple (pipeline 3 étages dans l'ordre et sans spéculation), le modèle de fuite en valeur ne suffit pas et il est nécessaire de prendre aussi en compte les fuites en transitions. Cependant, pour pouvoir les prendre en compte efficacement, il est nécessaire de connaître en détail la micro-architecture de la plate-forme ciblée.

La connaissance du niveau ISA est ici insuffisante pour appliquer correctement la contre-mesure. En effet, la présence de registres intermédiaires non visibles au niveau ISA peut suffire à amoindrir voire à éliminer l'effet de la contre-mesure comme c'est le cas pour l'injection de fautes [Laurent et al., 2019].

De plus, le back-end du compilateur est également capable de détériorer la contre-mesure dans certains cas : nous avons constaté qu'il pouvait créer des fuites en valeur en regroupant des lectures mémoire lors de la sélection d'instruction, mais d'autres optimisations pourraient aussi créer de nouvelles fuites. En particulier, avec un modèle de fuites en transition il est nécessaire de modifier l'allocation de registres pour qu'un registre ne soit pas utilisé successivement pour deux shares, et pour éviter que deux sauvegardes de données sur la pile à la suite ne créent une fuite.

Il en résulte un besoin de caractériser la cible pour adapter le back-end du compilateur au types de fuites constatés, en extension des travaux [Wang et al., 2019] qui proposent d'adapter la passe d'allocation de registres. Ce besoin est d'autant plus marqué si l'on considère un masquage d'ordre supérieur, afin que des fuites en transition ne diminuent pas l'ordre du masquage [Balasch et al., 2015]. Il faut alors modifier les passes du back-end pour garantir qu'aucune transition ne vienne faire diminuer l'ordre du masquage.

4.10 Conclusion

Dans ce chapitre, nous avons présenté une approche pour appliquer automatiquement une contre-mesure de masquage booléen de premier ordre. Dans le cadre du masquage des accès à des tables constantes, nous avons choisi de suivre une approche par masquage de l'évaluation d'un polynôme interpolateur [Coron et al., 2014c], et nous avons proposé plusieurs optimisations pour améliorer l'efficacité de cette approche.

Nous avons pu constater des surcoûts en performance très variés en fonction du code ciblé : certaines fonctions présentent un surcoût faible de ×1,4, tandis que les fonctions contenant des SBoxes induisent des surcoûts plus importants. Le surcoût de masquage des SBoxes a ensuite été évalué séparément : il apparaît être fortement dépendant de la taille de la SBox à masquer.

Ensuite, un t-test a été effectué sur le code sécurisé d'un accès à la SBox de l'AES. Nous avons observé d'importants pics de la valeur `t` qui indiquent la présence de fuite d'information. Nous avons constaté que la sélection d'instructions peut introduire des fuites en valeur, mais surtout que le modèle de fuite en valeur ne semble pas adapté pour la plate-forme cible puisque nous sommes parvenus à obtenir des t-tests réussis en introduisant simplement des instructions inutiles pour essayer de supprimer les transitions.

La contre-mesure de masquage nous apparaît comme particulièrement sensible à la fois aux éventuels choix du compilateur, notamment dans les passes du back-end, ainsi qu'aux effets micro-architecturaux. Bien que la passe permette une application de la contre-mesure indépendante de l'architecture cible, un travail important de caractérisation est nécessaire pour pouvoir adapter le back-end en fonction de la plate-forme cible pour préserver le travail effectué par la passe du middle-end. La prise en compte de la micro-architecture ouvre donc des pistes de recherche.

De plus, l'approche pourrait être également étendue pour gérer les masques avec une analyse de remasquage plus fiable, puisque celle que nous avons utilisée ne détecte pas toujours les problèmes de démasquage. L'approche pourrait également supporter d'autres types de masquage comme le masquage arithmétique, et permettre de masquer un code à un ordre quelconque.

Chapitre 5

Application automatique de polymorphisme de code à la compilation

Sommaire

5.1 Introduction

Dans le chapitre précédent, nous avons proposé une approche pour appliquer automatiquement un schéma de masquage. Comme nous l'avons montré, l'application automatisée du masquage booléen d'ordre 1 dans le middle-end avec un modèle de fuite en valeur se heurte à plusieurs problèmes. Tout d'abord, les passes du back-end comme la sélection d'instructions peuvent affecter la contre-mesure. Ensuite, nous avons pû constater que le modèle de fuite en valeur n'est pas toujours suffisant. Un modèle de fuite en transition nécessite quant à lui de connaître la micro-architecture de la plate-forme cible. Enfin, les surcoûts en temps d'exécution sont très variables d'un code à l'autre et peuvent être très importants, et le compromis sécurité / performance n'est pas configurable.

Dans ce chapitre, nous nous intéressons aux contre-mesures de dissimulation, pour déterminer si l'une de ces contre-mesures pourrait fournir une alternative permettant à la fois une application automatisée sur un code non sécurisé au sein du compilateur sans risque que la compilation n'altère la contre-mesure, des surcoûts moins variables d'un code à l'autre, et un compromis entre sécurité et performance configurable pour s'adapter aux contraintes d'une application et d'une plate-forme.

Nous donnons en section 5.2 les raisons qui ont justifié le choix de la contre-mesure de polymorphisme de code que nous avons considéré, puis une présentation générale de notre approche en section 5.3. Nous présentons le détail des transformations de code effectuées en section 5.3.3, puis la gestion de la mémoire en section 5.4. Enfin, nous présentons l'implémentation de notre approche dans LLVM en section 5.5, et les résultats en section 5.6.

5.2 Motivation et contributions

Nous avons vu en section 2.3.1 les différents principes de dissimulation.

Afin de répondre à des exigences de performance, sécurité, et mémoire variées, nous souhaitions une contre-mesure combinant plusieurs principes (mélange d'opérations, insertion de délais, etc...) pour qu'un développeur puisse adapter l'application de la contre-mesure à ses contraintes. La contre-mesure de polymorphisme de code développée par [Couroussé et al., 2016] fournis cette combinaison de transformations, et a l'avantage de ne pas pouvoir être altérée par les optimisations du compilateur. De plus, elle est suffisamment légère pour être utilisée sur des systèmes embarqués contraints.

Cependant, l'application de cette contre-mesure n'était pas automatisée et requérait l'emploi d'un DSL. De plus, il n'existait pas de stratégie pour allouer la zone mémoire chargée d'accueillir le code généré dynamiquement, et cette zone devait rester disponible en écriture et en exécution systématiquement.

Dans cette thèse, nous avons automatisé à la compilation l'application de cette contre-mesure. Nous avons apporté une approche pour gérer les droits mémoires de manière à garantir que seul un code légitime puisse écrire dans une zone mémoire qui pourra être exécutée. Nous montrons également comment allouer la mémoire pour que la taille

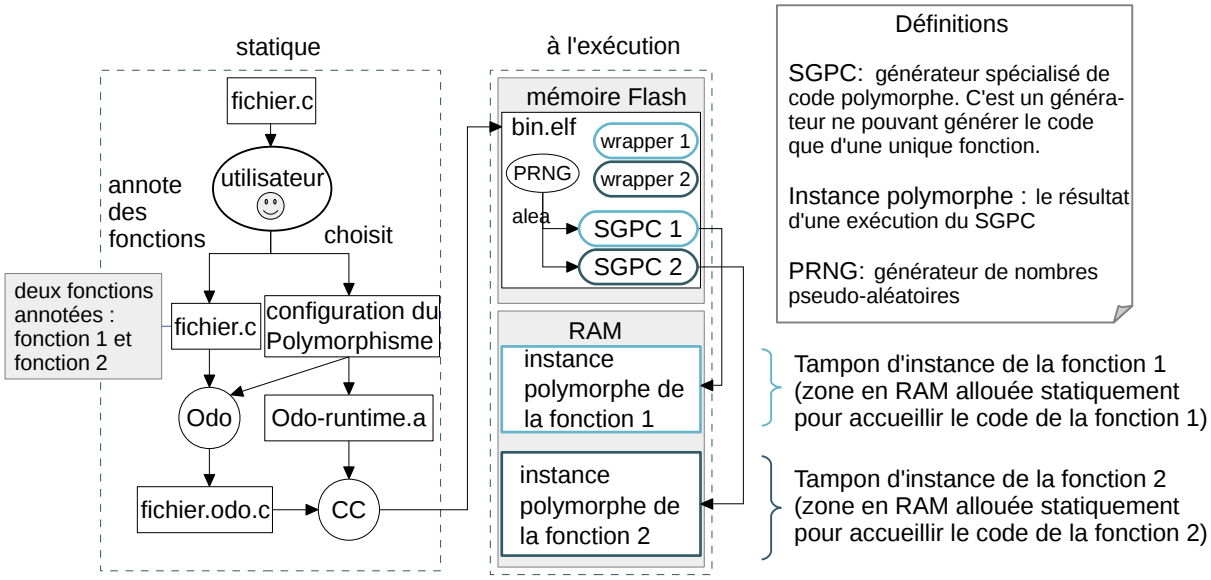

FIGURE 5.1 – Vue générale du flot de compilation pour l'application automatisée du polymorphisme de code. Les binaires des fonctions polymorphes sont générées à l'exécution par leurs SGPCs respectifs.

d'allocation soit réaliste, tout en gérant les problèmes de dépassement de tampon. Enfin, nous avons étendu les transformations proposées par [Couroussé et al., 2016] pour obtenir de meilleurs compromis entre sécurité et performance.

5.3 Application automatique du polymorphisme

Dans cette section, nous présentons l'approche développée dans ce travail de thèse, implémentée dans un outil appelé Odo. Odo permet de protéger automatiquement toute fonction contre les attaques par canal auxiliaire avec du polymorphisme de code. L'idée clé de notre approche est de baser la génération de code polymorphe sur des générateurs de code spécialisés, qui ne peuvent générer du code que pour la fonction à sécuriser. Ceux-ci sont embarqués dans le binaire de l'application, et regénèrent un code différent régulièrement à l'exécution, ce qui donne la propriété de polymorphisme au code.

5.3.1 Vue générale du flot d'application de la contre-mesure

Cette section donne une vue générale de notre approche. Le flot d'application de la contre-mesure est illustré en figure 5.1. Notre approche s'appuie sur la compilation pour générer automatiquement un générateur spécialisé pour chaque fonction spécifiée par le développeur (partie "statique" sur la figure 5.1). Nous mettons aussi à profit le flot de compilation statique pour recueillir des informations et optimiser le code produit par le générateur à l'exécution. La phase de compilation statique permet également d'effectuer une allocation statique de mémoire d'une taille adaptée au code à générer.

Listing 5.1 – Fichier C original annoté par l'utilisateur

```
#pragma odo_polymorphic
int f_critical(int a, int b) {
  int c = a^b;
  a = a+b;
  a = a % c;
  return a;
}
```

Listing 5.2 – Fichier C généré par Odo lorsque les options de polymorphisme sont désactivées

```
code code_f[CODE_SIZE]; /* tampon d'instance */
void SGPC_f_critical() { /* generateur */
  interrupt_rm_X_add_W(code_f);
  reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
  push_T2_callee_saved_registers();
  eor_T2(r[4], r[1], r[0]);
  add_T2(r[0], r[1], r[0]);
  sdiv_T2(r[1], r[0], r[4]);
  mls_T2(r[0], r[1], r[4], r[0]);
  pop_T2_callee_saved_registers();
  interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) { /* wrapper */
  if (SHOULD_BE_REGENERATED())
    SGPC_f_critical();
  return code_f(a, b);
}
```

Lors de l'exécution (partie "à l'exécution" sur la figure 5.1), les générateurs spécialisés utilisent les informations collectées statiquement et plusieurs transformations de code pour générer de manière efficace un code variable. Certaines de ces transformations ont déjà démontré leur efficacité contre les attaques par canal auxiliaire : le mélange des registres, le mélange des instructions, les variantes sémantiques et l'insertion d'instructions de bruit [Couroussé et al., 2016]. Par rapport à [Couroussé et al., 2016], les transformations de mélange des registres, de variantes sémantiques et d'insertion d'instructions de bruit ont été partiellement modifiées : le mélange des registres se base sur une allocation de registres statiques et une permutation aléatoire à l'exécution, et non pas une allocation aléatoire à l'exécution, des variantes sémantiques ont été ajoutées pour les lectures et écritures mémoires, et un nouveau modèle de probabilité appelé `high-var` a été ajouté pour l'insertion d'instructions de bruit. Les générateurs spécialisés peuvent également utiliser une nouvelle transformation, qui est une contribution de cette thèse, appelée bruit dynamique pour introduire de la variabilité entre deux exécutions consécutives d'un même code généré (section 5.3.3). Comme chaque transformation peut être activée/désactivée ou paramétrée, l'approche proposée est très configurable.

Le flot d'application de la contre-mesure est présenté dans la figure 5.1. L'utilisateur commence par annoter les fonctions à sécuriser par polymorphisme. Ensuite, il choisit une configuration de polymorphisme, c'est à dire quelles transformations il souhaite avoir à l'exécution, et leurs paramètres. Le fichier C annoté (`fichier.c` dans la figure 5.1) est ensuite compilé par notre outil, Odo, vers un autre fichier C (`fichier.odo.c` dans la figure 5.1). Dans ce nouveau fichier, le code de chaque fonction à sécuriser a été remplacé par : (1) un *wrapper* qui s'interface avec le code externe à la fonction polymorphe, et (2) un générateur dédié. Le wrapper gère les appels vers le générateur dédié et vers le code produit. Le wrapper rend la transformation transparente au reste du code : la fonction reste appelée de manière identique. Le wrapper se charge ainsi d'appeler le code qui a été produit par le générateur spécialisé avec les bons arguments et de retourner sa valeur de retour. Le code produit par le générateur spécialisé est appelé *instance polymorphe*.

Le générateur dédié de chaque fonction polymorphe est généré automatiquement par Odo. L'implémentation de ce générateur est créée à partir du code initial de la fonction et de la configuration polymorphe choisie par l'utilisateur. Nous appelons ces générateurs *SGPC* (*Specialized Generator of Polymorphic Code* : générateur spécialisé de code polymorphe) par la suite.

Les listings 5.1 et 5.2 montrent des exemples des fichiers fichier.c et fichier.odo.c de la figure 5.1. Dans le listing 5.2, la fonction `f_critical` correspond au wrapper : elle appelle le générateur `SGPC_f_critical`, puis le code généré dans la zone mémoire `code_f`. Cette zone mémoire, appelée *tampon d'instance*, est réservée statiquement pour héberger le code de la fonction polymorphe générée à l'exécution.

Le fichier C produit est ensuite compilé par le compilateur usuel de la plate-forme cible (CC sur la figure 5.1) avec la bibliothèque `Odo-runtime`, qui fournit le support pour l'architecture et les outils de transformation du code, en un fichier binaire qui est ensuite chargé sur la plate-forme. Dans la figure 5.1, deux fonctions sont annotées dans le code source. Comme les générateurs sont spécialisés pour chaque fonction polymorphe, il y a un SGPC et un tampon d'instance pour chacune d'elles. Cependant, le coût en taille de

code est sous-linéaire avec le nombre de fonctions polymorphes, puisque certaines parties des codes des SGPCs sont mutualisées dans la librairie `Odo-runtime`.

Lors de l'exécution, le SGPC est appelé par le wrapper chaque fois qu'une nouvelle instance polymorphe de code doit être générée dans le tampon d'instance. Les appels réguliers au SGPC donnent la propriété du polymorphisme à la fonction : son code change à chaque appel. La fréquence de regénération peut être contrôlée par l'utilisateur. Nous appelons période de regénération, notée ω, le nombre d'exécutions consécutives de la même instance polymorphe avant une nouvelle regénération. Lorsque le SGPC est appelé, les permissions de son tampon d'instance sont basculées d'exécution seule à écriture seule au début de la génération de code, puis d'écriture seule à exécution seule à la fin de la génération de code. Ceci garantit que le tampon d'instance n'est jamais accessible en écriture et en exécution simultanément. La section 5.4.3 présente plus en détails la gestion de ces permissions.

5.3.2 Génération des générateurs spécialisés de code polymorphe

Dans cette section, nous présentons comment Odo génère un SGPC pour chaque fonction choisie par l'utilisateur. Les transformations de code mises en œuvre à l'exécution pour obtenir du code polymorphe sont présentées dans la section 5.3.3.

Odo est un compilateur source à source. Il effectue une compilation avec une émission de code modifiée pour émettre du code C dépendant de l'architecture au lieu d'émettre des instructions machine. Le processus de compilation permet d'obtenir une suite d'instructions machine correspondant à la fonction à sécuriser. À partir de cette suite d'instructions, la passe d'émission de code génère le SGPC dédié à cette fonction. Le code d'un SGPC est composé a minima d'une séquence d'appels à des émetteurs d'instruction machine. L'exécution de cette séquence minimale émet en mémoire dans le tampon d'instance la séquence d'instructions machine produite par la compilation statique du code d'origine.

Dans Odo, la génération des SGPCs se fait grâce à un nouveau back-end, qui est entièrement identique au back-end de l'architecture cible, à l'exception de la passe d'émission de code. Comme seule cette passe diffère du back-end, la suite d'instructions à partir de laquelle un SGPC est construit bénéficie de toutes les optimisations préalables du compilateur. Au lieu d'émettre du code machine, la passe d'émission émet le code C des SGPCs correspondant aux fonctions annotées. Les SGPCs produits lorsque le polymorphisme est activé sont similaires à ceux obtenus avec le polymorphisme désactivé : du code supplémentaire (décrit en section 5.3.3) est simplement ajouté en fonction des transformations activées afin de permettre l'exécution de ces transformations lors d'une génération de code.

Pour comprendre le code du SGPC obtenu dans le listing 5.2, le listing 5.3 présente la suite d'instructions assembleur générée par LLVM lors d'une compilation de la fonction `f_critical` du listing 5.1 pour ARM Thumb. Le SGPC créé pour cette fonction dans le listing 5.2 est composé d'une suite d'appels de fonctions à la bibliothèque `Odo-runtime`. Chaque appel correspond à une des instructions du listing 5.3. Des fonctions permettant l'encodage des différentes instructions machine sont disponibles dans cette bibliothèque.

Listing 5.3 – Séquence d'instructions ARM générée par LLVM pour `f_critical`

```
f_critical:           @ r0=a; r1=b
  push r4, pc
  eor r4, r1, r0      @ r4←r1^r0
  add r0, r1, r0      @ r0←r1+r0
  sdiv r1, r0, r4     @ r1←⌊r0/r4⌋
  mls r0, r1, r4, r0  @ r0←r0−r1×r4
  pop r4, lr          @ return r0
```

Ici, nos exemples utilisent les jeux d'instructions ARM Thumb1 et Thumb2. Par exemple, dans la figure 5.2, l'appel `eor_T2(r[4],r[1],r[0])` écrit dans le tampon d'instance l'instruction machine `eor r4,r1,r0`. Le suffixe `_T2` indique que l'encodage Thumb2 est utilisé. Tous les émetteurs d'instructions machine définis dans la bibliothèque `Odo-runtime.a` (figure 5.1) prennent pour paramètres les opérandes d'instructions, à savoir les noms de registres physiques et/ou les valeurs constantes à utiliser, comme ce serait le cas pour les instructions machine classiques. Cette paramétrisation permet au SGPC de changer les opérandes d'une génération à l'autre. Par exemple, `r[4]` peut faire référence à un registre physique différent d'une génération à l'autre.

Comme déjà évoqué, les permissions d'accès du tampon d'instance sont modifiées en début et fin du code du SGPC. Cela est réalisé via des interruptions. Ceci est illustré dans le listing 5.2 par les appels à `interrupt_rm_A_add_B`. Dans la pratique, ces interruptions sont expansées (*inlinées*) à l'aide de primitives assembleur. Les mécanismes permettant la gestion des permissions mémoire sont présentés en détail dans la section 5.4.3.

5.3.3 Transformations de code à l'exécution et leur génération

Dans cette section, nous présentons les transformations de code utilisées pour générer un code différent chaque fois que le SGPC est appelé. Nous expliquons comment le flot de compilation est exploité pour aider les transformations de code au moment de l'exécution sans nécessiter une analyse de code coûteuse, notamment pour la gestion des registres libres. Nous présentons également comment le code du SGPC généré par Odo diffère lorsque ces transformations sont activées.

Cinq transformations différentes peuvent être utilisées par les SGPCs pour faire varier le code des instances polymorphes :

1. le mélange des registres, qui est une permutation aléatoire parmi les registres sauvegardés par l'appelé (*callee-saved*),
2. le mélange des instructions, qui consiste à émettre dans un ordre aléatoire des instructions indépendantes,
3. des variantes sémantiques, qui visent à remplacer aléatoirement certaines instructions par une séquence d'instructions conduisant au même résultat,
4. des instructions de bruit, qui sont des instructions inutiles insérées entre les instructions originales de la fonction,

5. du bruit dynamique, qui consiste en une séquence d'instructions de bruit précédée par un saut aléatoire, afin de que le nombre d'instructions exécutées varie à chaque exécution.

Ces transformations peuvent être activées indépendamment les unes des autres.

Listing 5.4 – Fichier C généré par Odo lorsque toutes les transformations sont activées

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
  interrupt_rm_X_add_W(code_f);
  reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
  shuffle_regs(r);
  push_T2_callee_saved_registers();
  gennoise();
  variant_eor_T2(r[4], r[1], r[0]);
  gennoise();
  add_T2(r[0], r[1], r[0]);
  gennoise();
  sdiv_T2(r[1], r[0], r[4]);
  gennoise();
  mls_T2(r[0], r[1], r[4], r[0]);
  gennoise();
  pop_T2_callee_saved_registers();
 interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
  if (SHOULD_BE_REGENERATED())
    SGPC_f_critical();
  return code_f(a, b);
}
```

Le listing 5.4 montre la sortie d'Odo pour l'entrée du listing 5.1, lorsque toutes les transformations sont activées. Le listing 5.5 est un exemple d'instance polymorphe pouvant être générée par le SGPC à l'exécution.

Mélange des registres. Contrairement à ce qui était proposé précédemment avec l'approche COGITO [Couroussé et al., 2016] où l'allocation aléatoire des registres était effectuée au moment de l'exécution, avec Odo l'allocation des registres est faite statiquement par le compilateur. Les SGPCs font une permutation aléatoire entre les registres généralistes (*general purpose register*) sauvegardés par l'appelé (`r4-r11` pour l'architecture ARM Thumb). Cette permutation est effectuée au début de chaque génération de code (`shuffle_regs` dans le listing 5.4).

Afin de permettre cette transformation, la sélection d'instructions faite dans Odo sélectionne uniquement des instructions qui encodent les registres sur suffisamment de bits pour échanger n'importe quel registre avec n'importe quel autre. Sur ARM Thumb, il n'utilise donc que des instructions ayant 4 bits pour l'encodage des registres. Un exemple

Listing 5.5 – Exemple d'un code assembleur ARM généré par `SGPC_f_critical` avec toutes les transformations de polymorphisme activées. `r5` est utilisé à la place de `r4` à cause du mélange de registres, une variante sémantique est insérée pour le OU EXCLUSIF (`eor`), plusieurs instructions de bruit sont insérées, et l'ordre de deux instructions indépendantes a été inversé.

```
f_critical:
  push r5, r7, r8, r9, lr
  eor r5, r1, #42              @ variantes semantiques
  eor r5, r5, r0
  add r0, r1, r0               @ mélange d'instructions
  eor r5, r5, #42              @ mélange d'instructions
  sub r9, #127                 @ instruction de bruit
  sdiv r1, r0, r5
  add r7, r9, #5               @ instruction de bruit
  eor r8, #3                   @ instruction de bruit
  mls r0, r1, r5, r0[18]
  pop r5, r7, r8, r9, pc
```

de mélange de registres est illustré dans le listing 5.5 : dans cet exemple, le registre `r5` est utilisé à la place du registre `r4`, qui était utilisé dans le code original (listing 5.3).

Mélange d'instructions. Cette transformation vise à réordonnancer dans chaque bloc de base les instructions indépendantes avant leur émission dans le tampon d'instance. Les instructions machine émises sont d'abord stockées dans un tampon temporaire, appelé *tampon de mélange*, d'une taille configurable (32 instructions dans nos expériences). Chaque instruction machine est associée aux registres qu'elle définie et utilise. Avant qu'une instruction ne soit ajoutée au tampon de mélange, `Odo-runtime` effectue une analyse arrière de la chaîne de dépendances *use-def*, à partir de la dernière instruction dans le tampon de mélange, pour calculer la liste des emplacements d'insertion possibles. Un emplacement est ensuite sélectionné aléatoirement parmi les emplacements d'insertion possibles. Le tampon de mélange est vidé dans le tampon d'instance à la fin de chaque bloc de base. Il est également vidé lorsqu'il est plein. La transformation de mélange d'instructions est effectuée de manière transparente par `Odo-runtime`, le code généré par Odo pour le SGPC est identique que la transformation soit active ou non. Cette transformation intervient après l'ajout des instructions de bruit et le choix des variantes sémantiques, elle permet donc de mélanger les instructions résultantes avec les instructions originales du code. Ainsi, dans l'exemple du listing 5.5, une instruction faisant partie d'une variante sémantique est mélangée avec une des instructions originales du code : la position du `add r0, r1, r0` et du deuxième `eor r5, r5, #42` est inversée.

Variantes sémantiques. Certaines instructions peuvent être remplacées par une suite d'instructions appelée variante sémantique qui permet d'obtenir le même résultat sans modifier les autres registres vivants (cela inclut les registres de statut, qui indiquent notamment la présence d'une retenue ou d'un dépassement de capacité). `Odo-runtime` fournit actuellement des variantes sémantiques pour les instructions qui sont fréquemment

utilisées dans les algorithmes de chiffrement pour manipuler des données sensibles : les instructions appartenant aux familles `eor`, `sub`, `load` et `store`. Les variantes des familles `eor` et `sub` étaient déjà disponibles dans [Couroussé et al., 2016]. On peut facilement ajouter d'autres variantes sémantiques. Actuellement, chaque instruction pour laquelle il existe une variante peut être remplacée par 1 à 5 instructions.

Odo génère des appels de fonctions spécifiques à la bibliothèque `Odo-runtime` pour l'émission de ces instructions lorsque les variantes sémantiques sont activées. Dans le listing 5.1, les appels en gras vert sont chargés de l'émission des instructions ayant des variantes sémantiques. Lors de l'exécution, le SGPC émet le code binaire d'une variante choisie aléatoirement parmi celles disponibles (dont l'instruction initiale). Par exemple, l'appel `variant_eor_T2(r[4], r[1], r[0])` du listing 5.4 peut générer l'instruction originale `eor r4, r1, r0` ou, par exemple, une séquence `eor rX, r0, #rand ; eor r4, r1, #rand ; eor r4, r4, rX` (comme indiqué dans listing 5.5) où `rX` est un registre libre choisi au hasard et `#rand` est une constante aléatoire.

Les variantes sémantiques pour les instructions arithmétiques (par exemple, `sub` et `xor`) sont basées sur des équivalences arithmétiques, les variantes pour les écritures mémoire de mots ou demi-mots décomposent l'écriture mémoire en plusieurs écritures mémoire d'octets ou de demi-mots, et les variantes pour les lectures mémoire utilisent des lectures mémoire non alignées ainsi que des séquences de lectures mémoire d'octets et de demi-mots.

Instructions de bruit. Nous appelons instructions de bruit des instructions fonctionnellement inutiles qui sont générées entre des instructions utiles. L'insertion des instructions de bruit est effectuée par les appels à `gennoise` (en bleu italique dans le listing 5.4). De la même manière que pour les autres transformations, Odo ne génère ces appels dans le code du SGPC que lorsque cette transformation de code est activée.

Le profil observable par canal auxiliaire des instructions de bruit doit être aussi proche que possible du profil des instructions utiles, afin que l'attaquant ne puisse pas les distinguer sur des mesures effectuées par canal auxiliaire [Durvaux et al., 2013]. Nous avons donc choisi d'insérer des instructions qui sont souvent utilisées dans les programmes, comme l'addition, la soustraction, le OU EXCLUSIF, et la lecture mémoire. L'utilisateur peut spécifier une plage d'adresses particulière pour les lectures mémoire, pour obtenir des lectures mémoire aléatoires sur la SBox de l'AES par exemple. Une petite table aléatoire statique est utilisée lorsque l'utilisateur ne spécifie pas de plage d'adresses.

Un modèle de probabilité contrôle l'insertion d'instructions de bruit. `Odo-runtime` propose actuellement deux modèles, tous les deux configurables, présentés dans la table 5.1. Le nombre p est la probabilité d'insertion d'une ou plusieurs instructions de bruit, et $P[X = i]$ représente la probabilité d'insertion de i instructions de bruit. Le paramètre N contrôle le nombre maximum d'instructions de bruit qui peuvent être insérées en même temps entre deux instructions originales.

Le premier modèle, appelé `low-var`, suit une loi de probabilité uniforme, qui est combinée avec un tirage de probabilité p. Sa variance est faible, ce qui implique que le temps d'exécution global de la fonction restera toujours relativement proche de la moyenne

TABLE 5.1 – Modèles de probabilités qui contrôlent le nombre d'instructions de bruit à insérer entre 2 instructions originales. Les deux modèles sont configurables par l'utilisateur. Le modèle `high-var` permet d'obtenir des fortes variances tout en conservant une moyenne contenue.

modèle `low-var`	modèle `high-var`
$P[X=0] = 1-p$ $\forall i \in [1,N[,\ P[X=i] = \frac{p}{N}$	$P[X=0] = 1-p$ $\forall i \in [0,N[,\ P[X=2^i] = p \times 2^{-(i+1)}$ $P[X=2^N] = p \times 2^{-N}$

théorique. Dans ce modèle, le paramètre N correspond directement au nombre maximum d'instructions de bruit qui peuvent être insérées entre deux instructions originales.

Le deuxième modèle, appelé `high-var`, a été conçu pour avoir une variance beaucoup plus élevée et une moyenne comparable. Il est basé sur une loi de probabilité binomiale. Le nombre d'instructions insérées, s'il n'est pas nul, est de 2 à la puissance du nombre obtenu par la loi binomiale. Dans ce modèle, le nombre maximum d'instructions de bruit qui peuvent être insérées entre deux instructions originales est de 2^N.

Les variables N et p ainsi que le modèle peuvent être choisis par l'utilisateur lors du choix d'une configuration de polymorphisme.

La mise à disposition de ces deux modèles différents, tous les deux configurables, rend la transformation d'insertion d'instructions de bruit très flexible. La valeur moyenne d'un modèle a un impact sur le temps d'exécution global et doit donc être maintenue basse pour des raisons de performance, alors qu'une variance élevée augmente la complexité des attaques [Mangard et al., 2007, Tunstall et Benoit, 2007]. Le modèle à faible variabilité est intéressant pour les applications soumises à de fortes contraintes de performance, pour lesquelles le temps d'exécution ne doit pas trop varier, sinon le modèle à variabilité élevée est préférable.

À chaque insertion d'une instruction de bruit, le SGPC choisit aléatoirement une instruction parmi les instructions `add`, `sub`, `eor` et `load`. Ensuite, il choisit au hasard les opérandes et choisit un registre mort pour le registre destination.

Bruit dynamique. Nous appelons bruit dynamique une séquence d'instructions inutiles comportant un branchement aléatoire. Celui-ci permet de faire varier dynamiquement le nombre d'instructions inutiles exécutées en sautant un nombre aléatoire des instructions de bruit de la séquence. Ce mécanisme dynamique permet d'obtenir une exécution variable d'une exécution à l'autre même sans regénération du code.

Cette transformation permet de réduire les contraintes de fréquence de regénération imposées par les exigences de sécurité, pour que l'approche puisse être utilisée même sur des systèmes qui ne peuvent se permettre de regénérer le code trop fréquemment : en réduisant la fréquence de regénération le coût de génération peut être amorti. De plus, elle permet de décorréler partiellement le code exécuté du chemin emprunté lors de la génération pour réduire le risque qu'un attaquant, qui saurait suivre l'exécution du générateur de code par canal auxiliaire, soit à même de connaître le code des instances

Listing 5.6 – Exemple d'une séquence d'instructions produite avec du bruit dynamique. La valeur de `Rrand` varie tout au long de l'exécution.

```
@ R7 = number of bytes to be skipped
and R7, Rrand, 3 ≪ 2
@ 2 must be added to R7 since, during the add R7 PC R7
@ instruction, @PC points to the bx inst whose size is
@ 2 bytes.
@ Also, the least significant bit must be set in the address
@ (Thumb mode) which gives an offset of 3
add R7, R7, 3
add R7, PC, R7
@ R7 = address of targeted noise inst
bx R7 @ jump into the sequence
add R7, R8, #41 @ 4 noise instructions
xor R8, R10, R7
load R7, R4, #34
add R8, R8, #101
```

polymorphes exécutées, et donc de filtrer facilement les traces ensuite en enlevant les échantillons correspondant à des cycles où des instructions de bruit sont exécutées par exemple.

L'insertion de bruit dynamique par le SGPC pendant la génération du code d'exécution suit la même procédure que l'insertion des instructions de bruit : les mêmes modèles de probabilités sont utilisés pour déterminer si des instructions de bruits sont insérées ou non, et si oui combien. Cependant, au lieu de choisir les instructions de bruit uniquement dans l'ensemble {`load`, `add`, `sub`, `xor`}, le SGPC choisit au hasard d'insérer une addition, une soustraction, une lecture mémoire, un OU EXCLUSIF, ou une séquence de bruit dynamique dont le nombre d'instructions est fixé mais configurable. Chaque fois que le SGPC est exécuté, différentes séquences de bruit dynamique sont générées, à des endroits variables dans le code de l'instance polymorphe générée.

La taille des sauts associés à chaque séquence de bruit dynamique d'une instance polymorphe doit être déterminée en peu d'instructions et doit varier aléatoirement. À cette fin, un registre est réservé pour stocker un nombre aléatoire qui est utilisé pour calculer une taille de saut aléatoire à chaque exécution de chaque séquence de bruit dynamique pendant l'exécution d'une instance polymorphe. Si le nombre d'instructions de bruit dans une séquence de bruit dynamique est une puissance de deux, la taille du saut peut être calculée efficacement en masquant avec une valeur immédiate la valeur aléatoire contenue dans le registre réservé, en utilisant un ET booléen. Le listing 5.6 montre un exemple d'une telle séquence avec 4 instructions de bruit. La partie avant le `bx` détermine la taille du saut, et 4 instructions de bruit choisies au hasard sont générées après le `bx`.
Le nombre d'instructions de bruit dans la séquence de bruit dynamique générée par le SGPC est configurable par l'utilisateur, et peut prendre plusieurs valeurs différentes à l'intérieur d'une fonction. À titre d'exemple, dans nos expériences, nous avons sélectionné une taille de séquence de bruit dynamique comportant 32 instructions de bruit au début et à la fin de la fonction (respectivement pour les 32 premières instructions utiles de la fonction et les 32 dernières), et de 4 instructions de bruit au milieu. La raison de ce choix

est la nécessité d'une plus grande variabilité au début et à la fin de la fonction, pour que la variabilité temporelle des instructions proches de points de synchronisation exploitables par l'attaquant augmente suffisamment vite.

Le registre qui contient les valeurs aléatoires est géré comme suit. Pour chaque fonction polymorphe, une place dédiée est réservée en mémoire pour contenir une valeur de graine qui varie d'une exécution à l'autre. Au début de chaque exécution d'une instance polymorphe, la valeur stockée de la graine est chargée pour initialiser un PRNG rapide. Le PRNG rapide est alors utilisé pour définir une nouvelle valeur aléatoire pour le registre réservé. Ensuite, tout au long de l'exécution, la valeur du registre peut être mise à jour par d'autres instructions de bruit (par exemple, une instruction d'addition de bruit insérée par le SGPC peut ajouter un immédiat aléatoire à la valeur du registre). Ceci fait changer la valeur du registre tout au long de l'exécution de la fonction. Enfin, en fin d'exécution de la fonction, la valeur du registre est stockée en mémoire, elle sera utilisée comme graine pour le PRNG rapide au début de l'exécution suivante de cette fonction.

Allocation de registres pour les instructions de bruit et les variantes sémantiques. L'insertion d'instructions de bruit et l'utilisation de variantes sémantiques peuvent nécessiter des registres supplémentaires. Une analyse de vivacité effectuée statiquement par le compilateur est utilisée pour aider le SGPC à allouer ces registres.

Lors de la compilation des SGPCs, Odo effectue d'abord une allocation statique des registres en ne tenant pas compte du polymorphisme. Ensuite, Odo effectue une analyse arrière de la vivacité des registres juste avant l'émission du code du SGPC. Pendant l'émission du code du SGPC, Odo émet des appels de fonction supplémentaires tout au long du code du SGPC afin de tracer les informations de vivacité. Ces appels indiquent quels registres sont libres (ou non) entre deux instructions utiles. Dans notre exemple de SGPC dans le listing 5.4, l'addition utilise `r1`, donc `r1` est vivant juste avant cette instruction. L'instruction `sdiv` définit ensuite `r1` sans l'utiliser. Ainsi `r1` est mort avant l'instruction `sdiv` et vivant juste après car utilisé par le `mls`. En conséquence, `r1` est libre d'être utilisé (écrit) entre l'instruction `add` et `sdiv`.

Grâce aux appels de fonction insérés pour transmettre les informations de vivacité, le SGPC connaît la vivacité de chaque registre à tout moment du programme. Il peut ainsi sélectionner des registres libres lorsqu'il en a besoin. Tous les registres alloués aux instructions de bruit et aux variantes sémantiques sont choisis de manière aléatoire parmi les registres libres. Comme les registres supplémentaires utilisés dans la séquence d'instructions résultant de ces transformations de polymorphisme sont morts immédiatement après leur utilisation, les résultats de l'analyse statique de la vivacité restent corrects quels que soient les registres utilisés pour les instructions de bruit ou les variantes sémantiques. Aucune analyse de vivacité n'est donc réalisée à l'exécution.

Calcul des sauts pour les branchements. Puisque l'insertion d'instructions de bruit et l'utilisation de variantes sémantiques font varier la taille du code, l'adresse cible des sauts et son encodage doivent être calculés dynamiquement par le SGPC. Le SGPC calcule l'écart entre branchement et sa cible lors de la génération, et choisit pour chacun

d'eux l'encodage de l'instruction de branchement le plus petit qui permette d'encoder correctement le branchement.

Nombre théorique d'instances polymorphes différentes. Afin d'avoir une idée du nombre théorique N_v d'instances polymorphes différentes qu'il est possible d'obtenir avec notre approche, nous proposons ici une sous-estimation de ce nombre en fonction de la variable N des modèles de probabilité de la table 5.1. En ne prenant en compte que l'insertion d'instructions de bruit, avec le modèle `low-var`, on a : $N_v \geq (\sum_{i=0}^{N} 4^i)^{nombre\_instructions-1}$. Le chiffre 4 vient du fait que les instructions de bruit sont sélectionnées parmi 4 instructions différentes (`add, sub, eor, load`). En choisissant $N = 4$, cette formule nous donne $N_v \geq 341^{nombre\_instructions-1} > 6 \times 10^{22}$ pour un code de 10 instructions seulement, et environ 10^{704} pour un code de 278 instructions comme l'implémentation T-table de l'AES utilisée dans notre évaluation expérimentale (section 5.6). Ce nombre est une sous-estimation du nombre de variantes puisqu'il ne considère que le bruit classique (le bruit dynamique et les autres transformations ne sont pas prises en compte), et il ne prend pas en compte les variations à l'intérieur même des instructions de bruit (les valeurs immédiates des instructions de bruits étant choisies aléatoirement).

5.4 Gestion mémoire

Cette section présente comment nous gérons les tampons d'instances afin de rendre notre approche utilisable dans des systèmes embarqués avec des ressources mémoire réduites. Plus précisément, la gestion de la mémoire doit prendre en compte plusieurs contraintes :

1. l'utilisation d'instructions de bruit et de variantes sémantiques fait varier la taille du code généré d'une génération à l'autre,
2. les plates-formes ciblées peuvent être des systèmes embarqués contraints sans gestion dynamique de la mémoire (c'est-à-dire sans `malloc`) et avec une quantité réduite de mémoire,
3. il ne doit jamais y avoir de débordement du tampon d'instance,
4. le tampon d'instance doit être accessible en écriture pendant la génération du code et être exécutable pendant l'exécution, mais les deux autorisations doivent être exclusives.

La seconde contrainte rend une réponse à la première et à la troisième contraintes plus difficile : l'absence d'allocation dynamique de mémoire empêche d'allouer un tampon d'instance à chaque génération de code. De plus, il n'est pas acceptable d'allouer systématiquement un tampon d'instance de la plus grande taille de code possible, car ce serait un énorme gaspillage de mémoire ; cela rendrait également l'approche inutilisable sur les systèmes très limités en mémoire. Nous montrons dans la suite que la probabilité de rencontrer le pire cas est extrêmement faible lorsque le nombre d'instructions de la fonction originale est raisonnablement élevé. Ainsi, une grande partie du tampon d'instance alloué en considérant le pire cas serait la plupart du temps inutilisée. Cependant, en allouant une quantité de mémoire plus petite, des débordements de tampon d'instance deviennent possibles, ce qui menace à la fois la fonctionnalité et la sécurité de la plate-forme entière.

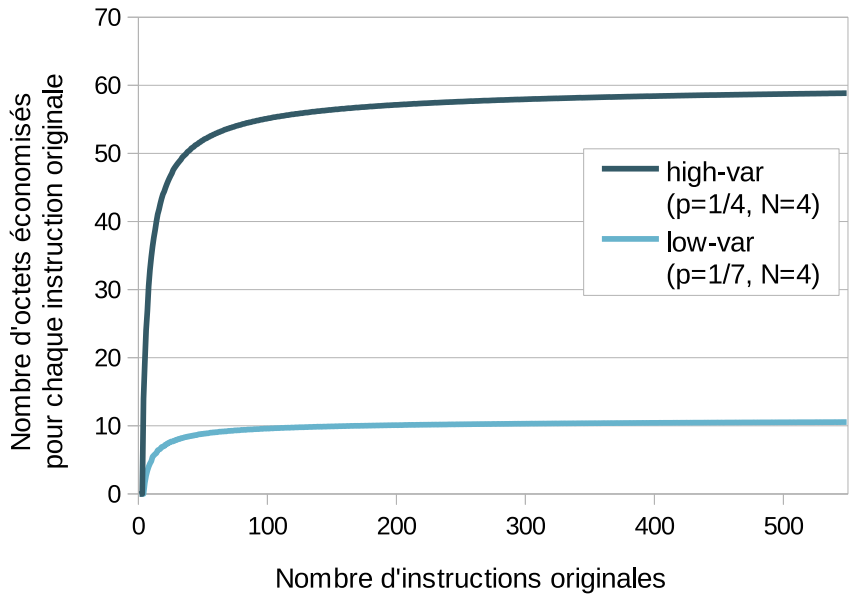

FIGURE 5.2 – Différence entre notre politique d'allocation (avec un seuil à 10^{-6}) et une politique de pire cas, en nombre d'octets par instruction. Pour le modèle `high-var` considéré sur l'architecture ARMv7M, on note un écart de 58 octets par instruction environ. Cela représente une différence de 29ko pour un code de 500 instructions.

Pour y remédier, nous exploitons la connaissance statique des instructions machine utiles de la fonction pour :

- limiter la taille du tampon d'instance en allouant statiquement une taille réaliste (section 5.4.1),
- empêcher les débordements de tampon en adaptant la génération de code lorsqu'un débordement possible est identifié (section 5.4.2). Pour cela, l'insertion des instructions de bruit s'adapte dynamiquement à l'espace disponible et à la taille des instructions utiles restant à générer, ce qui permet d'éviter qu'un dépassement de tampon ne survienne.
- calculer une taille de tampon tout en contrôlant la probabilité maximale d'avoir à limiter l'insertion d'instructions de bruit faute de place suffisante en mémoire,
- garantir que seul l'unique SGPC légitime peut écrire dans le tampon d'instance, et qu'aucune autre partie du programme, ni aucun autre programme, ne peut le faire (section 5.4.3). Pour ce faire, une gestion dédiée des permissions mémoire de chaque tampon d'instance est mise en place, en tirant profit de la spécialisation des SGPCs et de l'allocation statique des tampons d'instance.

5.4.1 Allocation des tampons d'instance

La taille d'un tampon d'instance est calculée lors de la génération du SGPC associé, en calculant la taille requise pour les instructions utiles et la taille requise pour les instructions de bruit. Pour les instructions utiles, Odo calcule la somme S_u des tailles des instructions utiles, en considérant les plus grands équivalents sémantiques lorsque des variantes sémantiques sont disponibles. Pour les instructions de bruit, Odo calcule une

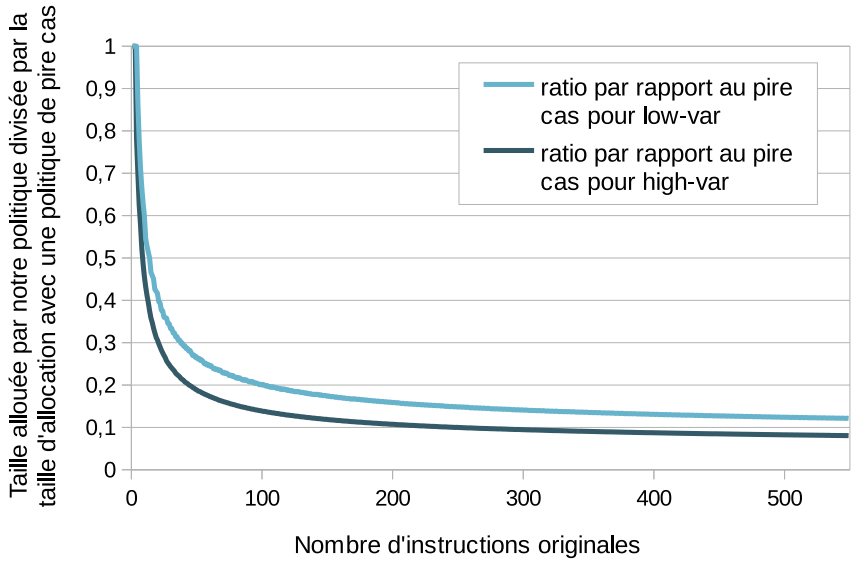

FIGURE 5.3 – Ratio entre la taille allouée avec notre politique d'allocation (avec un seuil à 10^{-6}) divisée par la taille allouée par une politique de pire cas.

taille S_n à allouer en considérant la loi de probabilité SP qui résulte des $n_i - 1$ tirages de la loi P (la loi utilisée pour déterminer le nombre d'instructions de bruit à insérer), où n_i est le nombre d'instructions originales. $SP[X = l]$ correspond alors à la probabilité que le nombre d'instructions de bruit insérées soit égal à l. On peut alors calculer la taille à allouer pour que la probabilité d'avoir un débordement soit inférieure à un seuil donné : cette taille S_n correspond à la taille d'une instruction de bruit multipliée par le plus petit entier k qui vérifie la condition $\sum_{j=k+1}^{\infty} SP[X = j] < seuil$.

Odo calcule automatiquement SP à partir de la connaissance de P et n_i, puis détermine la taille appropriée à allouer en considérant soit un seuil fourni par l'utilisateur, soit un seuil par défaut fixé à 10^{-6}.

Il en résulte qu'avec le seuil par défaut, la probabilité de générer directement un code qui tienne dans la mémoire allouée est supérieure à 999999 chances sur un 1000000, quelle que soit la taille originale du code. De plus, le SGPC empêche les débordements de tampon à l'exécution, comme expliqué dans la section suivante, pour garantir que le code tient toujours dans le tampon d'instance alloué.

L'écart en terme de taille qui résulte de l'utilisation de notre politique d'allocation à la place d'une politique d'allocation pour le pire cas (instance polymorphe la plus grande possible avec les transformations considérées) est considérable. La figure 5.2 montre cet écart en fonction du nombre d'instructions originales de la fonction. Cet l'écart est asymptotiquement constant à mesure que le nombre d'instructions originales augmente ; pour l'ISA ARMv7M, et avec le modèle `high-var` considéré ici, compte tenu d'un tirage de probabilité suivant P entre chaque paire d'instructions consécutives, la différence des tailles attribuées représente environ 58 octets pour chaque instruction originale de la fonction, et environ 10 octets pour chaque instruction originale dans le cas du modèle `low-var`. Si l'on considère une fonction originale de 200 instructions, il en résulte une différence de 2ko pour le modèle à `low-var` et de 11,6ko pour le modèle `high-var` considéré.

La figure 5.3 illustre le ratio entre la taille allouée par notre politique d'allocation et la taille allouée par une politique de pire cas. Elle montre que la taille allouée par notre politique représente seulement 10% à 20% de la taille allouée par une politique de de pire cas pour les deux modèles de bruit dès que le nombre d'instructions originales dépasse les 100 instructions.

5.4.2 Prévention des dépassements de tampon

Odo calcule statiquement la taille maximale des instructions utiles S_u, en considérant les plus grands équivalents sémantiques lorsque des variantes sémantiques sont disponibles, et donne cette information au SGPC. Lors de l'exécution, le SGPC initialise avec S_u une variable traçant l'espace nécessaire pour les instructions utiles qui restent à générer avec les variantes sémantiques les plus volumineuses. Cette variable est décrémentée tout au long de la génération du code, après l'émission de chaque instruction utile. De plus, à chaque génération d'instructions de bruit, le générateur d'instructions de bruit calcule le nombre maximum d'instructions qu'il peut insérer en considérant la valeur de cette variable et l'espace disponible dans le tampon d'instance. Ces informations permettent au SGPC de limiter la génération d'instructions de bruit pour garantir qu'aucun débordement ne puisse se produire.

La politique d'allocation proposée via l'utilisation d'un seuil permet de limiter la taille des tampons tout en maîtrisant la probabilité de limiter le polymorphisme. Il est important d'allouer un tampon d'instance de taille réaliste, afin que la prévention des débordements de tampon n'introduise pas un biais dans les modèles probabilistes utilisés pour mettre en œuvre le polymorphisme, c'est à dire qu'aucune vulnérabilité ne soit introduite. Par exemple, si l'on considère uniquement l'insertion d'instructions de bruit : si la taille allouée ne permet pas d'émettre plus que les instructions d'origine, les instructions de bruit ne seront jamais émises, et l'instance polymorphe ne présentera aucune variabilité comportementale.

5.4.3 Gestion des permissions des tampons d'instance

La génération de code à l'exécution nécessite que les tampons d'instance soient accessibles avec des droits d'écriture (W) lors de la génération et avec des droits d'exécution (X) lors de l'exécution de l'instance polymorphe. Dans les systèmes embarqués, les permissions d'écriture sont systématiquement désactivées sur la mémoire du programme pour empêcher l'injection de code. Pour résoudre ce problème, les JITs fournissent généralement un accès à la mémoire du programme avec des permissions d'écriture uniquement lors de la génération du code, puis d'exécution uniquement lors de son exécution (W$\oplus$X) [Chen et al., 2011, Chen et al., 2013, Jauernig et al., 2014]. Nous suivons la même approche, mais en plus de garantir que le tampon d'instance n'est jamais à la fois disponible en écriture et en exécution, le tampon est protégé pour que seul le SGPC associé à ce tampon d'instance puisse y écrire. Dans cette section, nous décrivons notre technique de sécurisation basée sur le fait que les tampons d'instance sont alloués statiquement, et que chaque tampon d'instance a un unique SGPC associé.

Si la plate-forme cible a une unité de protection de mémoire (MPU), nous l'utilisons pour changer les autorisations d'accès au tampon d'instance d'exécution seule à écriture seule (et vice-versa) lorsque nécessaire. Le mécanisme proposé est illustré en figure 5.4. Par défaut, un tampon d'instance ne dispose que du droit d'exécution. Au début de l'exécution du SGPC, le SGPC génère une interruption pour demander l'autorisation d'écriture. Le gestionnaire d'interruption vérifie l'adresse où l'interruption a été levée. Si l'adresse est bien celle du début d'un SGPC, le gestionnaire d'interruption remplace le droit d'exécution par le droit d'écriture seule pour le tampon d'instance associé au SGPC ayant levé l'interruption. Grâce à l'allocation statique des tampons d'instance, le gestionnaire d'interruptions connait quelle zone mémoire est associée à quel SGPC (pour permettre un changement de permissions uniquement pour les paires correctes d'adresses des interruptions et zones mémoire) et les adresses de chaque tampon (pour ne changer les autorisations que pour une zone tampon). À la fin de l'exécution du SGPC, le droit d'écriture est supprimé et le droit d'exécution est ajouté en suivant le même principe. Cette solution est légère (voir section 5.6.3.2). Elle peut être étendue aux systèmes qui fournissent une MMU au lieu d'une MPU, mais il faut alors n'avoir qu'un seul tampon d'instance par page mémoire, afin de garantir que seul le SGPC légitime puisse écrire dans son tampon d'instance. Cela pourrait augmenter l'impact mémoire de l'approche. De plus, cette solution ne convient que si le système n'effectue pas de multitâche préemptif.

Pour les systèmes dotés d'un système d'exploitation multitâche préemptif, le système d'exploitation doit être adapté pour gérer les droits d'accès pour garantir un accès exclusif aux tampons de code. Le système d'exploitation pourra tirer parti des informations statiquement disponibles pour valider la légitimité des changements de permissions de la même manière que le gestionnaire d'interruptions dans l'approche présentée.

Pour toute plate-forme sans unité de protection de mémoire (MPU) ni système d'exploitation, des techniques d'intégrité de flot de contrôle peuvent être utilisées pour garantir qu'un tampon d'instance peut :

— être modifié uniquement par le SGPC dédié,
— être exécuté seulement à partir de l'adresse où le code polymorphe est appelé, par exemple, dans le cas du listing 5.4 l'adresse correspondant à l'appel `code_f(a,b)`.

Le principe est d'insérer des vérifications avant chaque écriture mémoire et chaque branchement (direct ou indirect) afin de vérifier la validité de toute écriture dans un tampon de code et de tout saut dans un tampon de code. Cette idée a été présentée dans l'extension SMAC du CFI présentée par Abadi et al. [Abadi et al., 2009], et induit un surcoût en performance bien supérieur à une approche utilisant la MPU.

5.5 Implémentation

L'implémentation d'Odo est basée sur LLVM 3.8.0. Le back-end pour générer les SGPCs cible l'ISA ARMv7M, et plus particulièrement les jeux d'instruction Thumb et Thumb 2. Ce back-end est un clone du back-end ARM de LLVM, dans lequel la passe d'émission de code a été modifiée.

Les modifications réalisé dans ce nouveau back-end permettent à la passe d'effectuer l'analyse de vivacité des registres, et d'émettre du code C comportant le SGPC et le

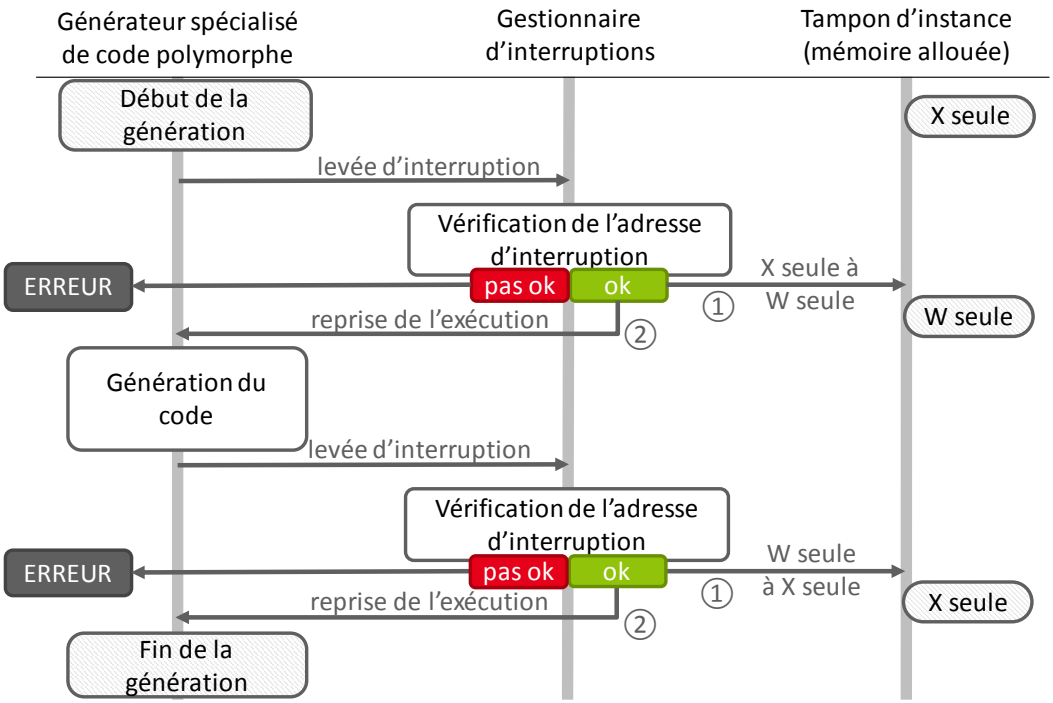

FIGURE 5.4 – Gestion des permissions en utilisant des interruptions et la MPU (unité de protection mémoire). Les tampons d'instance ne sont jamais disponibles en écriture (W) et en exécution (X) simultanément. L'allocation statique des tampons d'instance et la spécialisation des SGPCs permettent au gestionnaire d'interruption de connaître les adresses de levée d'interruption des différents SGPCs ainsi que les adresses des tampons d'instance associés.

wrapper des fonctions à sécuriser au lieu du code assembleur. Pour permettre à l'utilisateur de choisir une configuration de polymorphisme (transformations activées et leurs paramètres), nous avons ajouté diverses options en ligne de commande à LLVM pour définir la configuration choisie.

5.6 Évaluation expérimentale

Dans cette section, nous évaluons expérimentalement l'approche proposée. Comme expliqué précédemment, le niveau de variabilité des instances à l'exécution peut être configuré avec les transformations à utiliser et leurs paramètres potentiels. Nous analysons d'abord les performances et la sécurité de 17 configurations différentes sur un AES T-table, et nous discutons des compromis possibles entre le renforcement obtenu et les surcoûts induits. Ensuite, nous appliquons 4 configurations de polymorphisme sur 15 programmes variés. Ces 4 configurations sélectionnées comprennent une configuration avec polymorphisme désactivé, et 3 configurations utilisant différentes options de variabilité. Elles sont utilisées pour évaluer les surcoûts en performance et en taille de code avec divers degrés de polymorphisme, induisant une variabilité nulle à une variabilité importante.

5.6.1 Environnement expérimental

La plate-forme expérimentale et le matériel de mesure des émissions électromagnétiques est identique à ceux utilisés lors de l'évaluation expérimentale de Maskara (section 4.7.1).

5.6.2 Étude d'un cas d'usage : AES

Cette section présente une étude des performances et de la résistance de l'implémentation d'un AES face aux attaques par canal auxiliaire. La sécurité contre les attaques par canal auxiliaire d'implémentations de l'AES est étudiée depuis plusieurs années, et cette fonction de chiffrement est souvent utilisée comme référence pour l'évaluation de la résistance d'un schéma de protection. L'implémentation choisie est celle de la bibliothèque `mbed TLS` [mbe], dénotée AES T-table dans la suite. La bibliothèque `mbed TLS` est utilisée dans de nombreux systèmes embarqués, des dispositifs IoT aux ordinateurs mobiles et de bureau. L'implémentation de l'AES T-table d'origine ne comporte aucune contre-mesure contre les attaques par canal auxiliaire.

L'efficacité des approches de dissimulation matérielles ou logicielles contre les attaques par canal auxiliaire est la plupart du temps évaluée à l'aide de CPA [Avirneni et Somani, 2014, Boey et al., 2010, Güneysu et Moradi, 2011, Menicocci et al., 2014, Ordas et al., 2014, Singh et al., 2018, Yu et Kose, 2018, Gornik et al., 2015, Wang et al., 2013, Agosta et al., 2012, Agosta et al., 2015b, Couroussé et al., 2016, Coron et Kizhvatov, 2009, Coron et Kizhvatov, 2010], ou de t-tests [Gornik et al., 2015, Moradi et Wild, 2015, Sasdrich et al., 2017, Singh et al., 2018, Agosta et al., 2015b]. Ces deux méthodes d'évaluation ont été présentées en section 2.5.

Nous utilisons ici le t-test non spécifique en complément de la CPA pour les raisons suivantes :

— il peut être calculé assez rapidement, ce qui nous permet d'évaluer un grand nombre de configurations, y compris des configurations très polymorphes qui pourraient être trop résistantes pour obtenir un résultat en CPA dans un temps raisonnable,
— il est indépendant d'un modèle comportemental, et ne cible pas une valeur intermédiaire particulière,
— il a été utilisé pour évaluer l'approche MEET [Agosta et al., 2015b], qui est une approche automatique de dissimulation logicielle à l'état de l'art à laquelle nous souhaitons nous comparer.

Nous utilisons également une CPA sur une des configurations finalement choisies afin d'évaluer le temps nécessaire pour réaliser cette attaque.

Tant pour le t-test que pour la CPA, un signal de synchronisation a été configuré, via une broche GPIO sur le dispositif, au début du chiffrement AES, et après exécution du SGPC, pour faciliter l'alignement temporel des traces. Nous avons vérifié que nos mesures couvrent une durée d'exécution suffisante, tant pour l'implémentation non protégée que pour les implémentations polymorphes de l'AES. Il est à noter que le SGPC ne manipule pas la clé de chiffrement secrète et qu'il n'est donc pas vulnérable aux attaques par canal auxiliaire qui sont utilisées ici. Il faut également souligner que notre dispositif de synchronisation par GPIO rend l'attaque plus facile car un attaquant devra d'abord aligner les traces.

5.6.2.1 Évaluation par t-test de la dissimulation apportée

Dans cette évaluation, le t-test non spécifique a été effectué en mesurant l'émission électromagnétique de notre plate-forme lors de l'exécution de deux groupes de 10000 chiffrements chacun. Comme décrit en section 2.5.2, les deux groupes de traces de mesure rassemblent respectivement les émissions électromagnétiques récupérées lors des chiffrements de textes clairs aléatoires et de textes clairs fixes, et les chiffrements des textes clairs des deux groupes sont entrelacés de manière aléatoire. Pour chaque t-test, nous avons retenu la valeur `t` maximale (en absolu). Dans ce qui suit, on note $\texttt{t}_{\texttt{max}}$ la valeur `t` maximale d'un t-test. Lorsque cette valeur est inférieure à 4,5, on dit que le t-test réussit.

Analyse des effets des transformations sur la valeur $\texttt{t}_{\texttt{max}}$. Nous avons réalisé 10 évaluations par t-test, pour 17 configurations différentes de polymorphisme afin d'évaluer la dissimulation ainsi obtenue sur l'AES T-table sur notre plate-forme. Chaque configuration est désignée avec les acronymes des transformations utilisées, présentés dans la table 5.2. Nous considérons soit des configurations comportant une seule transformation activée, soit utilisant plusieurs transformations avec divers paramètres, ainsi qu'une configuration appelée `none` qui ne contient aucune transformation activée. Pour toutes les configurations, la période de regénération a été fixée à 1.

Bien que nous ayons limité l'étude à 17 configurations, d'autres configurations pourraient aisément être ajoutées en jouant sur les différents paramètres des transformations.

TABLE 5.2 – Acronymes utilisés pour les noms des configurations polymorphes utilisées dans l'évaluation expérimentale. Lorsque plusieurs transformations sont activées dans une configuration, celle-ci est désignée par les acronymes des différentes transformations séparés par des + (par exemple, `RS+IS` signifie que le mélange des registres et le mélange des instructions sont activés).

Nom	Configuration
RS	mélange des registres
IS	mélange d'instructions
SV	variantes sémantiques
N1	instructions de bruit sans bruit dynamique, modèle de probabilité `low-var`, p=1/7, N=4
N2	instructions de bruit sans bruit dynamique, modèle de probabilité `high-var`, p=1/4, N=4
DN1	instructions de bruit avec bruit dynamique, modèle de probabilité `low-var`, p=1/7, N=4
DN2	instructions de bruit avec bruit dynamique, modèle de probabilité `high-var`, p=1/4, N=4

La figure 5.5 montre un diagramme en violon des valeurs $\mathtt{t}_{\mathtt{max}}$ obtenues pour les 10 t-tests effectués pour chaque configuration. Le diagramme en violon permet de représenter les valeurs $\mathtt{t}_{\mathtt{max}}$ minimales et maximales, la médiane des valeurs $\mathtt{t}_{\mathtt{max}}$ obtenues lors des 10 t-tests, et la distribution des valeurs $\mathtt{t}_{\mathtt{max}}$.

Sans aucune protection, le t-test échoue et les valeurs $\mathtt{t}_{\mathtt{max}}$ sont très élevées : la configuration `none` présente une médiane des valeurs $\mathtt{t}_{\mathtt{max}}$ de 110. Les 5 transformations polymorphes que nous déployons ont deux objectifs : introduire de la désynchronisation (mélange d'instructions, variantes sémantiques, bruit et bruit dynamique) et modifier le profil de la fuite (mélange de registres, variantes sémantiques, bruit et bruit dynamique). Les résultats montrent qu'au fur et à mesure que de nouvelles transformations sont activées, le nombre de t-tests réussis augmente et les valeurs $\mathtt{t}_{\mathtt{max}}$ diminuent fortement. Une des configurations (`RS+IS+SV+DN2`) réussit les 10 t-tests, ce qui signifie qu'aucun t-test n'a détecté de fuite d'information.

Chacune des transformations polymorphes utilisées isolément a un impact différent sur la fuite d'information observée (figure 5.5). Le mélange des registres (`RS`) semble avoir peu d'impact sur la valeur $\mathtt{t}_{\mathtt{max}}$ pour notre plate-forme, mais peut être intéressant pour d'autres plate-formes où l'index du registre a un effet sur les mesures [Seuschek et Rass, 2015]. Le mélange d'instructions (`IS`) a un effet plus important sur la réduction de la valeur $\mathtt{t}_{\mathtt{max}}$ que le mélange de registres, mais a un effet plus faible que les variantes sémantiques (`SV`). Les instructions de bruit (`N1`, `N2`, `DN1`, `DN2`) ont un effet plus important que les variantes sémantiques sur la réduction de la valeur $\mathtt{t}_{\mathtt{max}}$, et le bruit dynamique (`DN1`, `DN2`) a le plus grand effet.

Lorsque les transformations sont combinées, l'effet résultant sur la valeur $\mathtt{t}_{\mathtt{max}}$ est plus difficile à analyser, leur combinaison pouvant présenter des interactions complexes. Par exemple, le bruit dynamique pourrait réduire l'effet du mélange d'instructions car il introduit des sauts qui forment des barrières pour celui-ci. Pourtant, seul ou combiné, le bruit dynamique semble être la transformation qui a l'impact le plus important sur la

valeur $\texttt{t}_{\texttt{max}}$: il augmente clairement le nombre de t-tests réussis par rapport au bruit non dynamique, puisque `DN1` et `DN2` passent plus de t-tests que `N1` et `N2` respectivement. Nous notons également que l'utilisation d'un modèle de bruit avec une plus forte probabilité d'insertion d'instructions de bruit (`high-var` avec p=1/4 et N=4 au lieu de `low-var` avec p=1/7 et N=4) améliore la métrique observée ; `DN2` et `N2` ont des valeurs $\texttt{t}_{\texttt{max}}$ plus faibles que `DN1` et `N1` respectivement.

Pour l'AES de cette étude et sur notre plate-forme, la configuration réussissant le plus de t-tests est celle où toutes les transformations sont activées. L'impact d'une configuration sur la fuite d'information par canal auxiliaire dépend cependant de la plate-forme et de l'application, il est donc important d'effectuer une telle étude sur chaque cas d'application pour pouvoir ajuster le compromis sécurité / performance.

Analyse des effets du bruit dynamique sur la valeur $\texttt{t}_{\texttt{max}}$ lorsque la période de regénération augmente. Nous étudions plus précisément l'effet du bruit dynamique sur la valeur $\texttt{t}_{\texttt{max}}$ lorsque la période de regénération augmente, de manière à déterminer à quel point cette transformation permet de maintenir le niveau de dissimulation lorsque la période de regénération ω augmente.

La figure 5.6 montre l'évolution de la médiane des valeurs $\texttt{t}_{\texttt{max}}$ recueillies sur 10 t-tests en fonction de ω, pour les configurations `N1`, `N2`, `DN1` et `DN2`. Les résultats montrent que les configurations `DN1` et `DN2` induisent une fuite d'information mieux dissimulée que `N1` et `N2` respectivement pour toutes les valeurs de période de regénération testées. De plus, les valeurs $\texttt{t}_{\texttt{max}}$ des configurations où le bruit dynamique est activé sont moins sensibles à l'augmentation de la période de regénération. Par exemple, alors que `N2` et `DN1` présentent des valeurs $\texttt{t}_{\texttt{max}}$ similaires pour de petites périodes de regénération, la dissimulation de la fuite d'information obtenue avec `N2` commence à perdre en importance à partir d'une période de 100, tandis que la dissimulation de la fuite d'information obtenue avec `DN1` commence à perdre en importance à partir d'une période de 1000. La configuration `DN2` présente une capacité encore plus grande à maintenir le niveau de dissimulation lorsque la période augmente, car les valeurs $\texttt{t}_{\texttt{max}}$ observées ne commencent à augmenter significativement que pour les périodes supérieures à 8000.

Ainsi, le bruit dynamique élargit le choix de la période de regénération. Avec cette transformation, l'utilisateur peut augmenter la période de regénération beaucoup plus qu'il ne peut le faire avec un bruit non dynamique. Cela peut lui permettre d'amortir le surcoût de génération.

5.6.2.2 Surcoûts en performance et taille de code

Dans cette section, nous discutons l'impact des différentes transformations sur les performances et la taille du code. Nous avons utilisé le niveau d'optimisation `-O2` pour Odo, et tous les fichiers C ont été compilés avec clang 3.8.0 en utilisant les options de compilation `-O2 - static -mthumb -mcpu=cortex-m3`. Les temps d'exécution ont été mesurés en nombre de cycles d'horloge du processeur. La taille des programmes (sections de données, de texte et de bss) a été mesurée en octets avec l'outil `arm-none-eabi-size` de la chaîne d'outils GCC. Nous avons mesuré les surcoûts en performance, exprimés en temps

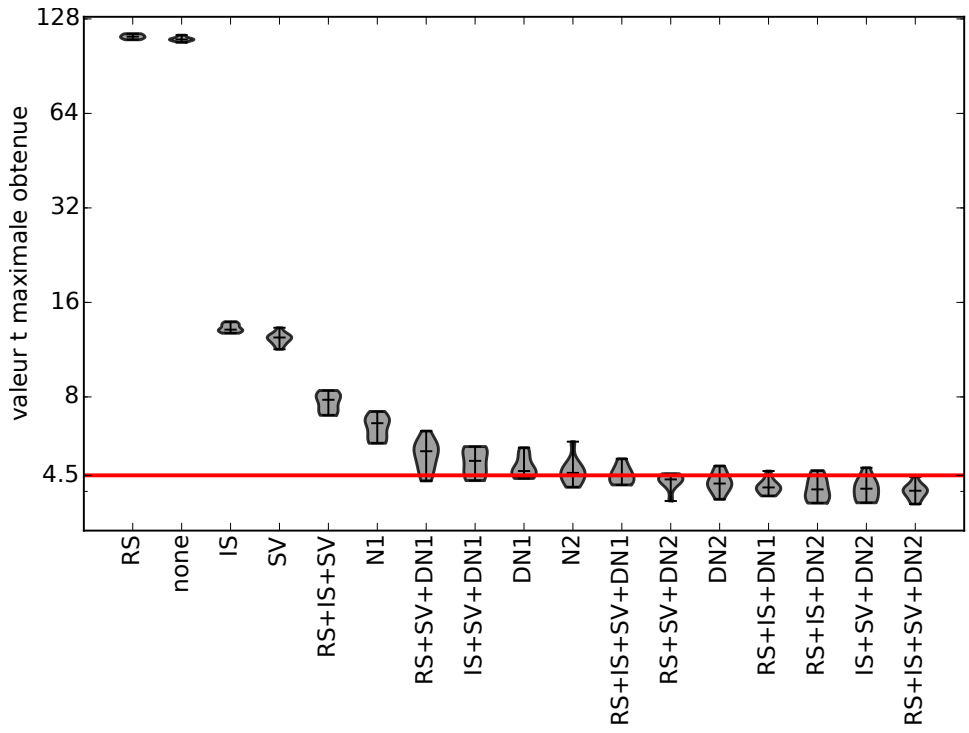

FIGURE 5.5 – Valeurs t_{max} obtenues pour 10 t-tests avec 17 configurations différentes. Cette figure représente la distribution de ces valeurs t_{max} ainsi que le minimum, maximum et la médiane obtenus. Les configurations sont triées par les valeurs médianes obtenues décroissantes. Plusieurs configurations ont passé le t-test plus de fois qu'elles ne l'ont échoué, et la configuration `RS+IS+SV+DN2` a passé tous les t-tests.

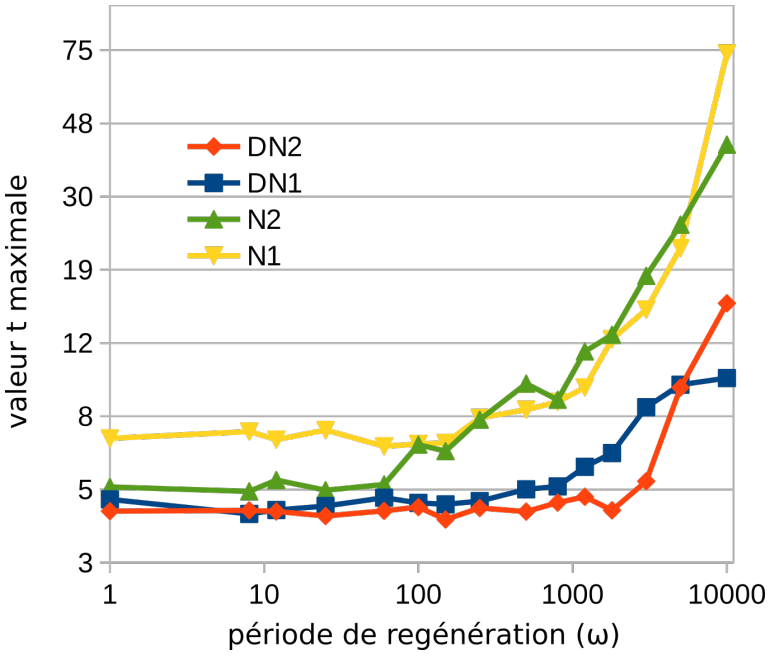

FIGURE 5.6 – Valeurs t_{max} observées (médiane des valeurs maximales obtenues lors des 10 t-tests) pour les configurations avec et sans bruit dynamique, en fonction de la période de regénération. Les configurations avec bruit dynamique montrent un meilleur niveau de dissimulation de la fuite d'information, et parviennent à mieux maintenir ce niveau lorsque la période de regénération augmente.

d'exécution relatif par rapport au code de référence, avec et sans prise en compte du surcoût de la génération. Les surcoûts en temps d'exécution indiqués représentent le rapport entre le temps d'exécution moyen de 10000 instances (sans le surcoût de génération du code) et le temps d'exécution du code original.

Temps d'exécution et taille de code. La table 5.3 présente le surcoût en temps d'exécution, le surcoût total (incluant le temps de génération) avec différentes périodes de regénération, le surcoût en taille de code et le temps de génération en cycles d'horloge obtenus pour les 17 configurations.

Le surcoût total en performance est prohibitif lorsque $\omega = 1$ pour les configurations présentant une valeur t_{max} proche du seuil de 4,5, cette période de regénération n'est probablement intéressante que si la génération peut être effectué à des moments n'impactant pas le temps de réponse du système (par exemple lorsque le système attend un paquet réseau : on dit dans ce cas que le surcoût de la génération est *masqué*). Cependant, il devient plus raisonnable à mesure que la période de regénération s'allonge. Le surcoût d'exécution de la configuration `none` de 1,34 peut s'expliquer en partie par le fait que l'instance polymorphe est générée en RAM (le code de référence est placé en mémoire Flash) ce qui provoque un accroissement du temps d'exécution des lectures mémoire et des écritures mémoire sur notre plate-forme. Plus précisément, lorsque l'instance est stockée en RAM, une instruction de load Thumb 1 (16 bits) prend en moyenne 1,5 cycle et une instruction de load Thumb 2 (32 bits) prend 2 cycles (au lieu de 1 cycle lorsqu'elle est stockée en mémoire flash). Cela provient sans doute du fait que les données placées en RAM et les instructions à charger depuis la mémoire utilisent les mêmes bus, alors

qu'avec un code placé en mémoire Flash les chemins pour charger les instructions et les données placées en RAM diffèrent. S'il est possible de générer le code en mémoire Flash, le surcoût en temps d'exécution peut probablement être réduit.

Les différentes transformations influencent différemment les surcoûts obtenus.

L'activation du mélange d'instructions en plus d'autres transformations double le temps de génération en raison du temps de calcul nécessaire à l'analyse use-def, qui est exécutée à l'insertion de chaque nouvelle instruction. Par exemple, le temps de génération est de 82077 cycles pour `RS+SV+DN1` et de 163873 cycles pour `RS+IS+SV+DN1`, 107820 cycles pour `RS+SV+DN2` et de 218909 cycles pour `RS+IS+SV+DN2`. Cependant, cette transformation a peu d'influence sur le temps d'exécution et sur la taille de code. Elle est donc intéressante si le surcoût de génération est masqué, ou si la période de regénération est suffisamment longue pour minimiser l'impact sur la performance.

L'activation du mélange des registres a un faible impact sur le temps d'exécution et le plus faible impact sur les surcoûts totaux.

Les variantes sémantiques ont un impact à la fois sur le temps d'exécution, le temps de génération et la taille. Cet impact est fortement couplé au nombre d'instructions pour lesquelles des variantes sont disponibles. La fréquence d'exécution de ces instructions peut aussi avoir un rôle sur l'impact en temps d'exécution, puisque des variantes situées dans un corps de boucle seront exécutées plus de fois que des variantes situées en dehors de toute boucle.

Enfin, les surcoûts engendrés par le bruit et le bruit dynamique dépendent beaucoup de la loi de probabilité P utilisée. Notamment, la valeur du paramètre p de la loi P impacte le temps de génération puisqu'elle définit la probabilité que le code en charge de déterminer quelles instructions de bruit insérer et combien en insérer soit exécuté.

5.6.2.3 Compromis dissimulation / performance

La configurabilité de notre approche permet d'explorer différentes configurations pour trouver un compromis adapté aux contraintes de l'utilisateur, en mesurant les surcoûts en performance et les niveaux de dissimulation (avec une mesure comme les valeurs $\mathtt{t}_{\mathtt{max}}$ par exemple) pour une plate-forme et une application données.

Impact du bruit dynamique sur le compromis dissimulation / performance. Nous étudions d'abord l'impact du bruit dynamique et de la période de regénération sur les compromis entre dissimulation et surcoûts en temps d'exécution.

La figure 5.7 montre le surcoût total obtenu avec bruit dynamique (`DN1` et `DN2`) ou avec bruit classique (`N1` et `N2`) lorsque ω varie de 1 (en haut à gauche) à 10000 (en bas à droite) tracés en fonction de la dissimulation observée (valeur $\mathtt{t}_{\mathtt{max}}$). La zone idéale se trouve en bas à gauche, là où les codes durcis dissimulent le mieux la fuite d'information et ont de meilleurs performances.

TABLE 5.3 – Temps d'exécution et taille de code pour chacune des configurations. "Exéc" correspond au surcoût en temps d'exécution exprimé relativement au temps de référence. "Total" correspond au surcoût total, exprimé relativement au temps de référence. "Gen" correspond au temps de génération en cycles d'horloges. "Taille" correspond au surcoût en taille exprimé relativement à la taille du code de référence. ω correspond à la période de regénération. Les configurations sont classées de la même manière que dans la figure 5.5

Config.	RS	none	IS	SV	RS+IS +SV	N1	RS+SV +DN1	IS+SV +DN1	DN1
Exéc	1,48	1,34	1,34	2,09	2,53	2,08	3,56	3,07	2,31
Total									
ω=1	8,39	6,91	37,51	17,04	70,41	43,38	67,43	115,44	49,58
ω=25	1,76	1,57	2,79	2,68	5,24	3,73	6,12	7,56	4,20
ω=100	1,55	1,40	1,70	2,24	3,20	2,49	4,20	4,19	2,78
ω=250	1,51	1,37	1,49	2,15	2,80	2,24	3,82	3,52	2,50
ω=1000	1,49	1,35	1,38	2,10	2,59	2,12	3,62	3,18	2,35
Gen									
(cycles)	8876	7147	46476	19216	87228	53077	82077	144402	60750
Taille	1,54	1,38	1,62	1,56	1,84	2,11	2,41	2,36	2,11

Config.	N2	RS+IS +SV+DN1	RS+SV +DN2	DN2	RS+IS +DN1	RS+IS +DN2	IS+SV +DN2	RS+IS +SV+DN2
Exéc	3,39	3,60	4,97	3,71	2,49	3,90	4,48	5,02
Total								
ω=1	58,93	131,13	88,87	71,06	99,29	143,25	159,57	175,38
ω=25	5,61	8,71	8,32	6,41	6,36	9,48	10,68	11,83
ω=100	3,95	4,88	5,81	4,39	3,46	5,30	6,03	6,72
ω=250	3,61	4,11	5,30	3,98	2,88	4,46	5,10	5,70
ω=1000	3,45	3,73	5,05	3,78	2,59	4,04	4,64	5,19
Gen								
(cycles)	71366	163873	107820	86540	124391	179065	199283	218909
Taille	2,19	2,43	2,49	2,18	2,34	2,42	2,44	2,51

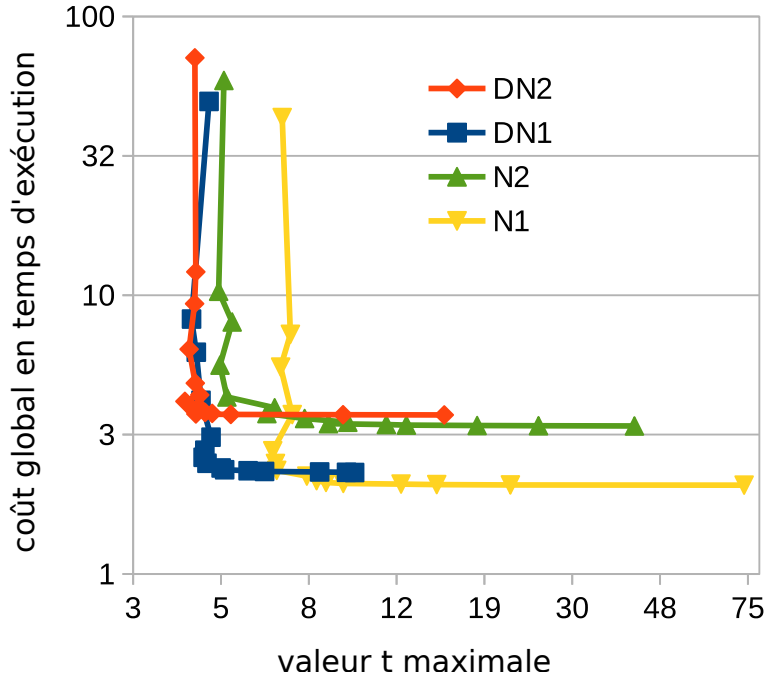

FIGURE 5.7 – Surcoûts totaux obtenus en augmentant la valeur de ω avec et sans bruit dynamique, en fonction de la valeur t_{max} obtenue. Les configurations avec bruit dynamique ont des surcoûts similaires ou inférieurs à valeur t_{max} fixée comparé aux surcoûts obtenus avec uniquement du bruit classique. Le bruit dynamique permet d'atteindre des valeurs t_{max} bien plus proches du seuil de 4,5 à surcoût en performance fixé

Les configurations avec bruit dynamique (`DN1` et `DN2`) se rapprochent plus de cette zone idéale que celles avec bruit classique. En particulier, à un surcoût total donné, ils offrent une bien meilleure dissimulation que les configurations avec du bruit classique.

Choix d'une configuration. Pour trouver un bon compromis, l'utilisateur peut commencer par sélectionner quelques configurations en tenant compte de l'impact de la génération de code sur son application et de ses contraintes de performance. Par exemple, si le surcoût de génération peut être masqué, alors le mélange d'instructions est une option intéressante car il augmente la variabilité tout en ayant peu d'impact sur le temps d'exécution. En revanche, si le surcoût de génération ne peut être masqué, cette option peut être éliminée pour limiter le surcoût total, et le bruit dynamique est un meilleur choix car il permet de choisir une période de regénération plus longue.

Ensuite, l'utilisateur peut effectuer des mesures de performance et de sécurité sur sa plateforme pour les configurations sélectionnées. Il peut alors éliminer les configurations qui ne correspondent pas à ses contraintes en termes de performances (par exemple celles qui conduisent à une exécution trop lente ou à un surcoût mémoire trop important). Enfin, il peut choisir la configuration qui présente la meilleure dissimulation et peut l'évaluer avec d'autres mesures de sécurité (un taux de succès en CPA par exemple) s'il le souhaite.

Pour le reste de l'évaluation menée dans le présent document, nous avons choisi quatre configurations qui présentent des compromis différents. Tout d'abord, nous avons choisi la configuration `none` (pas de variabilité) car elle permet de montrer les surcoûts minimaux induits par la génération et l'exécution du code généré dynamiquement. Ensuite, nous avons considéré une configuration nommée `low` adaptée à des environnements contraints. La configuration `low` est constituée du mélange de registre `RS` et du bruit dynamique `DN1` avec $\omega = 250$, pour avoir un impact limité sur les performances, et car la configuration `DN1` avec $\omega = 250$ se trouve en bas à gauche de la courbe `DN1` de la figure 5.7. Ensuite, nous avons choisi la configuration `RS+IS+DN1` comme configuration `medium` avec $\omega = 1$, en supposant que le surcoût de génération est masqué. Elle a un surcoût en temps d'exécution proche de celui de la configuration `low`, mais a montré un meilleur niveau de dissimulation. Elle induit cependant un temps de génération bien supérieur à celui de la configuration `low`. Enfin, nous avons choisi la configuration `RS+IS+SV+SV+DN2` avec $\omega = 1$ comme configuration `high`. Cette configuration a passé tous les t-tests. Son impact sur le temps d'exécution et sur le temps de génération est plus important que celui des autres configurations.

Les paramètres des configurations sélectionnées sont rappelés ci-après :

`none` - Aucune option de variabilité.

`low=RS+DN1` - Les options activées sont le mélange des registres (`RS`), l'insertion d'instructions de bruit avec le modèle de probabilité `low-var` (p=1/7, N=4) avec bruit dynamique (`DN1`). La période de regénération ω est fixée à 250.

`medium=RS+IS+DN1` - Tous les mécanismes sont activés sauf les variantes sémantiques. L'insertion des instructions de bruit utilise le modèle de probabilité `low-var` (p=1/7, N=4) avec bruit dynamique (`DN1`) et la période de regénération ω est fixée à 1.

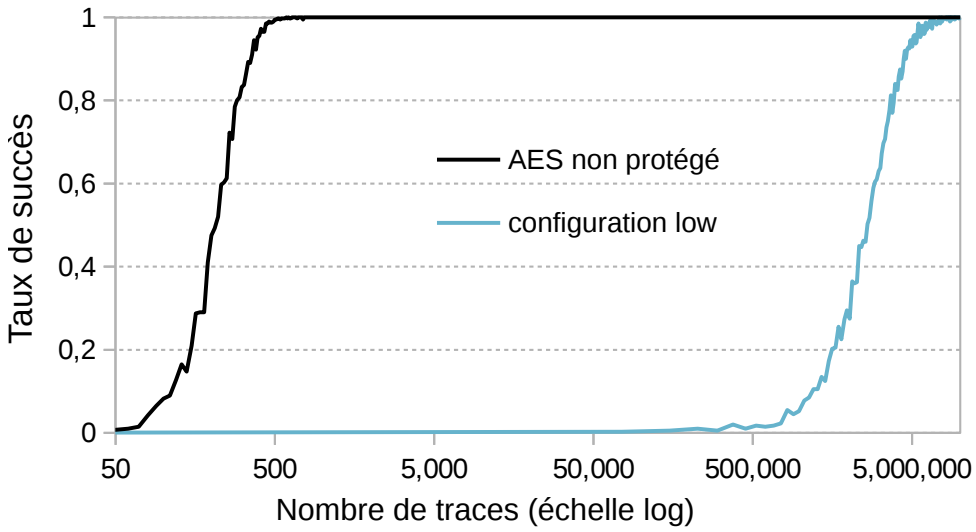

FIGURE 5.8 – Taux de succès en CPA pour une implémentation non protégée et une implémentation renforcée en utilisant la configuration `low`. Le nombre de traces nécessaires pour obtenir un taux de succès de 0,8 est multiplié par $1{,}3 \times 10^4$, alors que le temps d'exécution est multiplié par 2,5 (temps de génération inclus).

`high=RS+IS+SV+DN2` - Tous les mécanismes sont activés. L'insertion des instructions de bruit utilise le modèle de probabilité `high-var` (p=1/4, N=4) avec bruit dynamique (`DN2`) et la période de regénération ω est fixée à 1.

5.6.2.4 Évaluation de résistance à la CPA

Méthodologie. Nous avons effectué une CPA de premier ordre contre l'implémentation de référence et contre une implémentation protégée avec la configuration `low`. L'attaque considérée vise la sortie de la première fonction SubBytes du chiffrement AES. L'attaque menée vise à récupérer le premier octet de la clé. Nous avons utilisé le poids de Hamming comme modèle de l'émission électromagnétique de la sortie du SubBytes.

Résultats. La figure 5.8 présente le taux de succès obtenu pour la CPA par rapport à l'AES de référence (non protégé) et à une implémentation protégée par Odo avec la configuration `low`. Pour rappel, le taux de succès représente la proportion statistique d'attaques réussies (l'octet de clé est retrouvé) pour un nombre de traces fixé. Un taux de succès de 1 signifie que 100 % des attaques réussissent. Les résultats montrent que l'implémentation de référence est très vulnérable, puisqu'un taux de réussite de 0,8 (80% des attaques sont réussies) est atteint avec environ 290 traces sur notre plate-forme de test. Un tel nombre de traces est considéré comme faible pour les attaques par canal auxiliaire, même sur des implémentations non protégées. De plus, en utilisant 290 traces, nous obtenons une corrélation entre les poids de Hamming calculés et les mesures de 0,53 sur la bonne valeur de clé, ce qui suggère que notre système de mesure fournit de bonnes conditions d'attaque. L'implémentation durcie par Odo avec la configuration `low` présente une bien meilleure résistance à la CPA que l'implémentation non protégée de référence, puisque $3{,}8 \times 10^6$ traces doivent être collectées pour atteindre un taux de réussite de 0,8

TABLE 5.4 – Cas d'usage considérés pour l'évaluation en performance et en taille de code.

Nature du cas d'usage	Noms
Chiffrement par bloc	AES 8 bits, AES T-table, Camellia, Triple DES, XTEA, PRESENT, MISTY1, Simon.
Chiffrement par flot	ARC4, Rabbit, Salsa20, Trivium.
Fonctions de hachage	SHA256, MD5.
Fonction de comparaison	bytecompare d'un VerifyPin [Dureuil et al., 2016].

avec cette implémentation. Par rapport à la référence, cela représente $1{,}3 \times 10^4$ fois plus de traces, avec un temps d'exécution multiplié par 2,5 (temps de génération inclus).

Cependant, la CPA réalisée ne visait qu'à récupérer le premier octet de la clé. La récupération des autres octets de la clé peut être effectuée de la même manière avec une complexité d'attaque similaire en terme de calculs, mais un nombre de traces plus important pourrait être nécessaire pour l'implémentation polymorphe, puisque les autres octets sont utilisés plus loin dans l'algorithme de l'AES, et donc à des points où la désynchronisation est encore plus importante.

Enfin, une configuration polymorphe avec un degré de variabilité plus fort nécessiterait sans doute encore davantage de traces pour atteindre un taux de succès de 0,8.

5.6.3 Évaluation en performance

Pour évaluer l'impact de notre approche sur les performances et la taille des applications, nous avons considéré 15 cas d'usage (appelés *benchmarks* par la suite) différents, présentés dans la table 5.4. Les benchmarks choisis sont pour la plupart des applications cryptographiques, provenant soit de la bibliothèque `mbed TLS` [mbe], soit du projet `eSTREAM` [est]. Nous avons aussi considéré une implémentation maison 8 bits de l'AES. Comme toute fonction C peut être sécurisée par Odo, ces fonctions cryptographiques sélectionnées ne représentent qu'un tout petit panel des usages possibles de notre approche. Par conséquent, nous avons également sélectionné la fonction `bytecompare` d'une implémentation `verifyPin` de `FISSC` [Dureuil et al., 2016]. Nous avons évalué les benchmarks pour les 4 configurations sélectionnées et présentées en section 5.6.2.3. Dans la suite, nous analysons d'abord le temps d'exécution des instances polymorphes. Ensuite, nous présentons une analyse de la vitesse de génération du code puis des surcoûts en taille de code.

Tous les fichiers C produits par Odo ont été compilés en utilisant le niveau d'optimisation `-O2`. En raison de la mémoire limitée disponible sur notre cible, nous avons dû compiler les benchmarks `Rabbit` et `Salsa20` en utilisant le niveau d'optimisation `-O1` pour obtenir un code dont la taille est plus adaptée à la plate-forme : le niveau `-O1` offrait pour ces programmes une taille plus compacte que `-Os`. Comme dans le cas de l'AES, tous les programmes exécutés sur la plate-forme (les programmes initiaux ou ceux générés par Odo) ont été compilés avec la chaîne d'outils LLVM/clang en version 3.8.0 en utilisant les options de compilation `-O2 -static -mthumb -mcpu=cortex-m3`. Les temps d'exécution ont été mesurés en nombre de cycles d'horloge du processeur. La taille des

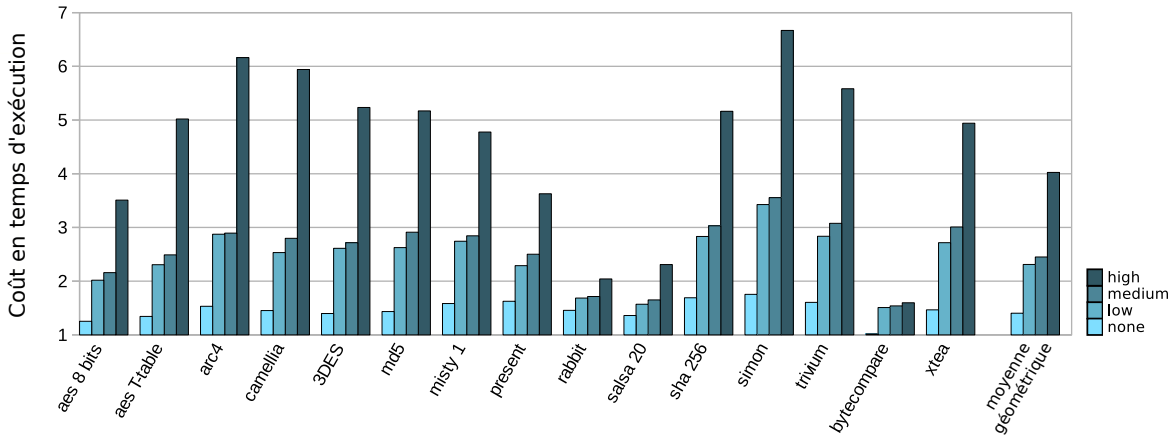

FIGURE 5.9 – Ratio des temps d'exécution obtenus par rapport aux temps d'exécution de référence de versions compilées statiquement (clang 3.8 `-O2`).

programmes (sections de données, de texte et de bss) a été mesurée en octets avec l'outil `arm-none-eabi-size` de la chaîne d'outils GCC.

5.6.3.1 Surcoûts en temps d'exécution

La figure 5.9 présente le surcoût en temps d'exécution (tel que défini à la section 5.6.2.2) pour les 15 benchmarks. En fonction des benchmarks et de la configuration, le surcoût en temps d'exécution varie d'un facteur 1, ce qui signifie que le code durci s'exécute aussi rapidement que le code de référence, jusqu'à un facteur 7. Le surcoût en temps d'exécution dépend de la nature des instructions du code initial. Premièrement, comme les variantes sémantiques ne sont pas disponibles pour toutes les instructions, elles affectent plus la performance de certains codes que d'autres. Deuxièmement, comme discuté en section 5.6.2.2, le code est généré en RAM, dans notre plate-forme, les chargements d'instruction (`fetch`) et les chargements de données (`load`) utilisent le même bus, et les lectures mémoire prennent alors plus de temps à exécuter que si le code était en flash.

L'impact global sur le temps d'exécution d'une application ayant une ou plusieurs fonctions polymorphes dépend beaucoup de la proportion du temps d'exécution initialement passé dans les fonctions transformées. Comme illustré sur le cas de l'AES en section 5.6.2.3, les différentes possibilités de configuration permettent d'adapter le polymorphisme aux contraintes en performance et exigences de sécurité de l'application.

5.6.3.2 Vitesse de génération

Afin d'analyser le surcoût de génération du code, nous avons corrélé le nombre d'instructions utiles au temps nécessaire à la génération du code pour toutes les configurations considérées. Nous avons mesuré le surcoût minimal de génération de code qui peut être obtenu avec un appel à un SGPC sans instruction à générer. Ce surcoût minimal est d'environ 1500 cycles. Il inclut le temps nécessaire pour changer les permissions des tampons d'instance avec la MPU ($\approx$ 100 cycles par génération). La figure 5.10 montre le temps de génération en fonction du nombre d'instruction utiles. Le temps de génération

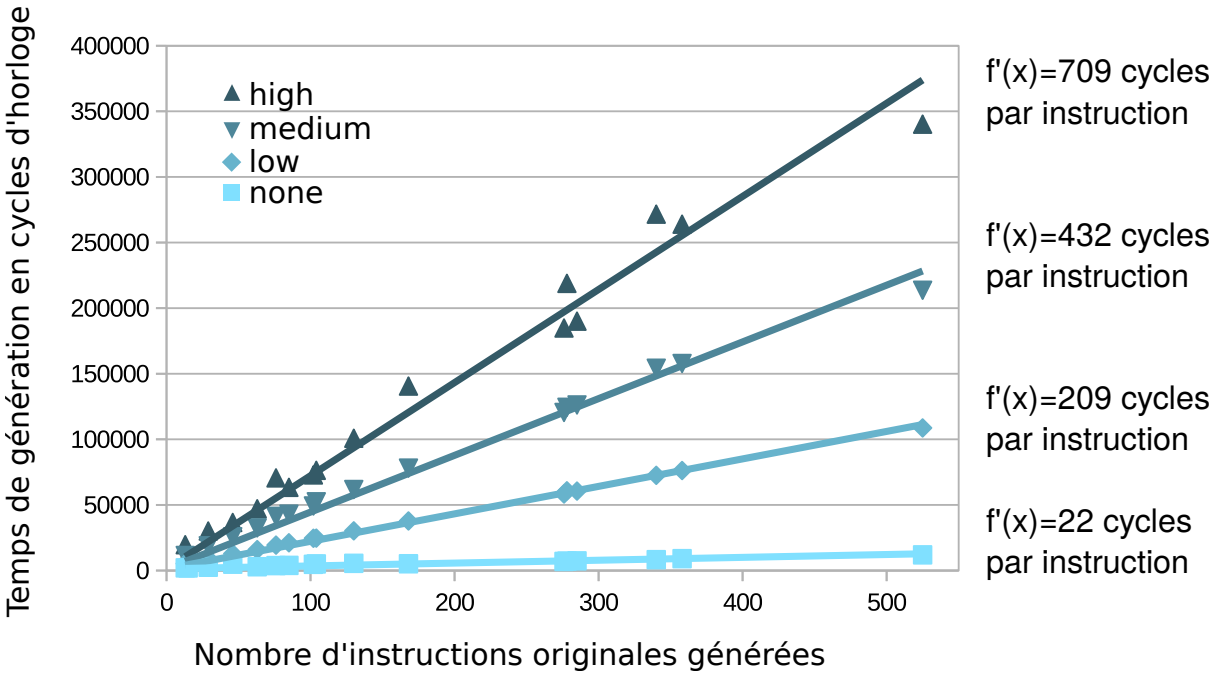

FIGURE 5.10 – Temps de génération en fonction du nombre d'instructions utiles générées. Les pentes des droites représentent le surcoût de la génération par instruction utile. Notre approche peut facilement passer à l'échelle à mesure que les fonctions s'agrandissent.

évolue linéairement avec le nombre d'instructions utiles à générer. Il est à noter que le nombre d'instructions utiles dépend uniquement du programme source et du niveau d'optimisation, mais ne dépend pas des options de polymorphisme. Les pentes des lignes de tendance de la figure 5.10 indiquent le nombre de cycles requis en moyenne pour générer une instruction utile. La génération nécessite environ 709 cycles par instruction utile pour la configuration `high`, et seulement 22 cycles par instruction quand aucun polymorphisme n'est introduit (configuration `none`). À titre indicatif, les compilateurs statiques comme LLVM prennent environ 3 millions de cycles par instruction [Charles et Lomüller, 2015], alors que le générateur de code spécialisé deGoal nécessite 233 cycles par instruction [Charles et al., 2014, Charles et Lomüller, 2015]. Nous pouvons donc en déduire que notre génération dynamique de code est efficace.

5.6.3.3 Surcoût en mémoire

La figure 5.11 illustre le surcoût en taille de code calculée comme le ratio entre la taille du binaire protégé et la taille du binaire non protégé. Le diagramme à barres donne la proportion des sections `.text` (code), `.bss` et `.data` (données). Les résultats montrent que les configurations présentant la plus grande variabilité induisent un surcoût en taille plus élevé que celle les moins polymorphes. Cela est dû à plusieurs facteurs. Tout d'abord, lorsque de nouveaux mécanismes sont activés, la bibliothèque et le code des SGPCs qui gèrent ces mécanismes sont ajoutés dans le binaire. Cela impacte la section `text`. Deuxièmement, les tampons d'instance des configurations les plus polymorphes doivent être plus grands, car la probabilité d'insertion d'instructions de bruit augmente ou parce que l'utilisation de variantes sémantiques exige de garder plus d'espace. Cela affecte la

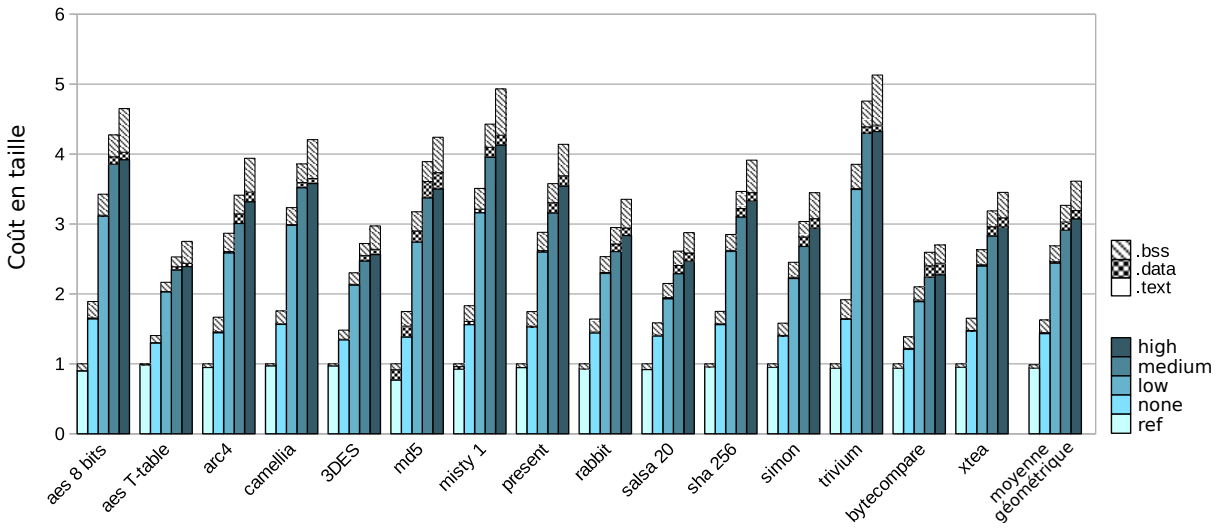

FIGURE 5.11 – Surcoût en taille mémoire, exprimé comme le ratio de la taille après application du polymorphisme divisée par la taille du code de référence compilé statiquement (clang 3.8 `-O2`).

section `bss`. L'augmentation de la section `data` est due à l'utilisation d'un tampon de mélange pour le mélange des instructions, et à certaines données privées de la bibliothèque `Odo-runtime`.

Le surcoût en taille de code peut sembler trop élevé pour les systèmes contraints, mais nous n'avons pris en compte que des benchmarks qui correspondent à une fonctionnalité critique d'un système embarqué. En limitant au minimum la partie du logiciel embarqué à sécuriser pour un produit donné, l'impact global devrait être beaucoup moins élevé. De plus, si plusieurs fonctions polymorphes sont déployées sur une même plate-forme, la bibliothèque `Odo-runtime` peut être partagée entre les SGPCs.

5.6.3.4 Conclusion de l'évaluation en performances

L'évaluation montre que notre approche est générique : l'approche peut s'appliquer sur des codes variés. L'impact sur la taille et le temps d'exécution varie en fonction de la configuration choisie et de l'application. La vitesse de génération évolue linéairement par rapport au nombre d'instructions originales du code, et est beaucoup plus élevée que la vitesse de compilation des compilateurs statiques. Les surcoûts obtenus peuvent être acceptables : la configurabilité de notre approche permet d'ajuster le compromis performance / sécurité. L'évaluation sécuritaire conduite sur un AES renforcé en utilisant la configuration `low` a montré que la CPA nécessitait 13000 fois plus de traces que sur la référence, alors que le temps d'exécution total n'est multiplié que par 2,5.

5.7 Discussion

Dans cette section, nous comparons notre approche avec les approches existantes les plus proches.

TABLE 5.5 – Comparaison des surcoûts en temps d'exécution pour Odo, Code Morphing [Agosta et al., 2012], et MEET [Agosta et al., 2015b].

Benchmarks. ω désigne la période de regénération	AES T-table	Camellia	3DES	MISTY1	PRESENT	XTEA	moyenne géométrique
Code Morphing ω=100 [Agosta et al., 2012]	**5,00**	-	-	-	-	-	5,00
Odo `low` ω=250	**2,50**	2,75	2,77	2,94	2,29	2,82	2,67
MEET [Agosta et al., 2015b]	6,76	8,99	8,61	10,1	2,79	6,02	**6,68**
Odo `high`, sans coût génération	5,01	5,94	5,23	4,78	3,63	4,94	**4,87**
Odo `high` ω=100	6,72	7,97	6,71	6,30	3,63	5,70	6,00
Odo `high` ω=1000	5,19	6,14	5,38	4,93	3,63	5,01	4,99

La libairie `Odo-runtime` est une version modifiée de COGITO, un cadriciel de génération de code d'exécution qui prend déjà en charge l'allocation aléatoire de registres, le mélange d'instructions, les variantes sémantiques, et l'insertion d'instructions de bruit [Couroussé et al., 2016]. Avec COGITO, le développeur doit implémenter ses composants polymorphes en utilisant un langage spécifique (DSL) de bas niveau. Cette approche apporte plus de flexibilité, mais nécessite de réimplémenter le composant polymorphe avec le DSL fourni. L'approche Odo offre une compilation automatique des SGPCs à partir du code source C annoté, un moyen précis d'estimer une taille de code réaliste pour l'allocation de mémoire statique ainsi qu'un mécanisme dynamique pour éviter les débordements de tampon d'instance pendant la génération du code d'exécution, et un mécanisme de protection mémoire utilisant la MPU. Enfin, notre approche met à disposition plus de transformations, dont le bruit dynamique qui permet d'obtenir de meilleurs compromis sécurité / performance.

Code Morphing [Agosta et al., 2012] et MEET [Agosta et al., 2015b] proposent aussi une approche automatisée pour déployer le polymorphisme. Code Morphing repose sur la modification dynamique du code et un compilateur pour appliquer le polymorphisme de code. Le moteur de modification dynamique de code utilise des équivalences sémantiques, mélange les instructions et effectue des permutations d'accès aux tableaux, mais il ne fournit aucune solution pour la gestion des permissions mémoire. Cela a été souligné par les auteurs de MEET comme une motivation pour leur approche qui élimine le besoin d'avoir des segments de code à la fois disponibles en écriture et en exécution. MEET repose sur une génération automatique et statique de multiples variantes sémantiques pour de petites séquences d'instructions du code à durcir. Les variantes sont ensuite sélectionnés au hasard au moment de l'exécution. Cette approche permet d'utiliser le polymorphisme de code sans avoir besoin de générer du code à l'exécution. Toutefois, elle peut souffrir de surcoûts élevés en taille de code pour obtenir un code très variable à l'exécution. Notre approche utilise la génération dynamique de code, mais elle permet d'atténuer les problèmes de sécurité liés à la génération dynamique.

Les tables 5.5 donnent le temps d'exécution total (incluant la génération de code dynamique) et les surcoûts en taille obtenus avec les différentes approches. Pour comparer les approches nous avons sélectionné des configurations qui sont proches en termes de sécurité. Les approches sur une même ligne ont des niveaux de dissimulation proches. Toutefois, le lecteur doit noter que l'évaluation de la sécurité effectuée dans chaque ar-

Table 5.6 – Comparaison des surcoûts en taille pour Odo et MEET [Agosta et al., 2015b].

Benchmarks	AES T-table	Camellia	3DES	MISTY1	PRESENT	XTEA	moyenne géométrique
MEET [Agosta et al., 2015b]	6,19	7,82	9,80	4,49	2,90	3,14	**5,18**
Odo `high`	2,75	4,20	2,97	4,93	4,13	3,45	**3,66**

ticle a été effectuée sur différentes plate-formes, ce qui impacte le niveau de sécurité avant même l'ajout de contre-mesures. Ainsi, les comparaisons du niveau de sécurité réalisable avec les différentes approches sont délicates. Pour comparer Odo avec Code Morphing, nous avons choisi la configuration `low` qui résiste à une CPA avec autant de traces que la CPA effectuée dans [Agosta et al., 2012], sur notre plate-forme qui est plus facile à attaquer (la clé est trouvée en moins de traces sur l'implémentation de référence sur notre plate-forme). Pour la comparaison avec MEET [Agosta et al., 2015b], nous avons choisi la configuration `high` car l'AES renforcé en utilisant cette configuration et l'AES renforcé en utilisant MEET passent tous les deux un t-test.

La table 5.5 donne le surcoût en temps d'exécution total (incluant la génération de code dynamique) pour notre approche, ainsi que les approches Code Morphing [Agosta et al., 2012] et MEET [Agosta et al., 2015b]. Notre approche peut être utilisée avec un surcoût bien plus faible (2× plus faible sur l'AES T-table) que l'approche dynamique automatique Code Morphing [Agosta et al., 2012], et avec un surcoût plus faible (1,37× moins en moyenne) que l'approche statique MEET lorsque le surcoût de la génération peut être masqué. Lorsque le surcoût de la génération ne peut pas être masqué, les surcoûts avec Odo `high` sont plus faibles que ceux de MEET même avec des périodes de regénérations faibles comme $\omega = 100$.

La table 5.6 donne le surcoût en taille pour notre approche et MEET [Agosta et al., 2015b]. Les surcoûts en taille de l'approche Code Morphing [Agosta et al., 2012] ne sont pas connus. Les surcoûts en taille obtenus avec notre approche sont bien plus faibles (1,4× moins en moyenne) que ceux de l'approche statique MEET.

De plus, Code Morphing induit des surcoûts importants lorsque la période de regénération est très petite. Considérant le temps d'exécution d'un AES T-table comme référence $T_{\texttt{ref}}$, nous avons estimé que le temps pris par les transformations de Code Morphing (le processus équivalent à notre regénération de code) est d'environ $393 \times T_{\texttt{ref}}$ alors que la configuration basse d'Odo prend $47 \times T_{\texttt{ref}}$, qui est presque un ordre de grandeur plus efficace. Par rapport à MEET, notre approche permet de réduire le surcoût en taille de code. Cependant, MEET ne comporte pas de surcoûts en temps d'exécution dus à la génération du code d'exécution, même si le temps d'exécution obtenu reste élevé. Avec Odo, l'impact sur le temps d'exécution peut être réduit si le surcoût de la génération du code peut être partiellement ou totalement masqué. De plus, les niveaux de sécurité obtenus par MEET sont impressionnants. Dans cette approche, le polymorphisme est combiné avec un masquage des sauvegardes et lectures sur la pile ainsi qu'un masquage des SBoxes pour éliminer les fuites d'informations liées à ces accès mémoire. Avec Odo, les accès mémoire sont brouillés par l'introduction d'accès mémoire de bruit, mais la fuite d'information due aux accès mémoire est toujours présente dans les observations par canal auxiliaire. Néanmoins, notre approche permet de cacher la fuite de telle sorte qu'elle n'est

pas détectable par un t-test, et ce sans technique de masquage. Notre approche pourrait aussi bien se combiner avec du masquage, pour un plus grand niveau de sécurité, mais aussi avec des surcoûts en performance plus élevés. Enfin, si le temps d'exécution de la génération du code ne peut pas être masqué, et pour des temps d'exécution comparables, le niveau de sécurité fourni par l'approche MEET est probablement supérieur à notre configuration high avec $\omega = 100$ en raison du masquage des accès aux SBoxes et des sauvegardes et chargements sur la pile.

5.8 Conclusion

Dans ce chapitre, nous avons présenté une approche automatique pour sécuriser le code contre les attaques par canal auxiliaire grâce au polymorphisme de code, implémenté par génération de code à l'exécution. Notre approche de durcissement automatique, implémentée dans l'outil Odo, basé sur LLVM, génère à partir d'un code source annoté des générateurs de code spécialisés pour chaque fonction à durcir. La spécialisation des générateurs de code permet d'obtenir une génération de code efficace. Notre approche basée sur la compilation permet d'optimiser le code à produire, de mettre à disposition des informations statiques pour les transformations du code faites à l'exécution ainsi que de gérer finement la mémoire qui va contenir le code généré. Plusieurs transformations appliquées à l'exécution font varier le code à chaque nouvelle génération. Nous avons également proposé la transformation de bruit dynamique pour introduire de la variabilité entre deux exécutions consécutives d'un même code généré et pour réduire la fréquence de génération du code.

Dans les résultats expérimentaux, nous avons évalué le niveau de sécurité de l'AES durci à l'aide de deux critères d'évaluation différents utilisés aujourd'hui pour l'évaluation des contre-mesures contre les attaques par canal auxiliaire : des t-tests non spécifiques et des attaques par analyse de consommation par corrélation (CPA). L'évaluation sécuritaire basée sur des t-tests montre que plusieurs niveaux de dissimulation peuvent être atteints. Nous avons également analysé l'impact des différentes transformations sur la dissimulation et la performance : les effets des transformations sur la taille de code, le temps d'exécution, le temps de génération et la dissimulation ont été abordés.

Nous avons sélectionné ensuite 3 configurations représentatives avec différents compromis de dissimulation et de performance. Une attaque CPA ciblant la moins variable de ces configurations s'est avérée être 13000 fois plus difficile qu'une CPA effectuée sur l'implémentation de référence non protégée.

Nous avons enfin évalué les surcoûts en termes de taille de code et de temps d'exécution en considérant 15 benchmarks et les 3 configurations sélectionnées afin d'obtenir une idée plus précise des coûts de notre approches sur des applications variées. L'évaluation montre que les surcoûts sont suffisamment faibles pour que notre approche soit applicable même sur des systèmes très contraints et qu'elle est compétitive par rapport à l'état de l'art. En effet, la génération de code est efficace et est environ 8 fois plus rapide comparée à la modification dynamique de code proposée dans l'approche de l'état de l'art la plus similaire [Agosta et al., 2012]. Les surcoûts sont également moins variables que pour la contre-mesure de masquage étudiée dans le chapitre précédent : ainsi les surcoûts en

temps d'exécution varient de ×1,5 à ×7 pour la configuration `high`, alors que les surcoûts observés pour la contre-mesure de masquage déployée dans le chapitre 4 variaient de ×1,4 à ×369.

Ainsi, les résultats expérimentaux ont montré que notre approche peut répondre à des besoins en termes de sécurité variés, grâce à une grande variabilité comportementale, tout en présentant un coût de performance acceptable ; sa haute configurabilité permet une adaptation des performances et niveaux de sécurité pour un cas particulier, ce qui permet une utilisation optimisée du polymorphisme sur toutes sortes de programmes. Notre approche élimine également le problème que constituait la génération de code à l'exécution pour les permissions mémoire, pour atteindre un degré de fiabilité similaire aux approches statiques à moindres coûts.

À court terme, les compromis entre performance et dissimulation obtenus pourraient probablement être améliorés en adaptant encore plus la contre-mesure à l'application et à la plate-forme visée. Tout d'abord, nous avons déposé une demande de brevet [Belleville et Couroussé, 2018] visant à augmenter arbitrairement la désynchronisation temporelle pour un point donné de la fonction, ce qui permettrait d'augmenter plus rapidement la variabilité lorsqu'une instruction sensible est proche du début ou de la fin de la fonction. Ce principe pourrait être intégré dans notre approche. Ensuite, la sélection des instructions de bruit pourrait tenir compte du contexte pour favoriser ou forcer l'insertion de lectures mémoires aléatoires entre deux lectures mémoires sensibles de manière à casser une éventuelle fuite en transition. La nature des instructions de bruit utilisées pourraient également varier en fonction des instructions utilisées dans le programme. Les lois de probabilités utilisées pour déterminer le nombre d'instructions de bruit à insérer pourraient évoluer au fur et à mesure de la génération en fonction du degré de sensibilité des différentes zones du code à sécuriser. Le bruit dynamique pourrait impliquer de multiples branchements aléatoires au lieu d'un seul, dans une même séquence d'instructions de bruit pour augmenter la variabilité.

À plus long terme, l'approche pourrait également être combinée avec du multiversionnement statique et avec du masquage pour renforcer encore plus le niveau de sécurité. La combinaison avec une contre-mesure de masquage soulève des questions notamment sur la gestion des registres, pour éviter de créer des fuites en transition : il est nécessaire de déterminer quels registres peuvent être utilisés sans risques pour les instructions de bruit et variantes sémantiques, et quels registres peuvent être sauvegardés sur la pile si aucun registre n'est libre.

L'application automatisée du polymorphisme de code ouvre donc des perspectives pour l'application de contre-mesures combinées et adaptatives.

Chapitre 6

Conclusion et perspectives

6.1 Conclusion

La sécurité des systèmes embarqués et objets connectés est aujourd'hui un enjeu majeur. Ceux-ci sont en effet utilisés dans des attaques de grande envergure, et manipulent des données sensibles. Lorsqu'un accès physique à ces systèmes est possible, il est nécessaire de les sécuriser contre les attaques physiques. Cependant, l'application des contre-mesures est difficile et coûteuse, elle nécessite de l'expertise et est encore manuelle. Il y a un donc fort besoin d'approches automatisées d'application de contre-mesures contre ces attaques.

Nous avons proposé dans cette thèse deux approches d'application automatique à la compilation de deux contre-mesures contre les attaques par canal auxiliaire reposant sur les principes de masquage et de dissimulation.

Nous sommes partis du constat que la compilation peut mettre à mal les contre-mesures, mais également qu'elle peut être un vecteur efficace pour les appliquer. En insérant dans le compilateur l'application d'une contre-mesure, on peut s'appuyer sur les formes variées par lesquelles le code passe lors de la compilation, ainsi que sur les analyses disponibles.

Pour la contre-mesure de masquage qui impacte beaucoup les calculs effectués, nous avons choisi de placer l'application de la contre-mesure en toute fin du middle-end, afin de conserver les optimisations du middle-end et afin d'être indépendants de l'architecture cible. Nous avons choisi d'opter pour le masquage booléen d'ordre 1 avec un modèle de fuite en valeur, et nous avons choisi de masquer les accès aux tables constantes en utilisant une approche par interpolation polynomiale. Nous avons proposé un algorithme d'application de la contre-mesure sur un code sous forme SSA, et nous avons proposé des améliorations pour cette contre-mesure sous la forme d'optimisations. Nous avons implémenté cette approche dans LLVM. Nous avons analysé les coûts en temps d'exécution obtenus pour diverses fonctions extraites d'implémentations de l'AES et de Simon, et plus particulièrement analysé les surcoûts liés au masquage des accès SBox pour les SBoxes de l'AES, de DES et de PRESENT. Les coûts en temps d'exécution varient énormément en fonction du code original. Les fonctions de Simon sont particulièrement adaptées au masquage booléen et montrent des surcoûts faibles de $\times$1,5 et $\times$1,6. Les fonctions de l'AES ont des surcoûts plus variables, compris entre $\times$1,4 et $\times$91,1. L'évaluation expé-

rimentale a aussi montré que d'importantes fuites à l'ordre 1 restaient présentes sur les codes masqués. Cependant, une analyse formelle sur un des codes masqués a montré l'absence de fuite en valeur d'ordre 1 dans le code. Ce constat nous a amenés à supposer la présence de fuites en transition sur la plate-forme cible. Nous avons ensuite validé cette hypothèse par l'ajout d'une nouvelle passe dans le back-end visant à ajouter des instructions inutiles pour éviter les transitions entre données secrètes. Nous en avons conclu que la plate-forme utilisée a des fuites en transitions liées à l'utilisation de l'ALU ainsi que des fuites en transitions liées aux accès mémoire.

Pour la contre-mesure de polymorphisme de code qui repose sur des transformations de niveau instruction assembleur, nous avons choisi de nous placer en fin de back-end du compilateur pour tirer parti de l'ensemble des optimisations du compilateur. Nous avons ensuite montré comment générer automatiquement des générateurs de code polymorphes spécialisés lors de la compilation. Nous avons proposé plusieurs améliorations pour cette contre-mesure. Tout d'abord, nous avons proposé un nouveau modèle de probabilités pour contrôler l'insertion d'instructions de bruit de manière à pouvoir obtenir une forte variance du nombre d'instructions insérées sans accroitre démesurément le nombre moyen d'instructions de bruit insérées. Ensuite, nous avons défini une nouvelle transformation appelée bruit dynamique, qui permet d'obtenir de la variabilité à l'exécution même lorsque le code n'est pas regénéré à chaque exécution. Nous avons montré que cette transformation permet de relâcher les contraintes sur la période de regénération, ce qui permet alors de regénérer moins souvent et par conséquent de mieux amortir le coût de la génération. De plus, nous avons créé une politique d'allocation mémoire réaliste qui permet de réduire fortement la consommation mémoire en comparaison d'une politique d'allocation mémoire pour le pire cas. Nous avons ajouté un mécanisme de prévention de dépassement de tampons d'instance, de manière à ce que de tels dépassements n'arrivent jamais. Enfin, nous avons mis en place des mécanismes de gestion des droits mémoire afin que seul un générateur légitime puisse écrire du code en mémoire, et de garantir que les zones mémoires ne soient jamais disponibles simultanément en écriture et en exécution. La configurabilité de l'approche permet d'adapter les transformations à une application et une plate-forme cible. L'évaluation expérimentale montre que l'utilisation de cette approche sur une implémentation d'AES renforcée avec une configuration induisant un surcoût en temps d'exécution de $\times 2{,}5$ (temps de génération inclus) permet d'obtenir un accroissement de la durée d'une CPA d'un facteur 13000. De plus, nous avons renforcé 15 fonctions différentes, montrant ainsi la généricité de cette approche. Les surcoûts obtenus dépendent de la configuration choisie, mais aussi du code à renforcer, bien que les écarts obtenus entre différentes fonctions soient bien moins prononcés que ceux constatés avec la contre-mesure de masquage. Enfin, nous avons montré que la génération dynamique de code est très efficace.

Lors de l'étude des contre-mesures de masquage et de polymorphisme de code appliquées ici, nous avons montré que ces contre-mesures se comportent différemment : le polymorphisme est moins dépendant du contenu des programmes à renforcer, il induit des surcoûts plus homogènes, il est très configurable, et il semble moins sensible à la micro-architecture de notre plate-forme, tandis que le masquage permet d'obtenir des surcoûts particulièrement faibles pour certains programmes.

6.2 Perspectives

Nous avons mis en évidence des limites du modèle de fuite en valeur sur notre plate-forme, ce qui guide naturellement une première perspective à court terme : l'application du masquage pourrait être améliorée en adaptant le back-end du compilateur pour prendre en compte le modèle de fuites plus réaliste.

D'autre part, la contre-mesure de polymorphisme de code pourrait également être améliorée à court terme en prenant en compte la position des instructions sensibles dans la fonction pour adapter l'insertion d'instructions de bruit de manière à ce que la désynchronisation temporelle soit forte aux instants d'exécution d'instructions sensibles. Notamment, la probabilité d'insertion d'instructions de bruit pourrait être plus forte en début et fin de fonction, de manière à ce que la désynchronisation augmente plus rapidement.

À plus long terme, les compromis sécurité / performance pourraient être améliorés en prenant en compte de manière encore plus fine la micro-architecture, à la fois pour le masquage mais aussi pour le polymorphisme de code. Le compilateur pourrait être muni de modèles d'émission des plate-formes cibles pour adapter l'application de la contre-mesure. Il pourrait ainsi prendre en compte des fuites dues à des registres cachés comme les registres présents entre le banc de registres et l'ALU sur le Cortex-M3 qui induisent des fuites en transitions entre deux utilisations de l'ALU [Le Corre et al., 2018]. La contre-mesure de masquage pourrait alors être mieux préservée. D'autre part, le polymorphisme pourrait être adapté en fonction du modèle. Par exemple, si une plate-forme fuit très peu en valeur et beaucoup en transition, utiliser du polymorphisme pour forcer les transitions à être aléatoires pourrait être un moyen efficace pour renforcer la sécurité. De plus, la prise en compte de la plate-forme est importante pour définir les transformations qui seront utiles à l'exécution : le mélange d'instructions pourrait être fortement atténué par un processeur capable de réordonner les instructions pour minimiser la latence. Sur une telle plate-forme, il serait alors contre-productif d'activer cette transformation, puisque l'impact en performance ne serait pas contre balancé par un renforcement de la sécurité.

Les contre-mesures ont été considérées ici séparément, mais la combinaison des deux contre-mesures offre des opportunités de recherche pour améliorer le compromis sécurité / performance. Par exemple, lorsqu'on masque un accès à une SBox, le compilateur pourrait générer plusieurs polynômes interpolateurs dans le code, et permettre un choix aléatoire entre eux à l'exécution pour obtenir une forte variabilité comportementale à faible coût en temps d'exécution. L'approche choisie pour masquer les SBoxes dans cette thèse offre en effet une grande flexibilité pour obtenir des codes différents et fonctionnellement équivalents : on peut soit changer les q_i, soit changer le polynôme primitif du corps fini, ce qui modifierait les multiplications de corps fini. Le fait de combiner les contre-mesures apporte aussi d'importantes questions quant à la gestion des ressources matérielles : quels registres utiliser pour les instructions de bruit de manière à éviter de créer de nouvelles fuites en transitions ? Au contraire, est-il intéressant d'utiliser un registre en priorité pour écraser sa valeur avec une instruction de bruit ?

La possibilité de contrôler le matériel et de le modifier, notamment avec l'architecture RISC-V qui permet des extensions du jeu d'instructions, ouvre plusieurs perspectives. Tout d'abord, des instructions pourraient être ajoutées pour faciliter l'application de la contre-mesure de masquage, et accélérer le code masqué. Certaines fonctions couramment

utilisées pourraient être supportées par le matériel, comme par exemple la fonction secMult, les fonctions de conversion de masquage booléen vers d'autres types de masquage, ou encore les multiplications de corps finis binaires. L'utilisation d'un tel support matériel permettrait alors d'obtenir des coûts de masquage plus faibles, ce qui pourrait faciliter le masquage à l'ordre supérieur. L'ajout de fonctionnalité de désynchronisation sur le matériel pourrait aussi apporter une complémentarité par rapport aux contre-mesures logicielles. Le polymorphisme apporte de la désynchronisation temporelle à gros grain, il ne modifie pas la durée des cycles d'horloges, alors que les fonctions de désynchronisation matérielles opèrent à un grain plus fin et permettent de faire varier la durée du cycle d'horloge. La combinaison de désynchronisation à grain fin et à gros grain pourrait compliquer encore d'avantage la tâche de l'attaquant pour resynchroniser ses traces.

Enfin, les progrès des attaques utilisant notamment de l'apprentissage machine soulèvent des questions importantes sur les moyens de sécurisation contre celles-ci. Une piste pourrait être d'utiliser également de l'apprentissage machine pour l'application des protections. Une contre-mesure comme le polymorphisme de code, déjà très configurable, pourrait l'être encore plus en laissant par exemple un choix complet sur la loi de probabilité d'insertion d'instructions de bruit, sur la nature des instructions à insérer, qui pourrait de plus dépendre du code utile environnent. Avec un espace de configuration aussi grand, il serait envisageable non plus de choisir les paramètres manuellement, mais automatiquement. On pourrait par exemple s'inspirer du principe de réseaux antagonistes génératifs (*generative adversarial networks*, ou *GAN*). Ce type d'apprentissage consiste à mettre deux réseaux en compétition, chacun s'améliorant petit à petit pour contrer son adversaire. On pourrait avoir un réseau qui essayerait d'attaquer une implémentation renforcée, et l'autre qui essayerait de générer une configuration pour faire en sorte que l'implémentation résiste au mieux aux attaques. Il faudrait alors imposer une contrainte en performance, pour forcer le réseau de défense à maximiser la sécurité tout en conservant un coût en performance tolérable. L'apprentissage machine pourrait alors aider à configurer une contre-mesure pour améliorer le compromis sécurité / performance d'un système.

Bibliographie personnelle

Article de revue

- Automated software protection for the masses against side-channel attacks
  N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  ACM Transactions on Architecture and Code Optimization (TACO) Volume 15 Issue 4, January 2019, Article No. 47. pp. 1-27
  https://doi.org/10.1145/3281662

Chapitre d'ouvrage

- Automatic Application of Software Countermeasures Against Physical Attacks
  N. Belleville, K. Heydemann, D. Couroussé, T. Barry, B. Robisson, A. Seriai, H.-P. Charles
  Koç Ç.K. (eds) Cyber-Physical Systems Security, Springer International Publishing. pp.135-155, 2018
  https://doi.org/10.1007/978-3-319-98935-8_7

Brevet

- Procédé d'exécution d'une fonction, par un microprocesseur, sécurisée par désynchronisation temporelle
  N. Belleville, D. Couroussé
  Institut National de la Propriété Industrielle, 23/07/2018
  Numéro de demande : FR1856781

Conférence avec acte

- All paths lead to Rome : Polymorphic Runtime Code Generation for Embedded Systems
  D. Couroussé, T. Barry, B. Robisson, N. Belleville, P. Jaillon, O. Potin, H. Le Bouder, J.-L. Lanet, K. Heydemann
  CS2-18 Proceedings of the Fifth Workshop on Cryptography and Security in Com-

puting Systems
https://doi.org/10.1145/3178291.3178296

Présentations

- Automated software protection for the masses against side-channel attacks
  N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  HiPEAC 2019, Valencia, Espagne
  https://doi.org/10.1145/3281662
- The Multiple Ways to Automate the Application of Software Countermeasures against Physical Attacks : Pitfalls and Guidelines
  N. Belleville, K. Heydemann, D. Couroussé, T. Barry, B. Robisson, A. Seriai, H.-P. Charles
  Cyber-Physical Security Education 2017, Paris, France
  https://doi.org/10.1007/978-3-319-98935-8_7
- Automated software protection for the masses against side-channel attacks
  N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  Séminaire sécurité des systèmes électroniques embarqués, mai 2019, Rennes, France
- Automated software protection for the masses against side-channel attacks
  N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  PHISIC 2018, Gardanne, France

Vidéo de médiation scientifique

- Processeurs, cuisines, et attaques par canal auxiliaire
  N. Belleville
  Accessit au *concours ScienceInfoStream* de la Société Informatique de France
  https://www.societe-informatique-de-france.fr/mediation/laureats-du-concours-video-scienceinfostream-2019/
  3ème prix au *concours ScienceTube* du CEA
  https://youtu.be/VHPqVm6zEzE

Posters

- Automated software protection for the masses against side-channel attacks
  N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  ACACES 2017, Fiuggi, Italie

Bibliographie

[com] Common criteria for information technology security evaluation. https://www.commoncriteriaportal.org/. (Citation page 3.)

[est] eSTREAM : the ecrypt stream cipher project. http://www.ecrypt.eu.org/stream/. (Citation page 105.)

[fas] Fast galois field arithmetic library in c/c++. http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/. (Citations pages 54 et 60.)

[llv] LLVM. http://llvm.org/. (Citation page 62.)

[mbe] mbedTLS library. https://tls.mbed.org/. (Citations pages 94 et 105.)

[pro] PROSECCO : formally-PROven SEcured Compiled COde. https://wp-systeme.lip6.fr/prosecco/. (Citation page 69.)

[pri] Table of primitive polynomials from "Error Correction Coding : Mathematical Methods and Algorithms". http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/primitive-polynomial-table.txt. (Citation page 52.)

[Abadi et al., 2009] Abadi, M., Budiu, M., Erlingsson, U., et Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 13(1). (Citation page 92.)

[Agosta et al., 2013a] Agosta, G., Barenghi, A., Maggi, M., et Pelosi, G. (2013a). Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE. (Citations pages 21, 24, 27 et 36.)

[Agosta et al., 2012] Agosta, G., Barenghi, A., et Pelosi, G. (2012). A code morphing methodology to automate power analysis countermeasures. *DAC*, pages 77–82. (Citations pages xv, 10, 11, 21, 24, 25, 94, 109, 110 et 111.)

[Agosta et al., 2013b] Agosta, G., Barenghi, A., Pelosi, G., et Scandale, M. (2013b). Enhancing passive side-channel attack resilience through schedulability analysis of data-dependency graphs. In *International Conference on Network and System Security*, pages 692–698. Springer. (Citation page 10.)

[Agosta et al., 2015a] Agosta, G., Barenghi, A., Pelosi, G., et Scandale, M. (2015a). Information leakage chaff : feeding red herrings to side channel attackers. pages 1–6. ACM Press. (Citations pages 3, 21, 24 et 26.)

[Agosta et al., 2015b] Agosta, G., Barenghi, A., Pelosi, G., et Scandale, M. (2015b). The MEET approach : Securing cryptographic embedded software against side channel attacks. *IEEE TCAD*, 34(8) :1320–1333. (Citations pages xv, 10, 21, 24, 25, 94, 95, 109 et 110.)

[Agoyan et al., 2010] Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B., et Tria, A. (2010). When clocks fail : On critical paths and clock faults. *LNCS*, 6035 LNCS :182–193. (Citation page 3.)

[Agrawal et al., 2003] Agrawal, D., Archambeault, B., Rao, J., et Rohatgi, P. (2003). The em Side-Channel(s). *LNCS*, 2523 :29–45. (Citation page 2.)

[Akkar et al., 2003] Akkar, M.-L., Goubin, L., et Ly, O. (2003). Automatic integration of counter-measures against fault injection attacks. *Pre-print found at http ://www. labri. fr/Perso/ly/index. htm.* (Citations pages 21 et 23.)

[Ambrose et al., 2007] Ambrose, J., Ragel, R., et Parameswaran, S. (2007). RIJID : Random Code Injection to Mask Power Analysis based Side Channel Attacks. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 489–492. (Citation page 3.)

[Amiel et al., 2007] Amiel, F., Villegas, K., Feix, B., et Marcel, L. (2007). Passive and active combined attacks - Combining fault attacks and side channel analysis -. pages 92–99. (Citation page 9.)

[ANSSI, 2008] ANSSI (2008). Certification cspn. `https://www.ssi.gouv.fr/administration/produits-certifies/cspn/`. (Citation page 3.)

[Appel et Ginsburg, 2004] Appel, A. W. et Ginsburg, M. (2004). *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA. (Citation page 14.)

[Aumüller et al., 2003] Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., et Seifert, J.-P. (2003). Fault Attacks on RSA with CRT : Concrete Results and Practical Countermeasures. *LNCS*, 2523 :260–275. (Citation page 2.)

[Avirneni et Somani, 2014] Avirneni, N. D. P. et Somani, A. K. (2014). Countering Power Analysis Attacks UsingReliable and Aggressive Designs. *IEEE TOC*, 63(6) :1408–1420. (Citations pages 3 et 94.)

[Baek et Noh, 2005] Baek, Y.-J. et Noh, M.-J. (2005). Differential power attack and masking method. page 15. (Citations pages 37, 38 et 46.)

[Balasch et al., 2015] Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., et Standaert, F.-X. (2015). On the Cost of Lazy Engineering for Masked Software Implementations. *LNCS*, 8968 :64–81. (Citations pages 13, 15, 36 et 73.)

[Bar-El et al., 2006] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., et Whelan, C. (2006). The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2) :370–382. (Citation page 9.)

[Barbosa et al., 2009] Barbosa, M., Moss, A., et Page, D. (2009). Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography. *Journal of Cryptology*, 22(2) :259–281. (Citation page 15.)

[Barenghi et al., 2010] Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., et Regazzoni, F. (2010). Countermeasures against fault attacks on software implemented AES : effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security - WESS '10*, pages 1–10, Scottsdale, Arizona. ACM Press. (Citations pages 3, 13 et 14.)

[Barry, 2017] Barry, T. (2017). *Outillage de conception et de compilation de protections logicielles pour la sécurité dans les systèmes embarqués.* Theses, École nationale supérieure des mines de Saint-Etienne. (Citations pages 28 et 33.)

[Barry et al., 2016] Barry, T., Couroussé, D., et Robisson, B. (2016). Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6. ACM. (Citations pages 21, 24, 28, 29, 31 et 33.)

[Bayrak et al., 2011] Bayrak, A., Regazzoni, F., Brisk, P., Standaert, F.-X., et Ienne, P. (2011). A first step towards automatic application of power analysis countermeasures. pages 230–235. (Citations pages 10, 21 et 30.)

[Bayrak et al., 2015a] Bayrak, A., Regazzoni, F., Novo, D., Brisk, P., Standaert, F.-X., et Ienne, P. (2015a). Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 64(2) :329–341. (Citation page 10.)

[Bayrak et al., 2015b] Bayrak, A. G., Regazzoni, F., Novo, D., Brisk, P., Standaert, F.-X., et Ienne, P. (2015b). Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 64(2) :329–341. (Citations pages 10, 21, 24, 27, 30 et 36.)

[Bayrak et al., 2012] Bayrak, A. G., Velickovic, N., Ienne, P., et Burleson, W. (2012). An Architecture-independent Instruction Shuffler to Protect Against Side-channel Attacks. *ACM Trans. Archit. Code Optim.*, 8(4) :20 :1–20 :19. (Citation page 3.)

[Belaïd, 2015] Belaïd, S. (2015). *Security of Cryptosystems Against Power-Analysis Attacks*. Theses, ENS. (Citation page 36.)

[Belleville et Couroussé, 2018] Belleville, N. et Couroussé, D. (2018). Procédé d'exécution d'une fonction, par un microprocesseur, sécurisée par désynchronisation temporelle. (Citation page 112.)

[Ben El Ouahma et al., 2017] Ben El Ouahma, I., Meunier, Q., Heydemann, K., et Encrenaz, E. (2017). Symbolic Approach for Side-Channel Resistance Analysis of Masked Assembly Codes. pages 17–0. (Citations pages 32, 50 et 69.)

[Biehl et al., 2000] Biehl, I., Meyer, B., et Müller, V. (2000). Differential Fault Attacks on Ellitic Curve Cryptosystems. In Bellare, M., editor, *Advances in Cryptology (CRYPTO 2000)*, volume 1880 of *LNCS*, pages 131–146. Springer. (Citation page 9.)

[Boey et al., 2010] Boey, K. H., Lu, Y., O'Neill, M., et Woods, R. (2010). Random clock against differential power analysis. *Circuits and Systems (APCCAS), 2010 IEEE Asia Pacific Conference on*, pages 756–759. (Citation page 94.)

[Boneh et al., 1997] Boneh, D., DeMillo, R. A., et Lipton, R. J. (1997). On the Importance of Checking Cryptographic Protocols for Faults. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 37–51. Springer. (Citation page 2.)

[Boneh et al., 2001] Boneh, D., DeMillo, R. A., et Lipton, R. J. (2001). On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14 :101–119. (Citation page 9.)

[Bréjon et al., 2019] Bréjon, J.-B., Heydemann, K., Encrenaz, E., Meunier, Q. L., et Vu, S. T. (2019). Fault attack vulnerability assessment of binary code. In *Cryptography and Security in Computing Systems (CS2'19)*, Valencia, Spain. (Citation page 32.)

[Brier et al., 2004] Brier, E., Clavier, C., et Olivier, F. (2004). Correlation power analysis with a leakage model. *LNCS*, 3156 :16–29. (Citation page 2.)

[Bringer et al., 2014] Bringer, J., Carlet, C., Chabanne, H., Guilley, S., et Maghrebi, H. (2014). Orthogonal Direct Sum Masking : A Smartcard Friendly Computation Paradigm in a Code, with Builtin Protection against Side-Channel and Fault Attacks. In Naccache, D. et Sauveron, D., editors, *Information Security Theory and Practice. Securing the Internet of Things*, volume 8501, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 13.)

[Carlet et al., 2012] Carlet, C., Goubin, L., Prouff, E., Quisquater, M., et Rivain, M. (2012). Higher-order masking schemes for s-boxes. In *Fast Software Encryption*, pages 366–384. Springer. (Citation page 12.)

[Carpi et al., 2013] Carpi, R. B., Picek, S., Batina, L., Menarini, F., Jakobovic, D., et Golub, M. (2013). Glitch It If You Can : Parameter Search Strategies for Successful Fault Injection. In *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 236–252. Springer, Cham. (Citation page 2.)

[CCN, 2018] CCN (2018). Definición de la certificación nacional esencial de seguridad. `https://oc.ccn.cni.es/index.php/es/documentos/criterios-y-metodologias/52-ccn-lince-001-v0-1/file`. (Citation page 3.)

[Charles et al., 2014] Charles, H.-P., Couroussé, D., Lomüller, V., Endo, F., et Gauguey, R. (2014). deGoal a tool to embed dynamic code generators into applications. *LNCS*, 8409 :107–112. (Citation page 107.)

[Charles et Lomüller, 2015] Charles, H.-P. et Lomüller, V. (2015). Is dynamic compilation possible for embedded systems ? *SCOPES*, pages 80–83. (Citation page 107.)

[Chen et al., 2011] Chen, P., Fang, Y., Mao, B., et Xie, L. (2011). JITDefender : A defense against JIT spraying attacks. *IFIP AICT*, 354 :142–153. (Citation page 91.)

[Chen et al., 2013] Chen, P., Wu, R., et Mao, B. (2013). JITSafe : A framework against just-in-time spraying attacks. *IET Information Security*, 7(4) :283–292. (Citation page 91.)

[Chen et al., 2017] Chen, Z., Shen, J., Nicolau, A., Veidenbaum, A., et Farhady, N. (2017). CAMFAS : A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 57–64. IEEE. (Citations pages 21, 24 et 29.)

[Clavier et al., 2000] Clavier, C., Coron, J.-S., et Dabbous, N. (2000). Differential Power Analysis in the Presence of Hardware Countermeasures. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, Lecture Notes in Computer Science, pages 252–263. Springer, Berlin, Heidelberg. (Citation page 10.)

[Colombier et al., 2019] Colombier, B., Menu, A., Dutertre, J.-M., Moellic, P.-A., Rigaud, J.-B., et Danger, J.-L. (2019). Laser-induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-bit Microcontroller. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10, McLean, VA, USA. IEEE. (Citation page 9.)

[Coron, 2014] Coron, J.-S. (2014). Higher Order Masking of Look-Up Tables. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Kobsa, A., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Terzopoulos, D., Tygar, D., Weikum, G., Nguyen, P. Q., et Oswald, E., editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441, pages 441–458. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 12.)

[Coron et al., 2014a] Coron, J.-S., Groszschaedl, J., Vadnala, P. K., et Tibouchi, M. (2014a). Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. Technical Report 891. (Citation page 12.)

[Coron et Kizhvatov, 2009] Coron, J.-S. et Kizhvatov, I. (2009). An efficient method for random delay generation in embedded software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5747 LNCS :156–170. (Citations pages 10 et 94.)

[Coron et Kizhvatov, 2010] Coron, J.-S. et Kizhvatov, I. (2010). Analysis and improvement of the random delay countermeasure of CHES 2009. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6225 LNCS :95–109. (Citations pages 10 et 94.)

[Coron et al., 2014b] Coron, J.-S., Prouff, E., Rivain, M., et Roche, T. (2014b). Higher-Order Side Channel Security and Mask Refreshing. In Moriai, S., editor, *Fast Software Encryption*, volume 8424, pages 410–424. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 59.)

[Coron et al., 2018] Coron, J.-S., Rondepierre, F., et Zeitoun, R. (2018). High order masking of look-up tables with common shares. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1) :40–72. (Citation page 12.)

[Coron et al., 2014c] Coron, J.-S., Roy, A., et Vivek, S. (2014c). Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 170–187. Springer. (Citations pages 12, 36, 51, 54, 59 et 73.)

[Couroussé et al., 2016] Couroussé, D., Barry, T., Robisson, B., Jaillon, P., Potin, O., et Lanet, J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. *WISTP*, 9895 :136–152. (Citations pages 3, 10, 11, 21, 22, 23, 76, 77, 79, 82, 84, 94 et 109.)

[Crane et al., 2015] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., et Franz, M. (2015). Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. Internet Society. (Citation page 11.)

[Daemen et Rijmen, 2002] Daemen, J. et Rijmen, V. (2002). *The Design of Rijndael : AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer-Verlag, Berlin Heidelberg. (Citation page 8.)

[Danger et al., 2014] Danger, J.-L., Guilley, S., Porteboeuf, T., Praden, F., et Timbert, M. (2014). HCODE : Hardware-Enhanced Real-Time CFI. pages 1–11. ACM Press. (Citation page 3.)

[Dassance et Venelli, 2012] Dassance, F. et Venelli, A. (2012). Combined fault and side-channel attacks on the AES key schedule. pages 63–71. (Citation page 9.)

[de Clercq et Verbauwhede, 2017] de Clercq, R. et Verbauwhede, I. (2017). A survey of Hardware-based Control Flow Integrity (CFI). *arXiv preprint arXiv :1706.07257*. (Citation page 13.)

[De Keulenaer et al., 2016] De Keulenaer, R., Maebe, J., De Bosschere, K., et De Sutter, B. (2016). Link-time smart card code hardening. *International Journal of Information Security*, 15(2) :111–130. (Citations pages 3, 21 et 31.)

[Dehbaoui et al., 2012] Dehbaoui, A., M Dutertre, J., Robisson, B., Orsatelli, P., Maurine, P., et Tria, A. (2012). Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system. (Citation page 2.)

[Dehbaoui et al., 2013] Dehbaoui, A., Mirbaha, A., Moro, N., Dutertre, J., et Tria, A. (2013). Electromagnetic glitch on the AES round counter. In *COSADE*. (Citation page 9.)

[Dhem et al., 2000] Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., et Willems, J.-L. (2000). A practical implementation of the timing attack. *LNCS*, 1820 :167–182. (Citation page 2.)

[Dinu et al., 2016] Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Gro\s sschädl, J., et Biryukov, A. (2016). Design Strategies for ARX with Provable Bounds : SPARX and LAX (Full Version). *IACR Cryptology ePrint Archive*, 2016 :984. (Citation page 12.)

[Dureuil, 2016] Dureuil, L. (2016). *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute*. PhD thesis, Communauté Université Grenoble Alpes. (Citation page 9.)

[Dureuil et al., 2016] Dureuil, L., Petiot, G., Potet, M.-L., Le, T.-H., Crohen, A., et de, C. (2016). FISSC : A fault injection and simulation secure collection. *LNCS*, 9922 :3–11. (Citation page 105.)

[Dureuil et al., 2015] Dureuil, L., Potet, M., de Choudens, P., Dumas, C., et Clédière, J. (2015). From code review to fault injection attacks : Filling the gap using fault model inference. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, pages 107–124. (Citation page 9.)

[Durvaux et al., 2013] Durvaux, F., Renauld, M., Standaert, F.-X., van Oldeneel tot Oldenzeel, L., et Veyrat-Charvillon, N. (2013). Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models. *CARDIS*, pages 123–140. (Citations pages 10 et 84.)

[Dusart et al., 2003] Dusart, P., Letourneux, G., et Vivolo, O. (2003). Differential Fault Analysis on AES. In Yung, M., Han, Y., et Zhou, J., editors, *Applied Cryptography and Network Security (ANCS 2003)*, volume 2846 of *LNCS*, pages 293–306. Springer. (Citation page 9.)

[Dutertre et al., 2014] Dutertre, J.-M., De, C., Sarafianos, A., Boher, N., Rouzeyre, B., Lisart, M., Damiens, J., Candelier, P., Flottes, M.-L., et Di, N. (2014). Laser attacks on integrated circuits : From CMOS to FD-SOI. (Citation page 2.)

[Eldib et Wang, 2014] Eldib, H. et Wang, C. (2014). Synthesis of Masking Countermeasures Against Side Channel Attacks. In *International Conference on Computer Aided Verification*, pages 114–130. Springer. (Citations pages 21, 22, 23, 24, 38, 47 et 50.)

[Famien, 2017] Famien, O. (2017). Attaque ddos : plus de 5000 objets connectés d'une université utilisée pour paralyser son propre réseau. https://www.developpez.com/actu/117426/Attaque-DDoS-plus-de-5000-objets-connectes-d-une-universite-utilisee-pour-paralyser-son-propre-reseau-Verizon-donne-des-details-dans-un-rapport/. (Citation page 2.)

[Fan et al., 2011] Fan, J., Gierlichs, B., et Vercauteren, F. (2011). To infinity and beyond : Combined attack on ECC using points of low order. *LNCS*, 6917 LNCS :143–159. (Citation page 9.)

[Federal Office for Information Security, 2019] Federal Office for Information Security (2019). Bsz certification. https://www.bsi.bund.de/EN/Topics/Certification/product_certification/Accelerated_Security_Certification/Accelerated-Security-Certification_node.html. (Citation page 3.)

[Gandolfi et al., 2001] Gandolfi, K., Mourtel, C., et Olivier, F. (2001). Electromagnetic analysis : Concrete results. *LNCS*, 2162 :251–261. (Citation page 2.)

[Genelle et al., 2010] Genelle, L., Prouff, E., et Quisquater, M. (2010). Secure Multiplicative Masking of Power Functions. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Zhou, J., et Yung, M., editors, *Applied Cryptography and Network Security*, volume 6123, pages 200–217. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citations pages 11 et 12.)

[Genelle et al., 2011] Genelle, L., Prouff, E., et Quisquater, M. (2011). Montgomery's Trick and Fast Implementation of Masked AES. In Nitaj, A. et Pointcheval, D., editors, *Progress in Cryptology – AFRICACRYPT 2011*, volume 6737, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 11.)

[Genkin et al., 2017] Genkin, D., Shamir, A., et Tromer, E. (2017). Acoustic Cryptanalysis. *Journal of Cryptology*, 30(2) :392–443. (Citation page 2.)

[Goodwill et al., 2011] Goodwill, G., Jun, B., Josh, J., Pankaj, R., et al. (2011). A testing methodology for side-channel resistance validation. *NIST non-invasive attack testing workshop*. (Citations pages 17 et 18.)

[Gornik et al., 2015] Gornik, A., Moradi, A., Oehm, J., et Paar, C. (2015). A hardware-based countermeasure to reduce side-channel leakage : Design, implementation, and evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8) :1308–1319. (Citation page 94.)

[Goubin, 2001] Goubin, L. (2001). A sound method for switching between boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer. (Citation page 12.)

[Goubin et Patarin, 1999] Goubin, L. et Patarin, J. (1999). DES and Differential Power Analysis (The "Duplication" Method). In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 158–172, London, UK, UK. Springer-Verlag. (Citation page 11.)

[Güneysu et Moradi, 2011] Güneysu, T. et Moradi, A. (2011). Generic side-channel countermeasures for reconfigurable devices. *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 33–48. (Citations pages 3 et 94.)

[Homescu et al., 2013] Homescu, A., Brunthaler, S., Larsen, P., et Franz, M. (2013). Librando : transparent code randomization for just-in-time compilers. pages 993–1004. ACM Press. (Citation page 11.)

[Hutter et Schmidt, 2014] Hutter, M. et Schmidt, J.-M. (2014). The temperature side channel and heating fault attacks. *LNCS*, 8419 LNCS :219–235. (Citation page 3.)

[INRIA, 2019] INRIA (2019). *Libre blanc cybersécurité*. (Citation page 2.)

[Ishai et al., 2003] Ishai, Y., Sahai, A., et Wagner, D. (2003). Private Circuits : Securing Hardware against Probing Attacks. In Goos, G., Hartmanis, J., van Leeuwen, J., et Boneh, D., editors, *Advances in Cryptology - CRYPTO 2003*, volume 2729, pages 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 11.)

[Jauernig et al., 2014] Jauernig, M., Neugschwandtner, M., Platzer, C., et Comparetti, P. (2014). Lobotomy : An architecture for JIT spraying mitigation. *ARES*, pages 50–58. (Citation page 91.)

[Josset, 2016] Josset, C. (2016). Cyberattaque massive : une c̈apacité d'attaque phénoménaleä été déployée. https://lexpansion.lexpress.fr/high-tech/cyberattaque-massive-une-capacite-d-attaque-phenomenale-a-ete-deployee_1843531.html. (Citation page 2.)

[Karroumi et al., 2014] Karroumi, M., Richard, B., et Joye, M. (2014). Addition with blinded operands. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 41–55. Springer. (Citation page 12.)

[Khandouzy et Azizzadeh, 2015] Khandouzy, M. et Azizzadeh, N. (2015). SIMON 32/64. https://github.com/mkhandouzy/SIMON-32-64. (Citation page 67.)

[Kocher, 1996] Kocher, P. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer. (Citation page 2.)

[Kocher et al., 1999] Kocher, P., Jaffe, J., et Jun, B. (1999). Differential power analysis. *LNCS*, 1666 :388–397. (Citation page 2.)

[Lalande et al., 2014] Lalande, J.-F., Heydemann, K., et Berthomé, P. (2014). Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer. (Citations pages 21 et 23.)

[Laurent et al., 2019] Laurent, J., Beroulle, V., Deleuze, C., et Pebay-Peyroula, F. (2019). Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. pages 252–255. (Citation page 73.)

[Le Corre et al., 2018] Le Corre, Y., Großschädl, J., et Dinu, D. (2018). Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In Fan, J. et Gierlichs, B., editors, *Constructive Side-Channel Analysis and Secure Design*, volume 10815, pages 82–98. Springer International Publishing, Cham. (Citations pages 71 et 115.)

[Luo et al., 2017] Luo, P., Athanasiou, K., Zhang, L., Jiang, Z. H., Fei, Y., Ding, A. A., et Wahl, T. (2017). Compiler-Assisted Threshold Implementation against Power Analysis Attacks. pages 541–544. IEEE. (Citations pages 21, 24 et 27.)

[Luo et al., 2015] Luo, P., Zhang, L., Fei, Y., et Ding, A. A. (2015). Towards secure cryptographic software implementation against side-channel power analysis attacks. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 144–148. IEEE. (Citations pages 10, 20 et 21.)

[Malagón et al., 2012] Malagón, P., de, G., Zapater, M., Moya, J., et Banković, Z. (2012). Compiler optimizations as a countermeasure against side-channel analysis in MSP430-based devices. *Sensors (Switzerland)*, 12(6) :7994–8012. (Citations pages 21, 24 et 25.)

[Mangard et al., 2007] Mangard, S., Oswald, E., et Popp, T. (2007). *Power Analysis attacks : Revealing the secrets of smart cards*. Power Analysis Attacks : Revealing the Secrets of Smart Cards. DOI : 10.1007/978-0-387-38162-6. (Citations pages 2, 3, 8, 11, 12 et 85.)

[Mathieu-Mahias et Quisquater, 2018] Mathieu-Mahias, A. et Quisquater, M. (2018). Mixing Additive and Multiplicative Masking for Probing Secure Polynomial Evaluation Methods. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1) :175–208. (Citation page 59.)

[Menicocci et al., 2014] Menicocci, R., Trifiletti, A., et Trotta, F. (2014). Experiments on two clock countermeasures against power analysis attacks. *2014 Proceedings of the 21st International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)*, pages 215–219. (Citation page 94.)

[Moon, 2005] Moon, T. (2005). *Error Correction Coding : Mathematical Methods and Algorithms*. (Citation page 52.)

[Moradi et Wild, 2015] Moradi, A. et Wild, A. (2015). Assessment of Hiding the Higher-Order Leakages in Hardware. *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 453–474. (Citation page 94.)

[Moro, 2014] Moro, N. (2014). *Security of assembly programs against fault attacks on embedded processors*. Theses, Université Pierre et Marie Curie - Paris VI. (Citation page 30.)

[Moro et al., 2013] Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., et Encrenaz, E. (2013). Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE. (Citation page 9.)

[Moro et al., 2014] Moro, N., Heydemann, K., Encrenaz, E., et Robisson, B. (2014). Formal Verification of a Software Countermeasure Against Instruction Skip Attacks. *Journal of Cryptographic Engineering*, 4(3) :145–156. (Citations pages 14, 21 et 33.)

[Moss et al., 2012] Moss, A., Oswald, E., Page, D., et Tunstall, M. (2012). Compiler assisted masking. *LNCS*, 7428 LNCS :58–75. (Citations pages 21, 24, 26, 36 et 47.)

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (Citation page 14.)

[Ordas et al., 2014] Ordas, S., Carbone, M., Ducharme, G., Tiran, S., et Maurine, P. (2014). Efficiency of the RDVFS countermeasure. *Faible Tension Faible Consommation (FTFC), 2014 IEEE*, pages 1–4. (Citation page 94.)

[Ordas et al., 2015a] Ordas, S., Guillaume-Sage, L., et Maurine, P. (2015a). EM injection : Fault model and locality. *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 00 :3–13. (Citation page 9.)

[Ordas et al., 2015b] Ordas, S., Guillaume-Sage, L., Tobich, K., Dutertre, J.-M., et Maurine, P. (2015b). Evidence of a larger EM-induced fault model. *LNCS*, 8968 :245–259. (Citations pages 2 et 9.)

[Peeters, 2013] Peeters, E. (2013). *Advanced DPA theory and practice : Towards the security limits of secure embedded circuits*, volume 9781461467830 of *Advanced DPA Theory and Practice : Towards the Security Limits of Secure Embedded Circuits*. DOI : 10.1007/978-1-4614-6783-0. (Citation page 2.)

[Proy et al., 2017] Proy, J., Heydemann, K., Berzati, A., et Cohen, A. (2017). Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.*, 14(4) :36 :1–36 :25. (Citations pages 21, 24, 28 et 29.)

[Quisquater et Samyde, 2001] Quisquater, J.-J. et Samyde, D. (2001). ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards. In *Smart Card Programming and Security*, Lecture Notes in Computer Science, pages 200–210. Springer, Berlin, Heidelberg. DOI : 10.1007/3-540-45418-7_17. (Citation page 2.)

[Rane et al., 2015] Rane, A., Lin, C., et Tiwari, M. (2015). Raccoon : Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX*

Conference on Security Symposium, SEC'15, pages 431–446, Berkeley, CA, USA. USENIX Association. (Citation page 11.)

[Rauzy et al., 2016] Rauzy, P., Guilley, S., et Najm, Z. (2016). Formally proved security of assembly code against power analysis : A case study on balanced logic. *Journal of Cryptographic Engineering*, 6(3) :201–216. (Citations pages 21 et 30.)

[Reis et al., 2005] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., et August, D. I. (2005). SWIFT : Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society. (Citations pages 21, 24 et 28.)

[Rivain et Prouff, 2010] Rivain, M. et Prouff, E. (2010). Provably Secure Higher-Order Masking of AES. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Mangard, S., et Standaert, F.-X., editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225, pages 413–427. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citations pages 12, 37 et 46.)

[Rivain et al., 2009] Rivain, M., Prouff, E., et Doget, J. (2009). Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science, pages 171–188. Springer, Berlin, Heidelberg. DOI : 10.1007/978-3-642-04138-9_13. (Citation page 10.)

[Roche et al., 2011] Roche, T., Lomné, V., et Khalfallah, K. (2011). Combined fault and side-channel attack on protected implementations of AES. *LNCS*, 7079 LNCS :65–83. (Citation page 9.)

[Roy et Vivek, 2013] Roy, A. et Vivek, S. (2013). Analysis and Improvement of the Generic Higher-Order Masking Scheme of FSE 2012. In Bertoni, G. et Coron, J.-S., editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086, pages 417–434. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 12.)

[Sasdrich et al., 2017] Sasdrich, P., Moradi, A., et Güneysu, T. (2017). Hiding higher-order side-channel leakage. *CT-RSA*, pages 131–146. (Citations pages 3 et 94.)

[Schneider et Moradi, 2015] Schneider, T. et Moradi, A. (2015). Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. *WISE*, (207). (Citations pages 17 et 18.)

[Seuschek et al., 2017] Seuschek, H., De Santis, F., et Guillen, O. M. (2017). Side-channel leakage aware instruction scheduling. pages 7–12. ACM Press. (Citation page 15.)

[Seuschek et Rass, 2015] Seuschek, H. et Rass, S. (2015). Side-channel leakage models for risc instruction set architectures from empirical data. *EUROMICRO DSD*, pages 423–430. (Citation page 96.)

[Sifakis, 2011] Sifakis, J. (2011). A vision for computer science — the system perspective. *Open Computer Science*, 1(1). (Citation page 2.)

[Singh et al., 2018] Singh, A., Kar, M., Mathew, S., Rajan, A., De, V., et Mukhopadhyay, S. (2018). Exploiting on-chip power management for side-channel security. *DATE*, pages 401–406. (Citations pages 3 et 94.)

[Skorobogatov, 2009] Skorobogatov, S. (2009). Local heating attacks on flash memory devices. pages 1–6. (Citation page 3.)

[Skorobogatov et Anderson, 2003] Skorobogatov, S. et Anderson, R. (2003). Optical Fault Induction Attacks. *LNCS*, 2523 :2–12. (Citation page 2.)

[Srikant et Shankar, 2007] Srikant, Y. et Shankar, P. (2007). *The Compiler Design Handbook : Optimizations and Machine Code Generation, Second Edition.* CRC Press. (Citation page 14.)

[Sung-Ming Yen et Joye, 2000] Sung-Ming Yen et Joye, M. (2000). Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9) :967–970. (Citation page 9.)

[Tang et al., 2017] Tang, M., Qiu, Z., Guo, Z., Mu, Y., Huang, X., et Danger, J.-L. (2017). A Generic Table Recomputation-Based Higher-Order Masking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(11) :1779–1789. (Citation page 12.)

[Timmers et Spruyt, 2016] Timmers, N. et Spruyt, A. (2016). Bypassing secure boot using fault injection. (Citation page 9.)

[Tunstall et Benoit, 2007] Tunstall, M. et Benoit, O. (2007). Efficient Use of Random Delays in Embedded Software. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, Lecture Notes in Computer Science, pages 27–38. Springer, Berlin, Heidelberg. (Citations pages 10 et 85.)

[Tunstall et al., 2014] Tunstall, M., Whitnall, C., et Oswald, E. (2014). Masking Tables—An Underestimated Security Risk. In Moriai, S., editor, *Fast Software Encryption*, volume 8424, pages 425–444. Springer Berlin Heidelberg, Berlin, Heidelberg. (Citation page 12.)

[Van Put et al., 2005] Van Put, L., Chanet, D., De Bus, B., De Sutter, B., et De Bosschere, K. (2005). DIABLO : a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12, Athens. IEEE. (Citation page 31.)

[VanLaven et al., 2005] VanLaven, J., Brehob, M., et Compton, K. (2005). A computationally feasible SPA attack on AES via optimized search. *IFIP Advances in Information and Communication Technology*, 181 :577–588. (Citation page 2.)

[Wang et al., 2017] Wang, C., Yan, M., Cai, Y., Zhou, Q., et Yang, J. (2017). Power Profile Equalizer : A Lightweight Countermeasure against Side-Channel Attack. pages 305–312. (Citation page 3.)

[Wang et al., 2019] Wang, J., Sung, C., et Wang, C. (2019). Mitigating Power Side Channels during Compilation. *arXiv :1902.09099 [cs].* arXiv : 1902.09099. (Citations pages 36, 50 et 73.)

[Wang et al., 2013] Wang, X., Yueh, W., Roy, D. B., Narasimhan, S., Zheng, Y., Mukhopadhyay, S., Mukhopadhyay, D., et Bhunia, S. (2013). Role of power grid in side channel attack and power-grid-aware secure design. *Proceedings of the 50th Annual Design Automation Conference*, page 78. (Citation page 94.)

[Yu et Kose, 2018] Yu, W. et Kose, S. (2018). Exploiting Voltage Regulators to Enhance Various Power Attack Countermeasures. *IEEE TETC*, 6(2) :244–257. (Citation page 94.)

[Yuce et al., 2018] Yuce, B., Schaumont, P., et Witteman, M. (2018). Fault attacks on secure embedded software : Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2) :111–130. (Citation page 9.)

Résumé

Les systèmes embarqués et objets connectés sont aujourd'hui de plus en plus répandus. Contrairement à d'autres systèmes accessibles uniquement par le réseau, les systèmes embarqués sont accessibles physiquement par un attaquant. Celui-ci peut alors exploiter cette proximité physique pour monter des attaques par canal auxiliaire afin de compromettre ces systèmes ou leurs données. Ces attaques non intrusives ont ainsi montré une grande efficacité pour récupérer les clés cryptographiques utilisées dans de tels systèmes. Il est alors primordial de protéger les systèmes embarqués contre cette menace sérieuse. Les contre-mesures logicielles sont la plupart du temps appliquées manuellement par des experts. Dans cette thèse, nous proposons d'appliquer automatiquement ces contre-mesures au sein du processus de compilation. Nous proposons deux approches, l'une pour appliquer une contre-mesure de masquage booléen de premier ordre, l'autre pour appliquer une contre-mesure de polymorphisme de code. Nous apportons des réponses à plusieurs problèmes liés à la génération dynamique de code pour permettre l'utilisation du polymorphisme de code sur des systèmes contraints. Enfin, nous adaptons les contre-mesures choisies afin d'obtenir de meilleurs compromis entre performances et sécurité.

Abstract

Embedded systems and connected objects are increasingly used nowadays. Unlike some other systems accessible only through the network, embedded systems are physically accessible by an attacker. The latter can then exploit this physical proximity to mount side-channel attacks to compromise these systems or their data. These non-intrusive attacks have shown great effectiveness in recovering cryptographic keys used in such systems. Embedded systems must therefore be secured against this severe threat. Software countermeasures are most often applied manually by experts. In this thesis, we propose to automatically apply these countermeasures within the compilation process. We propose two approaches, one to apply a first-order Boolean masking countermeasure, the other to apply a code polymorphism countermeasure. We address several problems related to dynamic code generation to enable the use of code polymorphism on constrained systems. Finally, we adapt the chosen countermeasures to obtain a better trade-off between performance and security.