Université Grenoble Alpes

CEA

Unité de recherche **CEA List, Dpt. Systèmes et Circuits Intégrés Numériques (DSCIN)**

Habilitation à diriger les recherches présentée par **Damien Couroussé**

Soutenue le **28 août 2024**

Spécialité **Mathématiques appliquées et informatique**

Application outillée de contre-mesures contre les attaques matérielles

Composition du jury

Rapporteurs	Guy Gogniat	professeur à l'Université Bretagne Sud
	Guillaume Hiet	professeur à CentraleSupélec Rennes
	Ingrid Verbauwhede	professeure à KU Leuven
Examinateurs	Aurélien Francillon	professeur à EURECOM
	Karine Heydemann	experte sécurité à Thalès DIS, chercheuse associée au LIP6
	David Hély	professeur à LCIS, Université Grenoble Alpes
	Marie-Laure Potet	professeure à Université Grenoble Alpes

[git] ▪ 0.6-0-g2545c0f @ master ▪ 2024-07-10 18:20:41 +0200

Colophon

Mémoire de thèse intitulé « Application outillée de contre-mesures contre les attaques matérielles », écrit par Damien Courroussé, achevé le 10 juillet 2024, composé au moyen du système de préparation de document LaTeX et de la classe yathesis dédiée aux thèses préparées en France.

Table des matières

[git] ▪ 0.6-0-g2545c0f @ master ▪ 2024-07-10 18:20:41 +0200

Chapitre 1

Introduction

Ce mémoire d'habilitation porte sur mon activité de recherche sur la sécurisation des systèmes embarqués contre les attaques par canal auxiliaire et par injection de fautes, de 2013 jusqu'à présent. J'ai obtenu un doctorat en 2008 sur l'intégration, dans une situation instrumentale multisensorielle, d'une technologie de retour d'effort et de modèles physiques simulés en temps réel. J'ai ensuite travaillé dans l'industrie comme expert en logiciel embarqué tout en gardant un pied dans des projets collaboratifs de recherche. Je suis arrivé au CEA en 2011, où j'ai d'abord travaillé sur l'optimisation de cœurs de calcul dans les systèmes embarqués, en collaboration avec Henri-Pierre Charles dont les recherches portaient sur la spécialisation du code au runtime. Dans la thèse de Fernando Endo [End15], nous avons montré que la spécialisation de code sur les données permettait d'obtenir, sur des noyaux de calcul exécutés sur des cœurs *in-order*, un temps d'exécution équivalent à celui obtenu sur des cœurs *out-of-order*, avec une meilleure performance énergétique [ECC17]. C'est en m'intéressant aux techniques de mesure de consommation de l'énergie consommée par les processeurs que j'ai découvert les attaques par canal auxiliaire. J'ai proposé une adaptation de la technologie sur laquelle nous travaillions alors, deGoal [Cha+14b], qui exploitait la génération de code au runtime pour la spécialisation de code sur les données, pour en faire une contre-mesure contre les attaques par canal auxiliaire, appelée plus tard *polymorphisme de code* [Cou13]. La sécurisation contre les attaques matérielles, d'abord du logiciel, puis d'un système englobant logiciel et matériel, est devenue progressivement ma thématique de recherche principale.

1.1 Contexte

Les attaques matérielles exploitent les caractéristiques physiques d'un circuit pour altérer une propriété de sécurité au niveau logique du système ciblé. Ces attaques exploitent deux principes : *observer* le comportement physique du système cible (attaques par canal auxiliaire), et *perturber* le comportement physique (attaques par injection de fautes). Les attaques par canal auxiliaire exploitent des observations physiques du circuit en fonctionnement pour inférer des informations additionnelles, non prévues dans un modèle d'attaquant en cryptanalyse pure, permettant d'altérer une propriété de sécurité, typiquement la confidentialité d'informations manipulées pendant un calcul. Les attaques par injection de fautes exploitent des perturbations physiques du circuit pour induire des effets inattendus dans son fonctionnement. Les conséquences logiques de ces perturbations matérielles permettent d'outrepasser la sécurité du système attaqué, par exemple faire fuir un secret ou obtenir des droits d'administration.

Les premières publications académiques sur ce sujet datent des années 1990, et l'industrie

a rapidement intégré des contre-mesures dans des produits de sécurité comme la carte à puce, qui est alors emblématique de la sécurité matérielle. Au début des années 2010, la carte à puce intègre un grand nombre de contre-mesures à différents niveaux : détection et protection contre les perturbations physiques par des capteurs dédiés intégrés dans le silicium, contre-mesures numériques dans l'architecture matérielle, et contre-mesures logicielles. Les différents éléments d'un système, logiciels et matériels, et leurs contre-mesures sont implémentés avec les outils du domaine en usage : outils de synthèse, compilateur, etc. Cependant, ces outils ne savent pas traiter les particularités de la sécurisation. Pire, il faut veiller à ce que les outils ne nuisent pas à l'implémentation des contre-mesures. Pour illustrer ces propos, considérons par exemple la redondance, un principe de protection courant contre les perturbations induites par les fautes. Les flots de synthèse et les compilateurs emploient des stratégies d'optimisation capables d'identifier et supprimer certaines redondances, inutiles du point de vue fonctionnel, qui peuvent nuire à la taille d'un circuit, à la taille ou au temps d'exécution d'un programme. Il est crucial de s'assurer que ces optimisations ne nuisent pas aux objectifs de protection des contre-mesures. De plus, la mécanique interne de ces outils est opaque pour l'utilisateur : les stratégies d'optimisation peuvent être très nombreuses, elles ne sont pas toujours documentées, et il est difficile, voire impossible, de prévoir le résultat de la combinaison d'un jeu d'optimisations sur l'implémentation finale. En conséquence, le spécialiste en charge de la sécurisation d'une implémentation doit faire appel à son savoir-faire et à un folklore d'heuristiques pour contourner le fonctionnement nominal des outils, de sorte que le schéma de sécurisation soit correct dans l'implémentation finale. Dans le pire des cas, il devient nécessaire de se passer de ces outils, par exemple écrire un programme en code assembleur pour éviter de faire appel au compilateur.

Dans le cas des implémentations de cryptographie, les premières concernées par les attaques matérielles, le principe de fonctionnement est relativement stable car les primitives dans le domaine public font l'objet de standardisations qui évoluent lentement. L'effort d'implémentation qu'implique l'absence d'outils adéquats est amorti sur la durée de vie des implémentations : une IP cryptographique ou une bibliothèque logicielle peuvent être utilisées dans plusieurs produits. De plus, si une vulnérabilité est identifiée, il est souvent jugé plus aisé de corriger directement l'implémentation concernée plutôt que de mettre à jour un outil de sécurisation potentiel pour ensuite produire une nouvelle implémentation. En l'absence d'outils cependant, chaque intervention humaine augmente le risque d'introduire une erreur dans l'implémentation.

Pour d'autres domaines applicatifs en revanche, les implémentations peuvent évoluer rapidement ou être dépendantes du produit ciblé. Prenons le cas de la télévision à péage, où des schémas d'intégrité du flot de contrôle, comme le schéma de traceur présenté en section 2.1.2, peuvent être utilisés. L'implémentation de la contre-mesure dépend du flot de contrôle à protéger, qui lui peut être lié à l'implémentation d'une couche logicielle applicative. Chaque modification du produit peut nécessiter un ajustement de la contre-mesure, et la modification est susceptible d'impacter une quantité importante de code si la contre-mesure couvre un grand programme, au flot de contrôle complexe. Dans ce cas, le bénéfice potentiel d'outils d'aide à la sécurisation est plus immédiat.

1.2 Questions de recherche et organisation du mémoire

J'ai commencé par m'intéresser à la question des outils pour la sécurisation des implémentations logicielles contre les attaques matérielles, plus particulièrement, au potentiel du compilateur pour l'application de contre-mesures logicielles contre les attaques matérielles. Cette question est traitée dans le chapitre 2, sous deux aspects. (i) Montrer qu'il est possible d'exploiter le compilateur pour l'application de contre-mesures. Cette question n'est pas triviale puisque, comme

évoqué ci-dessus, un compilateur traditionnel s'intéresse seulement aux propriétés fonctionnelles du programme cible, alors que dans l'application de contre-mesures on s'intéresse à des propriétés de sécurité, non fonctionnelles. Cet axe concerne également le potentiel du compilateur pour optimiser la contre-mesure elle-même et le code ciblé par la protection. (ii) L'utilisation du compilateur comme levier pour la sécurisation de programmes : on cherche à exploiter le compilateur pour augmenter le potentiel d'une contre-mesure ou pour la rendre facilement paramétrable, donc plus flexible d'application. Aussi, le compilateur peut tout simplement permettre l'implémentation de contre-mesures qui seraient autrement impossibles à mettre en œuvre en pratique.

Dans l'objectif d'adresser des modèles d'attaquants plus puissants, il devenait évident que des schémas de protection logiciels, seuls, ne seraient pas suffisants. Cette problématique est traitée dans le chapitre 3 sous deux aspects. (i) Comment exploiter des contre-mesures matérielles et les articuler avec des contre-mesures logicielles pour obtenir une protection à la couverture large. (ii) Comment articuler un schéma de protection logiciel avec des éléments de protections matérielles pour obtenir un niveau de protection global élevé.

Le chapitre 4 conclut le mémoire et ouvre des perspectives de recherche.

La figure 1.1 donne un aperçu chronologique des jalons majeurs au cours de mes activités de recherche liées à ce mémoire. Cette figure reprend : (i) les thèses déjà soutenues ; (ii) les projets collaboratifs dans lesquels je me suis particulièrement impliqué et qui ont contribué à la réalisation des travaux décrits dans ce mémoire ; (iii) les résultats scientifiques les plus marquants ainsi que les publications scientifiques associées les plus représentatives.

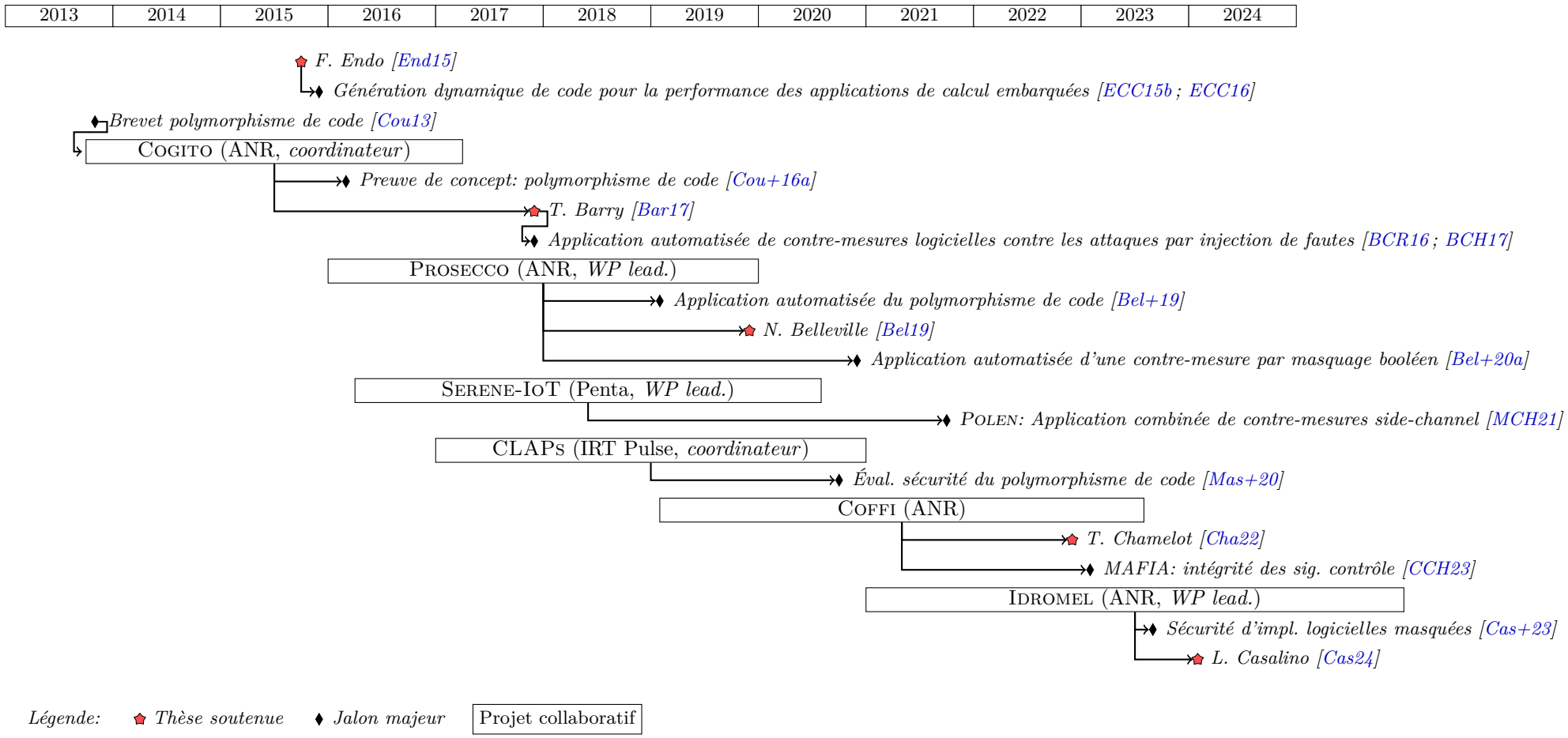


FIGURE 1.1 – Chronologie des jalons principaux relatifs aux travaux présentés dans ce mémoire

Chapitre 2

Tissage de code pour la sécurisation contre les attaques matérielles

Le compilateur occupe une place centrale dans un flot de production de logiciel, et l'infrastructure d'un compilateur offre une grande variété de techniques d'analyse, de modification ou d'instrumentation de programmes. De fait, le compilateur est donc une place de choix pour une application outillée, si possible automatisée, de contre-mesures. En cyber-sécurité, le support du compilateur est parfois exploité pour la sécurité logicielle (e.g. protection contre le buffer overflow [CH01]), ou pour la sûreté de fonctionnement (e.g. [Rei+05]) qui exploite des principes de protection similaires aux techniques de protection contre les injections de fautes.

Lors du démarrage de ces activités de recherche, dans le domaine de la sécurité matérielle, cette thématique de la compilation de contre-mesures est encore balbutiante. À notre connaissance, il n'existe pas encore de travaux concernant la compilation de contre-mesures contre les attaques par injection de fautes. Concernant les attaques par canaux auxiliaires, Amarilli et al. questionnent l'utilisation du compilateur pour la protection contre le side-channel [Ama+11], Agosta et al. publient une technique de protection très proche du polymorphisme de code [ABP12] que nous découvrons lors de la finalisation du dépôt du projet COGITO à l'ANR. Dans cette même période, la communauté commence à prendre en compte les effets potentiellement néfastes de la compilation logicielle sur la sécurité attendue des protections [Bal+15]. Pour aborder cette question de recherche, mes premiers travaux visaient à montrer le potentiel d'utilisation du compilateur dans le cadre de la protection contre les injections de fautes : pour une application plus efficace des contre-mesures en exploitant les techniques d'optimisations offertes par le compilateur, et pour une plus grande flexibilité des modèles de contre-mesures. Ce point est développé dans la première partie de ce chapitre, en section 2.1.

Les implémentations de fonctions cryptographiques sont les cibles privilégiées des attaques matérielles. De fait, beaucoup de contre-mesures de la littérature adressent uniquement certaines classes de fonctions cryptographiques, voire une primitive spécifique comme AES. Cependant, dans le cadre d'une application outillée de contre-mesures, on souhaite que la contre-mesure soit applicable à n'importe quel programme susceptible de bénéficier d'une protection contre les attaques matérielles. Cette question fut sous-jacente aux travaux présentés dans la deuxième partie de ce chapitre, en section 2.2. Premièrement dans le choix de cibler une technique de protection par dissimulation, applicable sans nécessiter de modifications algorithmiques du programme ciblé

(section 2.2.1). Deuxièmement dans la mise en œuvre d'un schéma de masquage à la compilation en levant les verrous qui en restreignent l'application (support du flot de contrôle, masquage de tables associatives) (section 2.2.2). Cette deuxième section illustre aussi le fait que l'utilisation d'outils pour la mise en œuvre de contre-mesures, comme ici le compilateur, permet d'envisager des schémas de protection complexes qu'il serait autrement impossible de mettre en œuvre.

2.1 Compilation de contre-mesures contre les attaques par injection de fautes

Principaux éléments contextuels de cet axe de recherche

Collaborations	Karine Heydemann (LIP6), Bruno Robisson (CEA SAS), Laurent Maingault (CEA CESTI)
Publications	[BC16 ; BCR16 ; Bar+17 ; BCH17]
Projets liés	Thèse de Thierno Barry [Bar17], projet PROSECCO, projet CLAPs

Je me suis intéressé au potentiel du compilateur pour l'application de contre-mesures logicielles contre les attaques par injection de fautes, en premier lieu dans le cadre de la thèse de Thierno Barry [Bar17]. L'objectif de ces travaux portait sur l'application automatisée de schémas connus de contre-mesures, mais le fait d'exploiter le compilateur pour la sécurisation de code offre des opportunités, comme celle d'augmenter la flexibilité de configuration de la contre-mesure appliquée, et apporte des leviers pour améliorer le schéma de protection original.

Cependant, l'approche visant à exploiter le compilateur pour l'application de contre-mesures est complexe, ce qui est probablement une des raisons concrètes pour lesquelles, au démarrage de ces travaux, il existait dans l'état de l'art si peu de travaux sur la compilation de contre-mesures. Un compilateur est une infrastructure logicielle de grande envergure, qui manipule plusieurs représentations différentes du programme à compiler. Le programme à compiler est progressivement transformé vers sa forme finale au travers d'un grand nombre de procédures d'analyses et de transformations appelées *passes*. Habituellement, un compilateur s'intéresse uniquement aux propriétés fonctionnelles du programme compilé. En conséquence, la manipulation de propriétés non fonctionnelles, comme celles supportées par des contre-mesures, s'avère complexe parce que rien n'est prévu pour cela dans une infrastructure de compilateur traditionnelle. En outre, les interactions entre chaque passe de compilation peuvent donner lieu à des effets subtils et multiples qu'on ne sait pas bien analyser encore à l'heure actuelle. Ces questions, orthogonales, ne seront pas davantage discutées dans ce chapitre. Nous y reviendrons dans la section 4.2.

2.1.1 Compilation d'un schéma de tolérance aux sauts d'instruction

Dans un premier temps, nous avons automatisé l'application par le compilateur d'un schéma de tolérance au saut d'instruction [BCR16], le saut d'instructions étant un modèle de faute courant dans la littérature [Mor+13]. Ce travail vise la mise en œuvre, par le compilateur, d'un schéma de protection conçu, validé expérimentalement, et formellement vérifié par Moro et al. [Mor+14b]. On se place dans un modèle de sécurité où l'attaquant est capable de sauter une ou plusieurs instructions machine du programme cible, par l'injection d'une ou plusieurs fautes.

2.1.1.1 Principe de protection

Le schéma de protection exploite la propriété d'idempotence de certaines instructions machine : la duplication d'une instruction idempotente assure que l'état du programme est préservé

après l'exécution de tous les duplicatas, et ce, quel que soit le nombre de duplicatas exécutés. En faisant en sorte que le programme protégé contienne uniquement des instructions idempotentes, et que ces instructions soient dupliquées, on assure l'intégrité d'exécution du programme en présence d'injection de fautes permettant le saut d'instructions, à partir du moment où au moins un duplicata est exécuté correctement. Le schéma de protection original de Moro et al. procède par le remplacement systématique de chaque instruction machine non idempotente par une séquence fonctionnellement équivalente d'instructions idempotentes (les séquences les plus longues pouvant contenir jusqu'à 14 instructions). Chaque instruction est ensuite dupliquée, assurant ainsi la tolérance au saut d'instruction quel que soit l'endroit où la faute est injectée dans la section de code ainsi protégée. Dans les travaux de Moro et al., le programme à protéger est compilé, puis la contre-mesure est appliquée sur le code assembleur par un script. Le code machine final est ensuite produit par le compilateur.

2.1.1.2 Mise en œuvre

Étant donné que le schéma de protection s'applique aux instructions machine, notre mise en œuvre opère naturellement dans le back-end du compilateur. L'ajout de contraintes dans l'allocateur de registres et dans la sélection d'instructions permet de privilégier l'utilisation d'instructions machine naturellement idempotentes, donc de minimiser l'utilisation de séquences de remplacement plus coûteuses. Sur l'architecture ARMv7-M, on observe ainsi en pratique que plus de 90 % des instructions machine prennent une forme idempotente grâce à ces aménagements du back-end, ce qui permet de limiter le recours au remplacement par une séquence d'instructions idempotentes, donc de réduire le surcoût lié à l'application de la contre-mesure. Aussi, si la duplication des instructions idempotentes intervient avant l'ordonnancement des instructions, l'ordonnancement permet de minimiser le coût d'exécution des instructions ayant une latence de plusieurs cycles. Le tableau 2.1 illustre le gain apporté par l'application de la contre-mesure dans le compilateur en comparaison au schéma original de [Mor+14b] : le surcoût lié à l'application de la contre-mesure est plus faible, en temps d'exécution et en taille de code, quel que soit le niveau d'optimisation choisi par l'utilisateur. Aussi, le surcoût sur le temps d'exécution est ici inférieur à $\times 2$ (à l'exception de AES 8 bits compilé en `Os`) alors que chaque instruction du programme protégé est dupliquée une fois (protection contre une faute). Ceci s'explique par le fait que chaque instruction dupliquée n'est pas nécessairement exécutée deux fois (par exemple les branchements).

En outre, notre implémentation dans le compilateur permet d'ajouter des leviers de configuration supplémentaires permettant de supporter un modèle d'attaquant plus large, illustré en figure 2.1. Un nombre de duplicatas configurable (paramètre M) apporte une tolérance à $M - 1$ injections de fautes distinctes. Une passe de compilation supplémentaire, ajoutée après l'ordonnancement d'instructions, permet d'assurer une distance minimale entre chaque duplicata d'instruction afin d'assurer qu'une faute de largeur n (i.e., permettant de sauter n instructions contiguës) n'atteigne qu'un seul duplicata. Dans la thèse de Thierno Barry, plusieurs modèles d'espacement des duplicatas sont considérés [Bar17]. Globalement, cette mise en œuvre apporte à la fois une plus grande flexibilité d'utilisation et un surcoût moindre, même si le coût de la contre-mesure reste conséquent.

2.1.1.3 Évaluation de sécurité

Durant le projet PROSECCO, en collaboration avec le LIP6, nous avons formellement vérifié que la contre-mesure, appliquée par notre compilateur, protège complètement les programmes de test considérés, dans notre modèle de sécurité [Bré20 ; Bel+21]. Cette étape permet d'assurer que le compilateur a correctement mis en œuvre le schéma de protection.

TABLEAU 2.1 – Surcoût en taille code (en octets) et en temps d'exécution (en cycles d'horloge) observé pour l'application du schéma de tolérance aux sauts d'instruction. Les programmes sont compilés avec les niveaux d'optimisation de `O0` à `O3` et `Os`, pour un processeur ARM Cortex-M3, et la contre-mesure protège contre 1 faute. Les deux dernières colonnes reportent les mesures de surcoût présentées par MORO et al. [Mor+14b], pour le même processeur. Extrait de [Bar17].

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

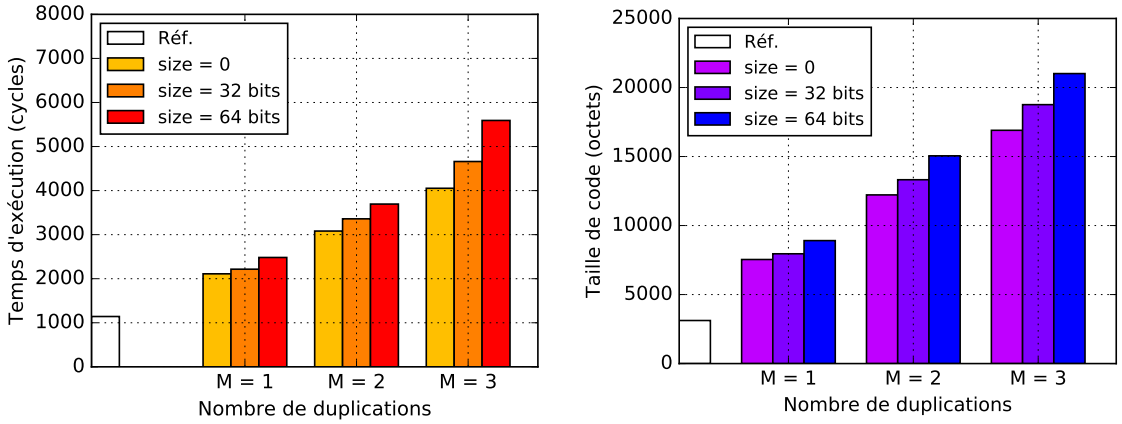


FIGURE 2.1 – Impact sur le temps d'exécution (gauche) et la taille du code (droite) d'une implémentation de AES protégée par `M` duplications, et pour une distance d'espacement entre chaque duplicata de 0, 32, ou 64 bits. Extrait de [Bar17].

Cependant, lorsqu'on se place dans un cadre expérimental plus large que le modèle de sécurité visé dans le contexte de nos travaux, on observe que la contre-mesure a un effet variable sur l'exploitation de l'injection de faute. Moro et al. montrent que la protection est variable contre des perturbations électromagnétiques, et qu'elle dépend notamment des paramètres d'injection [Mor+14a]. En collaboration avec le CESTI du CEA-Leti, nous avons mené une campagne d'évaluation expérimentale de la contre-mesure sous injection de fautes par laser, montrant que la contre-mesure est globalement inefficace contre l'injection laser. Ce résultat était attendu puisque les effets obtenus par injection laser sont beaucoup plus larges que le saut d'instruction. Une implémentation de AES et plusieurs implémentations de VerifyPIN [Dur+16], protégées avec des configurations différentes, ont été testées sur une carte STM32-F4 mise sur un banc d'injection laser. L'effet de la perturbation laser n'a pas été caractérisé sur la plateforme cible, et l'expérimentation ne cherchait pas à obtenir des sauts d'instruction. Les perturbations par injection laser peuvent créer des effets de natures très diverses sur le circuit, qui ne peuvent pas être couverts par la seule protection des sauts d'instruction. Dans le cadre de cette expérimentation, les effets obtenus par injection laser dépassaient donc largement le modèle d'attaquant prévu par la contre-mesure. Par exemple, l'implémentation AES, évaluée en DFA, est très sensible aux perturbations sur les données (e.g. par perturbation des valeurs de la matrice d'état interne utilisée pour le calcul AES), et ces fautes ne sont pas couvertes par le schéma de protection évalué. Qui plus est, l'application de la contre-mesure entraîne une augmentation de la surface d'attaque, et nous avons observé que le programme instrumenté avec la contre-mesure était dans certains cas plus simple à attaquer que le programme original.

Ces résultats corroborent d'autres travaux de l'état de l'art. Yuce et al. montrent qu'une faute en *clock glitch* permet de sauter deux duplicatas d'instruction consécutifs, s'ils sont simultanément en vol dans le pipeline du processeur [Yuc+16]. Cet effet est aussi observé par Cojocar, Papagiannopoulos et Timmers, qui montrent également que ce type de contre-mesure est susceptible d'entraîner une surface d'attaque plus importante face aux attaques par canal auxiliaire [CPT18].

2.1.1.4 Ouverture

Ces travaux sur la compilation d'un schéma de tolérance aux sauts d'instructions illustrent comment mettre en œuvre efficacement un schéma de protection à la compilation, comment généraliser le schéma de protection original et comment apporter une flexibilité dans son application, en rendant paramétrable le nombre de fautes tolérées et le nombre d'instructions que chaque faute est susceptible de sauter. Cependant, le surcoût induit par la protection reste important : la contre-mesure implique autant de duplicatas d'instructions que le nombre d'injections de fautes possibles, et la distance entre chaque duplicata doit être au moins égale au nombre d'instructions sautées par faute injectée. Le surcoût est d'autant plus important que la couverture de la contre-mesure est faible, puisque comme évoqué ci-dessus, en pratique une injection de fautes peut induire de nombreux autres effets que le saut d'instructions.

Il aurait été intéressant de faire l'évaluation de notre mise en œuvre de ce schéma de protection dans une configuration expérimentale permettant de cibler plus précisément le saut d'instructions. Par exemple, Menu et al. montrent la possibilité d'obtenir plusieurs sauts d'instruction consécutifs [Men+20]. L'approche de sécurisation présentée ici apporte une grande flexibilité de mise en œuvre qui permettrait de choisir des configurations de la contre-mesure qui répondent à une mise en œuvre de ce type d'attaque. Cependant, cette approche est viable seulement s'il est possible de borner le modèle d'attaquant (i.e., largeur de faute et nombre de fautes supportés), ce qui n'est souvent pas possible en pratique. En outre, Péneau et al. montrent que l'injection non bornée d'instructions *nop* est Turing-complète, c'est-à-dire qu'un attaquant capable d'in-

jecter un nombre illimité de sauts d'instruction a un potentiel de transformation illimité sur un programme cible [Pén+20].

Le principe de protection par copie d'instructions idempotentes s'avère en revanche une technique intéressante pour parer à des vulnérabilités ciblées, ou comme moyen de protection complémentaire à d'autres approches. Ce schéma est intéressant notamment pour la protection de branchements ou d'appels de fonction, puisqu'il n'implique pas de surcoût sur le temps d'exécution (un seul duplicata d'instruction étant exécuté). L'application ciblée par le compilateur, qu'il est possible de piloter par exemple par des annotations dans le code source ou des options de compilation dédiées, devient alors un outil puissant pour le concepteur.

2.1.2 Compilation d'un schéma de traceur

Également dans la thèse de Thierno Barry, nous nous sommes intéressés dans un deuxième temps à un autre mécanisme couramment utilisé dans la carte à puce, appelé *traceur*, qui vise à détecter les fautes entraînant une déviation du flot de contrôle. Ce mécanisme est similaire à certaines techniques logicielles d'intégrité du flot de contrôle (e.g. [OSM02]) protégeant contre un attaquant disposant seulement d'un accès logique à sa cible, sans la possibilité d'injecter des perturbations matérielles.

2.1.2.1 Principe de protection

Le principe de protection consiste à entrelacer le code à protéger avec des instructions de manipulation d'une variable de contrôle, aussi appelée *signature*. Par extension, on appelle *instructions de signature* la ou les instructions utilisées pour mettre à jour l'état de la signature courante. Au runtime, la valeur de signature est mise à jour régulièrement (typiquement : au moins une fois par bloc de base), de sorte qu'une déviation du flot de contrôle induite par l'injection d'une faute entraîne une valeur de signature incohérente. Suivant les mises en œuvre, la mise à jour de la valeur de signature peut se faire par logiciel, par des instructions dédiées (e.g. [OSM02]), ou par du support matériel dédié (e.g. [WWM15]). Le programme à protéger est également instrumenté avec des *points de vérification*, c'est à dire une séquence d'opérations comparant la valeur courante de la signature à la valeur attendue pour une exécution correcte du programme protégé. En l'absence de concordance, une alerte est levée.

La technique de traceur est régulièrement employée dans le domaine de la carte à puce, mais, à notre connaissance, c'est un spécialiste qui implante manuellement la contre-mesure, au cas par cas, dans le code source C ou assembleur. La mise en œuvre est *fragile*, puisqu'elle doit être adaptée dès que le code applicatif est modifié, en particulier en cas de changement du flot de contrôle. Aussi, la mise en œuvre dépend du nombre d'opérations de mise à jour de la signature, ce qui implique de mettre à jour les valeurs de références utilisées pour les vérifications. Notons enfin que l'efficacité de ce principe de protection dépend de nombreux facteurs, en premier lieu la capacité des opérations sur la signature à capturer une perturbation due à une injection de faute, la fréquence des mises à jour de la signature et des vérifications.

Nous avons repris le principe de traceur proposé par Lalande et al. [LHB14]. Dans leur mise en œuvre, la contre-mesure est appliquée dans le code source, au niveau C. La signature est un compteur entier, et la mise à jour incrémente ce compteur (`cnt++;`). Une mise à jour de signature courante est insérée entre chaque expression C originale du programme, et des expressions de vérification de la valeur courante de la signature sont insérées dans chaque bloc de base. Une des originalités de l'approche consiste à utiliser une variable de signature différente par bloc de base. Les instructions de vérification de la signature manipulée dans le bloc prédécesseur sont entrelacées avec les instructions de mise à jour de la signature courante pour assurer la

continuité de protection du flot de contrôle. La couverture de la contre-mesure est globalement bonne dans les évaluations menées dans [LHB14]. Cependant, la contre-mesure étant appliquée au niveau du code source, à la granularité d'expressions C, elle reste fragile aux transformations et aux optimisations appliquées par le compilateur jusqu'à la production du code machine, sur lesquelles il est difficile d'avoir un contrôle.

Le schéma original de Lalande et al. propose également une solution originale pour la protection des branchements conditionnels, qui vise à détecter les fautes correspondant à une inversion de branchement [LHB14]. Ce type de faute permet à l'attaquant de rester sur le flot de contrôle normal du programme, et en conséquence n'est habituellement pas traité par les solutions d'intégrité du flot de contrôle de l'état de l'art qui visent uniquement la protection des branchements indirects. La solution proposée crée une copie de la condition de branchement conditionnel, avant que le branchement soit pris. Lorsque la divergence sur les chemins d'exécution est résolue (e.g. en sortie d'une structure *if-then-else*, ou en sortie de boucle), la vérification choisit la variable de signature manipulée dans le chemin correspondant à la valeur de la condition de branchement. Ce mécanisme permet de détecter les inversions de branchement conditionnel, et les fautes sur la condition de branchement après sa copie par le mécanisme d'intégrité. Une des limites de cette approche est qu'elle met en œuvre des expressions C complexes, qui contiennent notamment des évaluations conditionnelles. Suivant le compilateur utilisé et l'architecture ciblée, ce type d'expression peut donner lieu à des constructions de code machine plus ou moins longues, pendant lesquelles la variable de signature n'est pas mise à jour, donc pendant lesquelles le mécanisme de protection lui-même est vulnérable aux fautes.

2.1.2.2 Mise en œuvre

La contre-mesure est appliquée au niveau du code machine, et ce de manière automatique par le compilateur [Bar17]. Le fait de pouvoir contrôler le code machine utilisé dans le système cible final assure un contrôle complet du code protégé au regard du modèle de sécurité. La mise en œuvre de la contre-mesure peut être automatiquement mise à jour si le code source est modifié, et n'est pas vulnérable aux transformations appliquées par le compilateur. Ici, le modèle de sécurité considère un attaquant capable de dévier le flot de contrôle original du programme à protéger, par injection de fautes. On souhaite également détecter les violations d'intégrité du flot de contrôle au sein d'un bloc de base, c'est-à-dire les fautes qu'il est possible de modéliser par l'introduction d'un saut au sein d'un bloc de base, ayant éventuellement pour destination une instruction du même bloc de base. Notre approche permet de détecter tous les sauts intra-bloc, à l'exception des sauts d'une seule instruction. Nous discutons plus loin comment il est possible d'étendre le modèle de sécurité à la détection de certaines perturbations sur le chargement des instructions.

La mise en œuvre, illustrée en figure 2.2, reprend le schéma original de Lalande et al. [LHB14]. Chaque bloc de base manipule une variable de signature différente, affectée à un registre processeur. Les instructions de vérification de la signature issue du bloc de base prédécesseur sont entrelacées avec les instructions de signature du bloc courant, afin de capturer aussi les fautes sur le flot de contrôle qui son injectées au moment de la vérification.

Les méthodes d'application de contre-mesures sur le flot de contrôle travaillent généralement en fin de middle-end du compilateur, e.g. [Pro+17 ; Ko+22], puisque la représentation intermédiaire du middle-end et son API sont les plus adaptées aux analyses et aux transformations du flot de contrôle. Cependant, nous avons fait le choix d'une mise en œuvre dans le back-end du compilateur, pour les trois raisons suivantes. i) Cela permet d'éviter que les passes de compilation pouvant impacter le flot de contrôle du programme cible n'altèrent la mise en œuvre de la contre-mesure. ii) Le schéma de protection exploite la mise en œuvre des évaluations de conditions spécifiques à l'architecture ARMv7 (section 2.1.2.3), et une partie au moins de l'im-

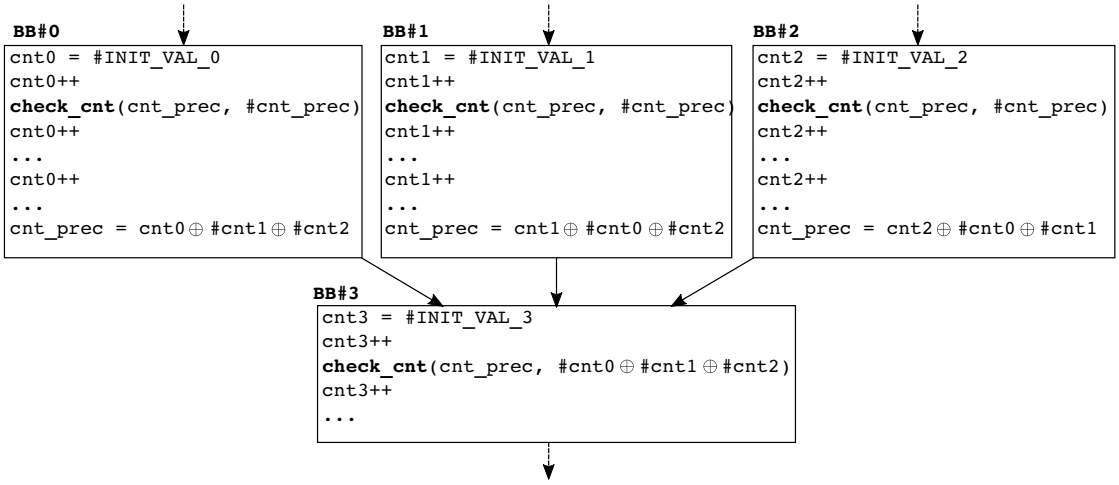


FIGURE 2.2 – Illustration de la mise en œuvre du schéma de traceur, exprimé en pseudo-assembleur. `#var` décrit la valeur de la variable `var`, qui est connue au moment de la compilation. Chaque ligne "`...`" correspond à une instruction originale du programme à protéger. Extrait de [Bar17].

plantation de la contre-mesure doit donc être calculée après le choix des instructions d'évaluation des conditions et des branchements conditionnels, dans la passe de sélection d'instructions du back-end. iii) Enfin, un placement judicieux des instructions de vérification permet de détecter certaines perturbations, notamment au chargement des instructions (section 2.1.2.4).

2.1.2.3 Extension à la protection des branchements conditionnels

Notre mise en œuvre de la protection des branchements conditionnels reprend les principes généraux de [LHB14] mais exploite les spécificités de notre ISA cible, ARMv7-M. Elle est illustrée en figure 2.3. Sur l'ISA ARMv7, les conditions de branchement sont déterminées à partir de la valeur de *drapeaux*, stockés dans 4 bits du registre de statut applicatif du processeur (APSR), appelés `N`, `Z`, `C`, et `V`. Ces drapeaux sont mis à jour par des instructions permettant l'évaluation de conditions de test (`cmp`) ou par la plupart des instruction de traitement de données. La valeur courante des drapeaux est copiée dès leur mise à jour, dans un champ de 4 bits sur les bits de poids fort du registre de signature. La vérification des conditions de branchement nécessite d'appliquer un masque sur la valeur des drapeaux, puisque les drapeaux exploités dépendent de la nature de la condition à évaluer, donc du type d'instruction de branchement conditionnel. La valeur de masque est stockée dans un autre champ de 4 bits du registre de signature, elle est statique, et déterminée à la compilation lorsque la nature de du branchement conditionnel est connue. Cette mise en œuvre permet d'insérer une instruction de signature entre chaque instruction de manipulation ou de vérification des conditions de branchement. Ainsi, la mise en œuvre du schéma de protection est plus robuste que la mise en œuvre originale de LALANDE et al. [LHB14], puisqu'elle assure l'insertion d'une instruction de signature entre chaque instruction originale du programme. L'attaque de ce schéma de protection suppose de cibler précisément une instruction machine en évitant de perturber les instructions immédiatement voisines, qui sont des instructions de signature.

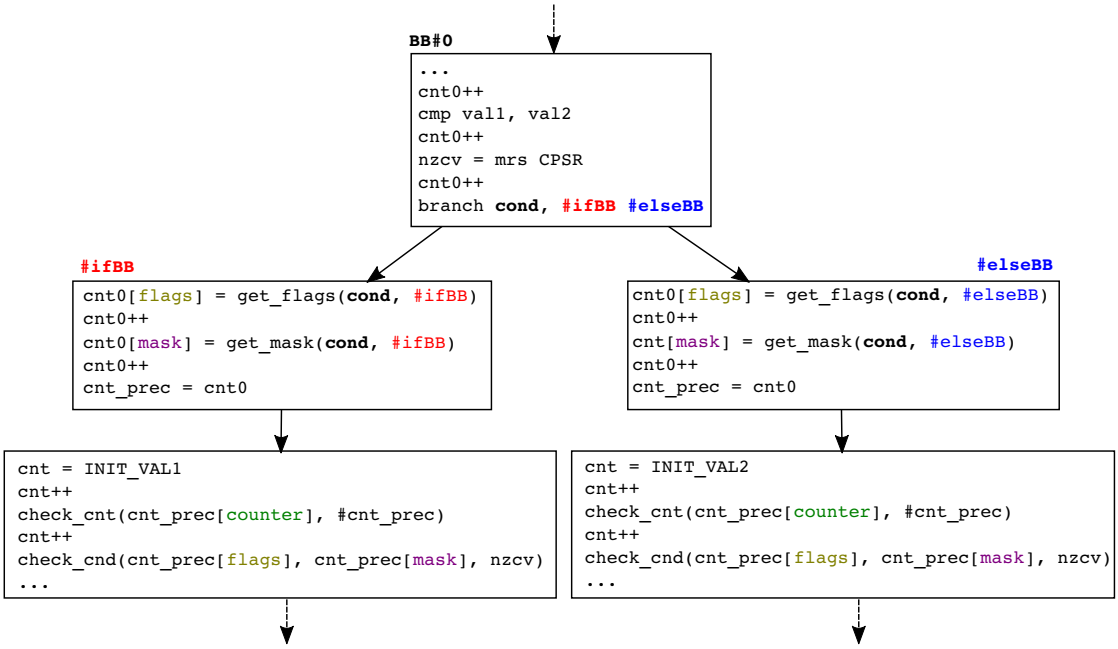


FIGURE 2.3 – Illustration de la mise en œuvre du schéma de traceur pour la protection d'un branchement conditionnel, exprimé en pseudo-assembleur. Extrait de [Bar17].

2.1.2.4 Capture de fautes au chargement des instructions

Plusieurs travaux de caractérisation montrent qu'une faute peut impacter plusieurs instructions machines qui sont voisines en mémoire programme. En particulier, dans le cas de microcontrôleurs exploitant un jeu d'instruction de taille variable comme l'ISA ARMv7-M, une perturbation du chargement d'un mot composé de plusieurs parties d'instructions [1] peut avoir un effet sur plusieurs instructions. MORO et al. étudient la sensibilité à l'injection électromagnétique d'une cible basée sur l'architecture ARMv7-M, équipée d'un microcontrôleur Cortex-M3 [Mor+14a]. Si les instructions machines sont encodées en Thumb-2, les instructions peuvent avoir une taille de 16 ou 32 bits. Sur cette architecture, les instructions sont chargées de la mémoire par mots de 32-bits, et l'étude de MORO et al. montre que le chargement des instructions est sensible aux perturbations électromagnétiques. Divers effets sont observés, en particulier des sauts d'instruction, ou le remplacement par une instruction de branchement. Aussi, en raison de la taille variable des instructions, il est possible que la faute perturbe deux instructions consécutives lorsqu'elles sont chargées simultanément. Notre mise en œuvre du mécanisme de traceur, sur l'architecture ARMv7-M, vise donc à exploiter cet effet observé expérimentalement sur ce type d'architecture : l'idée est de pouvoir capturer des perturbations sur un mot de 32 bits, en faisant en sorte que 16 bits de chaque mot de la mémoire programme soit occupé par une instruction de signature. De la sorte, l'instruction de signature joue aussi le rôle de *fusible* : on fait l'hypothèse qu'une perturbation du fetch aura tendance à impacter entre autres la partie du mot chargé qui encode l'instruction de signature. Le principe est illustré en figure 2.4. Les instructions de signature sont ici des incréments d'une variable de signature (`cnt++`), allouées à des instructions machine de 16

1. E.g., dans le cas de ARMv7-M, un mot de 32 bits composé de deux moitiés de 16 bits d'instructions de 32 bits non alignées, ou bien d'une instruction de 16 bits et de la moitié de l'instruction de 32 bits voisine.

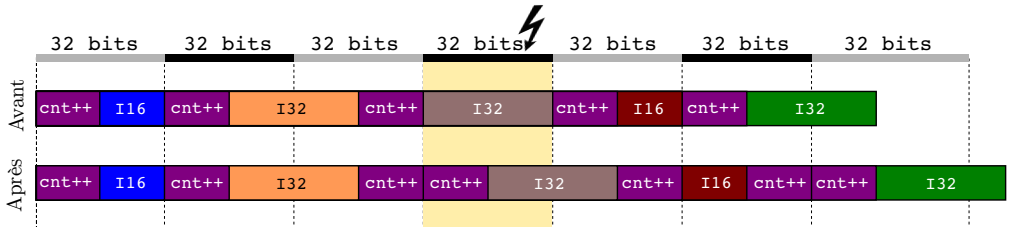


FIGURE 2.4 – Alignement des instructions dans des blocs de taille 32 bits. `I16` et `I32` représentent respectivement des instructions de taille 16 bits et 32 bits. Extrait de [Bar17].

bits. L'entrelacement des instructions de signature, sur 16 bits, et des instructions originales du programme, sur 16 ou 32 bits, permet d'obtenir cet effet de fusible. Cependant, une faute sur une instruction alignée de 32 bits ne serait pas capturée par les instructions de signature voisines (figure 2.4, ligne supérieure). Dans notre implémentation, on ajoute une passe de compilation dédiée qui assure que chaque mot mémoire de 32 bits contient une instruction de signature (figure 2.4, ligne inférieure), de manière à amplifier le potentiel de détection de perturbations par les instructions de signatures.

2.1.2.5 Évaluation

Le tableau 2.2 illustre le coût de la contre-mesure lorsqu'elle est appliquée de manière systématique à l'ensemble de la fonction ciblée. Le coût est important, notamment en raison de l'insertion d'au moins une instruction de signature par instruction originale du programme (suivant les contraintes d'alignement des instructions, il est possible que deux instructions successives de signature soient utilisées, comme illustré dans la figure 2.4).

2.1.2.6 Ouverture

La mise en œuvre de ce schéma de traceur dans le back-end du compilateur est complexe, puisqu'elle suppose de pouvoir faire des analyses sur le flot de contrôle et de modifier le flot de contrôle original du programme, par exemple par l'insertion de nouveaux blocs de base dédiés au fonctionnement de la contre-mesure. Les passes travaillant sur le flot de contrôle sont d'ordinaire mises en œuvre dans le middle-end du compilateur, mais certaines passes du back-end peuvent entraîner une dégradation de la contre-mesure [Pro+17 ; HLB19]. Nous avons fait le choix d'une implémentation dans le back-end pour avoir de meilleures garanties sur la préservation du schéma de protection, en plaçant la passe d'application de la contre-mesure le plus tard possible, après les autres passes du compilateur pouvant modifier l'implantation de la contre-mesure dans le code machine final. A contrario, une mise en œuvre sur une représentation plus abstraite, par exemple l'IR du middle-end, aurait simplifié quelques aspects de mise en œuvre comme la gestion des variables de signatures, qui sont actuellement affectées à des registres réservés. Dans le cadre de l'application d'une contre-mesure de sécurisation des boucles, PROY et al. font une analyse méthodique des passes postérieures à l'application de la contre-mesure qui peuvent nuire au schéma de protection, et de leurs effets [Pro+17]. Le choix d'un niveau d'abstraction et d'une représentation adaptés, et du placement de la passe dans le flot de compilation sont des problématiques courantes lors de la conception d'un compilateur. Cependant, dans le cadre de la sécurisation de programmes, cette problématique est exacerbée puisqu'une infrastructure conventionnelle de compilateur n'apporte pas de support pour la préservation de propriétés non fonctionnelles.

Tableau 2.2 – Surcoût en temps d'exécution (en cycles d'horloge) et en taille code (en octets) observé pour l'application du schéma CCFI, compilé avec les niveaux d'optimisation de `O0` à `O3` et `Os`, pour un processeur ARM Cortex-M3. Extrait de [Bar17].

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

Étant donné que les instructions de vérification représentent des points de vulnérabilité du programme protégé, Thierno Barry a montré comment le schéma de tolérance présenté ci-dessus (section 2.1.1, page 6) peut s'appliquer aux instructions de vérification seulement (i.e. duplication des instructions de vérification), pour durcir le schéma de protection à moindre coût [Bar17, Chapitre 7]. De manière similaire, dans un schéma de protection des boucles, Proy et al. dupliquent certains branchements inconditionnels qui ne sont pas protégés parce qu'ajoutés après l'application de la contre-mesure dans le middle-end du compilateur [Pro+17, section 4.2.1].

D'une manière générale, un environnement de sécurisation de programmes devrait donc permettre d'appliquer des contre-mesures i) exhaustivement sur une portion complète de code ; ii) sélectivement sur quelques motifs ou instructions, par exemple parce qu'elles ne sont pas protégées par d'autres contre-mesures.

2.2 Compilation de contre-mesures contre les attaques par canal auxiliaire

Principaux éléments contextuels de cet axe de recherche

Collaborations	Bruno Robisson (CEA SAS), Henri-Pierre Charles (CEA List), Karine Heydemann (LIP6), Nicolas Belleville (CEA List), Loïc Masure et Eleonora Cagli et Cécile Dumas (CEA CESTI), Hélène Le Bouder et Jean-Louis Lanet (INRIA LHS Rennes), Philippe Jaillon et Olivier Potin (EMSE)
Publications	[Cou+16a ; Le +16a ; Bel+18b ; Bel+19 ; Bel+20a ; Mas+20 ; BC21]
Projets liés	Thèse de Nicolas Belleville [Bel19], projets COGITO, PROSECCO, CLAPs

Deux grands principes de protection sont utilisés contre les attaques par canal auxiliaire. Le *masquage* sépare chaque variable secrète en plusieurs parties statistiquement indépendantes, appelées *shares*, de sorte que seule la connaissance de l'ensemble des shares permet d'accéder à la valeur secrète ainsi encodée. Dans une attaque par canal auxiliaire, la valeur des shares est inférée des observations physiques faites sur le circuit en fonctionnement. L'application du masquage implique de ré-exprimer les calculs sur les données secrètes dans un autre espace arithmétique correspondant à la manière dont les shares sont calculés à partir du secret. La *dissimulation* est un principe de protection qui couvre un grand nombre de contre-mesures matérielles et logicielles avec des mécanismes variés. Il s'agit de diminuer le rapport signal à bruit pour un attaquant, ou d'augmenter la surface d'observation nécessaire à un attaquant (e.g. taille de la fenêtre temporelle d'observation), impliquant une augmentation du coût de la mise en œuvre ou de l'exploitation d'une attaque. Contrairement au masquage, les techniques de dissimulation n'impliquent pas nécessairement de repenser l'implémentation du programme à sécuriser. Ainsi, les techniques de dissimulation matérielles (e.g. gigue sur l'horloge [CDP17], logique en double rail [TAV02 ; MSS09]) sont complètement transparentes du point de vue logiciel. Les techniques logicielles de dissimilation peuvent impliquer des transformations sur le programme protégé, mais sans porter atteinte à ses propriétés fonctionnelles. En pratique, un niveau de protection suffisant est obtenu par la combinaison de techniques de masquage et de dissimulation.

Sur cet axe, nous avons travaillé sur l'application à la compilation de deux contre-mesures : le polymorphisme de code comme technique de dissimulation (section 2.2.1), et le masquage Booléen (section 2.2.2).

2.2.1 Dissimulation par polymorphisme de code

Avant le démarrage des travaux relatés dans ce mémoire, mes travaux de recherche portaient sur l'optimisation logicielle exploitant la connaissance des données au runtime, par génération dynamique de code, en collaboration avec Henri-Pierre Charles. Nous travaillions alors sur une technologie de génération dynamique de code appelée *deGoal*, qui se démarquait des JITs traditionnels par une empreinte mémoire réduite et un faible coût de génération de code [CC12 ; LC12 ; AC13 ; Cha+14a ; Cha+14b ; CQY15a ; CQY15b ; QCC15]. L'idée est d'utiliser des générateurs de code au runtime, appelés *compilettes*, spécialisés en fonction du code à générer, donc plus rapides et moins volumineux qu'un moteur JIT traditionnel. Au runtime, une compilette produit le code machine de la fonction cible, et la génération de code est pilotée par la connaissance des données au runtime, ce qui permet de mettre en œuvre diverses techniques d'optimisation sur les codes générés, par exemple pour réduire le temps d'exécution ou la consommation énergétique [End15].

Je me suis aperçu que la variabilité comportementale apportée par cette technique de génération de code pouvait avoir un intérêt contre les attaques par canal auxiliaire. L'idée centrale de cette idée, appelée *polymorphisme de code*, est la suivante : plutôt que de piloter la génération de code par la connaissance de données au runtime, la génération de code est pilotée par un aléa. Ce principe a fait l'objet d'un brevet préliminaire [Cou13]. J'ai constitué un consortium autour du projet ANR COGITO, rassemblant les compétences nécessaires pour évaluer l'intérêt de cette idée dans le contexte de la sécurité matérielle, qui était alors un nouveau domaine de recherche pour moi. Nous avons démontré le potentiel du polymorphisme de code dans une mise en œuvre dérivée de deGoal, à la fois du point de vue sécurité, et du point de vue de l'applicabilité de la solution technique à des systèmes embarqués contraints en ressources de calcul. Les travaux sur cette contre-mesure se sont poursuivis dans les activités de recherche suivant le projet COGITO, pour renforcer le potentiel de protection et le potentiel d'applicabilité de cette solution.

Lors du démarrage de ces travaux, les attaques par canal auxiliaire sont déjà bien connues, leurs principes théoriques et pratiques sont activement étudiés dans une communauté de re-

cherche autour de la sécurité matérielle. Les principes de sécurisation des implémentations logicielles sont déjà avancés : masquage à l'ordre supérieur [RP10], introduction de nouveaux schémas de masquage dans des espaces arithmétiques différents du masquage Booléen habituellement utilisé (e.g. masquage affine [Fum+11]), dissimulation par logiciel pour disperser dans le temps la fuite d'information [CK09 ; CK10], etc. En pratique, on sait qu'il est nécessaire de combiner des techniques de dissimulation et de masquage pour obtenir un haut niveau de sécurité (e.g. [RPD09]) : le masquage permet de supprimer la fuite d'information jusqu'à un certain ordre, mais nécessite d'être combiné à des techniques de dissimulation pour amplifier la complexité calculatoire des attaques d'ordre supérieur, en particulier des attaques multi-variées. À titre d'exemple, on peut citer Veyrat-Charvillon et al. [Vey+12] : « *This suggests the design of fully shuffled (and efficient) implementations, where both the execution order of the instructions and the physical resources used are randomized, as an interesting scope for further research.* ».

Toutefois, à cette même époque, la littérature s'intéresse rarement aux applications outillées de ces contre-mesures. La mise en œuvre des contre-mesures est toujours *ad hoc*, appliquée seulement à quelques primitives cryptographiques, majoritairement l'AES-128. Cela est vrai aussi pour les travaux sur la dissimulation, alors que, contrairement au masquage, la dissimulation est une contre-mesure qui peut s'appliquer indépendamment de la nature algorithmique de la fonction à protéger. Une des questions ouvertes par ces travaux était donc, comme pour les contre-mesures contre les fautes présentées précédemment, de savoir quel était le potentiel apporté par le compilateur pour une application automatisée, ou au moins outillée.

Concernant les techniques logicielles de dissimulation, la littérature s'intéresse alors essentiellement aux techniques de *shuffling*, qui consistent à exécuter dans le désordre des calculs indépendants, comme les calculs AddRoundKey ou SubBytes sur tous les éléments de la matrice interne AES [RPD09]. Ces techniques peuvent être amplifiées par l'introduction de calculs sur des données factices (e.g. *dummy rounds*) [HOM06 ; MOP07] ou par l'insertion de routines de désynchronisation dont le temps d'exécution est variable [CK10]. Cependant, dans ces approches, le code utilisé pour la dissimulation est distinguable du code original à protéger, et en conséquence il est possible, par des techniques de pré-traitement des traces, d'annuler tout ou partie de l'effet de la dissimulation [vWB11 ; SP12 ; Vey+12 ; Dur+13]. On constate donc que, d'une part, les techniques de dissimulation sont des candidates à l'application automatisée puisque le schéma de protection peut s'appliquer indépendamment des aspects fonctionnels du programme à protéger (e.g., l'exécution d'instructions factices [CK10] peut se faire quelque soit le code instrumenté). D'autre part, on constate que les techniques de dissimulation doivent être renforcées pour être efficaces. Si la mise en œuvre de la contre-mesure est plus sophistiquée, son application sera facilitée si elle est supportée par un outil.

2.2.1.1 Preuve de concept par DSL

Dans la première phase de ces travaux, réalisés en grande partie dans le projet ANR COGITO, je me suis tourné vers le développement d'un outil de prototypage permettant d'expérimenter plusieurs solutions de protection, à partir d'une solution dérivée de la technologie deGoal. La solution développée permet de mettre en œuvre plusieurs mécanismes de transformation de code intégrés dans ce que nous avons appelé le *polymorphisme de code*. L'idée est de mettre en œuvre une grande variabilité comportementale sans porter atteinte aux propriétés fonctionnelles du programme à protéger. La variabilité comportementale est obtenue par régénération au runtime du code à protéger. Un générateur de code, appelé *SGPC* (*Specialized Generator of Polymorphic Code*), est associé à chaque composant logiciel à protéger (e.g. une fonction), et chaque exécution du SGPC produit un nouveau code machine appelé *instance polymorphe*. C'est l'exécution suc-

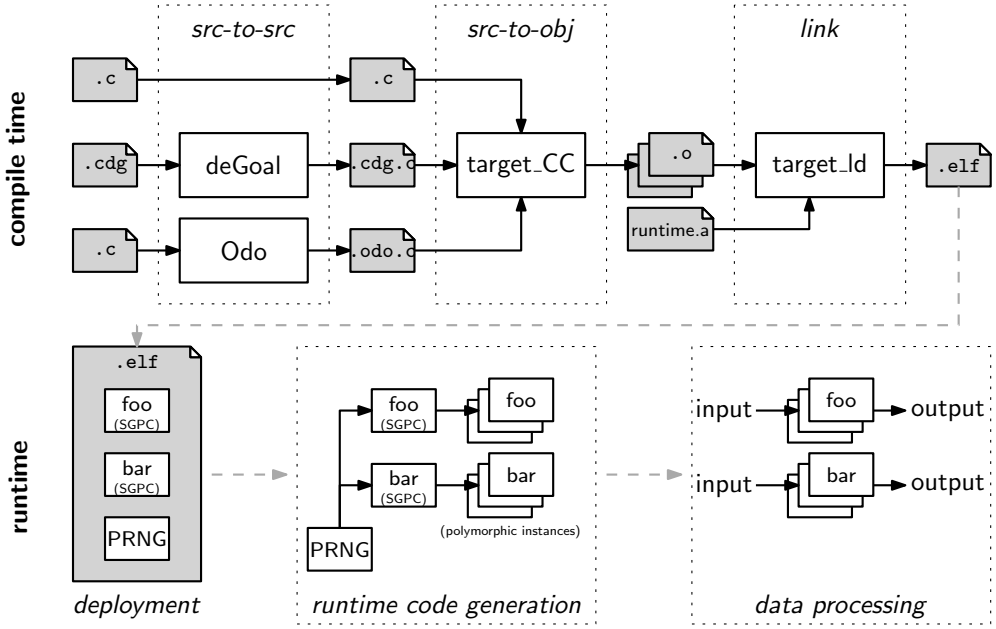


FIGURE 2.5 – Chaîne de compilation pour l'application de la contre-mesure de polymorphisme de code

cessive de plusieurs instances polymorphes qui permet d'obtenir une variabilité comportementale visant à augmenter le coût de réalisation d'une attaque par observation. Il faut noter que dans ce contexte, contrairement à la mise en œuvre originale de deGoal [Cha+14b], la génération de code au runtime reste agnostique des données d'entrée de l'instance polymorphe : c'est une propriété importante pour une application à la sécurité, de sorte que l'attaquant ne puisse pas obtenir d'informations directes sur des secrets par l'observation de l'activité de génération de code.

La figure 2.5 illustre la chaîne de compilation mise en œuvre, à la fois pour cette première phase de développement puis sur la phase suivante basée sur un compilateur dédié (section 2.2.1.2). La première preuve de concept mise en œuvre au début de ces travaux exploite un DSL qui permet d'exprimer simplement les parties du code qui feront l'objet d'une génération de code au runtime [Cou+14 ; Cou+16a]. Le DSL, illustré en figure 2.6, peut être combiné avec du code fonctionnel écrit en C, ce qui permet de prototyper facilement des techniques hybrides intégrant de la génération de code au runtime. Les composants logiciels implémentés dans le DSL (fichiers `.cdg`, figure 2.5) sont ensuite générés automatiquement (par l'outil `deGoal`, figure 2.5) sous la forme de SGPC implémentés en C (fichiers `.cdg.c`, figure 2.5), puis compilés avec la chaîne de compilation habituelle du système cible (`target_CC`, figure 2.5). Un léger runtime est également inséré dans le code applicatif cible lors de l'édition des liens : ce runtime fournit le support pour la génération de code machine ainsi que quelques utilitaires système. Seules les fonctions de génération de code utiles aux fonctions ciblées seront incluses dans le binaire final, ce qui permet de réduire l'empreinte mémoire de l'applicatif final en comparaison aux solutions conventionnelles de JIT.

Une fois le code déployé sur le système cible, chaque SGPC produit, à chaque exécution, une nouvelle version du code machine de la fonction cible (*polymorphic instances*, figure 2.5). La figure 2.5 illustre aussi le fait que les données utilisées par le calcul dans la fonction cible ne sont pas manipulées par les SGPC, lors de la phase de génération de code au runtime.

```
void gen_subBytes (cdg_insn_t* code,
                   uint8_t* sbox_addr, uint8_t* state_addr)
{
#[
  Begin code Prelude
  Type uint32 int 32
  Alloc uint32 state, sbox, i, x, y
  mv state, #(state_addr)
  mv sbox, #(sbox_addr)
  mv i, #(0)
  loop:
    lb x, @(state+i)  //  x := state[i]
    lb y, @(sbox+x)   //  y := sbox[x]
    sb @(state+i), y  //  state[i] := y
    add i, i, #(1)
    bneq loop, i, #(16)
  rtn
  End
]#;
}
```

FIGURE 2.6 – Exemple d'implementation de SGPC pour la fonction SubBytes de AES. Le DSL est utilisé dans les sections `#[...]#`. Des expressions C sont incluses dans le DSL à l'aide de `#(...)`. Extrait de [Cou+16a]

Le prototype de la première chaîne de compilation, basée sur un DSL, réalisé dans le projet COGITO, supporte les quatre transformations de code au runtime suivantes : [Cou+14 ; Cou+16a] allocation aléatoire de registres (*random register allocation*), *mélange des instructions* (*instruction shuffling*), *sélection aléatoires de variants sémantiques* pour une opération (*semantic variants*), et *insertion d'instructions factices* (*noise instructions*). Chaque transformation de code supporte quelques paramètres et est configurable lors de la phase de compilation statique. Ces paramètres peuvent impacter à la fois la variabilité comportementale, le temps d'exécution des SGPC et des instances polymorphes, et l'empreinte mémoire du code. Cette flexibilité de configuration offre donc une grande flexibilité dans le choix du compromis coût-sécurité. De plus, étant donné que la configuration de la contre-mesure se fait à la compilation, la flexibilité apportée par l'approche n'induit pas de surcoût au runtime. Enfin, il est intéressant de noter que l'insertion d'instructions factices manipule des instructions de même nature que celles utilisées dans le programme original (typiquement pour une fonction AES : accès mémoire, opérations arithmétiques et opérations bit-à-bit), afin d'augmenter la difficulté des techniques de pré-traitement par filtrage ou compression de motifs comme [SP12] ou [vWB11].

Notre prototype cible l'architecture ARMv7-M, et l'implémentation supporte les systèmes embarqués contraints en ressources de calcul et en mémoire, qui ont des caractéristiques similaires à un système de type carte à puce. À titre d'illustration, nous montrons la possibilité de protéger une implémentation complète de la fonction de chiffrement AES sur un micro-contrôleur pourvu de seulement 8 ko de RAM [Cou+16a]. Ce type de système cible est habituellement inexploitable par les techniques de génération dynamique de code de l'état de l'art, car trop contraint.

À l'issue du projet COGITO, outre les résultats des évaluations de sécurité qui sont présentées ci-dessous (page 22), plusieurs conclusions sont tirées, dont je synthétise les principales ici. Certaines sont issues de discussions avec des industriels de la carte à puce ou de l'informatique

embarquée à l'issue de présentations de nos résultats préliminaires.

— La mise en œuvre du polymorphisme de code au travers d'un DSL impose un effort d'ingénierie supplémentaire et conséquent, pour la réécriture de composants dont l'implémentation est parfois figée depuis longtemps. C'est un verrou pratique important, qui a orienté la suite de nos travaux autour du polymorphisme de code.

— Ensuite, l'utilisation de la génération de code au runtime dans les systèmes embarqués est inhabituelle, voire considérée contraire aux bonnes pratiques, puisqu'elle suppose de laisser un accès en écriture à une partie de la mémoire programme. Ce point est source de vulnérabilités potentielles et a également été adressé dans la suite de nos travaux.

2.2.1.2 Généralisation du principe de polymorphisme de code et application à la compilation

Dans le cadre de la thèse de Nicolas Belleville [Bel19], nous réalisons une automatisation complète de l'application du polymorphisme de code à la compilation, appelée *Odo* [Bel+19]. L'implémentation s'appuie sur la preuve de concept par DSL réalisée auparavant, mais cette fois le polymorphisme de code est mis en œuvre à partir de code C standard, provenant par exemple d'une implémentation originale, dépourvue de contre-mesures, du programme à protéger. On se libère donc de la contrainte de ré-implémenter les composants logiciels à protéger.

Dans cette nouvelle mise en œuvre, c'est le compilateur Odo qui produit le code C des SGPC pour chaque fonction dans le code source à protéger par polymorphisme de code (figure 2.5). Le reste de la chaîne logicielle est similaire à l'approche prototype présentée ci-dessus. La chaîne de compilation permet, à l'aide d'options dédiées, de configurer les mécanismes de transformation de code appliqués au runtime et d'appliquer sélectivement la contre-mesure sur des portions de code cible. Le compilateur Odo est basé sur LLVM, et utilise une approche originale pour la génération des SGPC. La compilation du programme ciblé suit le flot habituel, jusqu'au back-end où seule la passe d'émission de code est modifiée : cette passe émet du code C au lieu de code machine ou de code assembleur comme c'est habituellement le cas. Pour chaque instruction machine, la passe d'émission de code cible des procédures de génération de code machine fournies par une bibliothèque dédiée (`runtime.a`, figure 2.5). Suivant la configuration choisie par l'utilisateur, le compilateur Odo peut également insérer du code exploité lors de la génération de code au runtime. Si le polymorphisme de code n'est pas appliqué, le code original est simplement reproduit à l'identique ; il sera compilé dans la phase suivante par le compilateur de la plateforme cible (`target_CC`, figure 2.5). Le code généré est également instrumenté pour l'application de la contre-mesure de polymorphisme de code, et l'instrumentation dépend de la configuration de la contre-mesure choisie à la compilation par l'utilisateur. Une description plus complète de ce procédé est détaillée dans [Bel+19] et [MCH21].

En outre, cette nouvelle mise en œuvre apporte de nombreuses améliorations, les plus importantes étant synthétisées ci-dessous.

Calcul automatisé de la taille des buffers recevant le code des instances polymorphes. La taille d'une instance polymorphe dépend de nombreux facteurs : en premier lieu la taille du code original, mais aussi la nature des transformations de code utilisées au runtime et leur configuration. D'une génération de code à l'autre, cette taille peut grandement varier, et l'estimation d'une taille de buffer adéquate est donc difficile si elle doit être faite manuellement. Lors de la génération du code du SGPC, le compilateur calcule automatiquement la taille du buffer recevant le code des instances polymorphes. Étant donné la forte variabilité du code des instances polymorphes, il n'est pas possible d'allouer la taille nécessaire correspondant à l'instance polymorphe la plus grande possible. Le compilateur détermine donc une taille de buffer de sorte qu'il existe

une probabilité p de débordement, choisie par l'utilisateur. Le compilateur met en œuvre un mécanisme de prévention du débordement du buffer alloué : au runtime, le SGPC peut modifier la politique de transformations de code (notamment pour l'insertion d'instructions factices) afin d'éviter le débordement. S'il s'avère que le débordement est inévitable malgré tout, la génération de code reprend depuis le début. Le choix de la valeur de la probabilité de débordement p est un compromis, qui influe sur la taille du buffer alloué (plus p est faible, plus le buffer alloué sera grand), et sur l'importance du biais introduit dans les lois de probabilités exploitées dans les modèles de transformations de code. On choisit la valeur par défaut $p = 10^{-6}$, ce qui correspond à des tailles de buffers raisonnables et à un biais dans les lois de probabilités qui nous semble inexploitable en pratique.

Gestion des permissions mémoire. La génération dynamique de code nécessite un accès en écriture à la mémoire programme, ce qui peut introduire des vulnérabilités potentielles (par exemple, simplifier l'exploitation d'un buffer overflow). Pour éviter cela, le SGPC modifie les droits d'accès au buffer d'instructions, qui basculent exclusivement entre accès en lecture (par défaut) et accès en écriture (seulement pendant le temps de la génération de code). Ainsi, seul un attaquant capable de détourner le flot d'exécution du SGPC (i.e., entre l'activation de la permission en écriture seule en début de génération de code et la restauration de la permission en lecture seule en fin de génération de code) est capable d'exploiter l'accès en écriture, et ce seulement pour la section mémoire à laquelle le SGPC a accès. La mise en œuvre s'appuie sur le module PMP (*Physical Memory Protection*) disponible sur le système utilisé dans nos évaluations (section 2.2.1.3), mais peut se transposer à d'autres mécanismes de partitionnement ou d'isolation mémoire.

Bruit dynamique. Dans la mise en œuvre initiale, la variabilité comportementale, susceptible de durcir une implémentation face à un attaquant par canal auxiliaire, est obtenue seulement par la génération successive d'instances polymorphes. La fréquence de re-génération de code devient un facteur de sécurité, et impacte aussi le surcoût global en temps d'exécution. Par ailleurs, la génération dynamique de code implique un surcoût en temps d'exécution qui reste non négligeable [Cou+16a], malgré les très bonnes performances de la solution de génération de code sur laquelle nous nous appuyons, deGoal.

Dans le but d'apporter de la variabilité comportementale indépendamment de la re-génération de code, nous proposons une technique appelée *bruit dynamique* [Bel+19, Fig. 2]. La technique consiste à introduire des sections courtes d'instructions factices, chaque section étant précédée d'un branchement ciblant aléatoirement une des instructions dans la plage d'instructions factices, de sorte que le nombre d'instructions exécutées varie aléatoirement à chaque exécution de cette section d'instructions. Les plages d'instructions de bruit dynamique sont insérées pendant la transformation polymorphe d'insertion d'instructions factices (*noise instructions*).

Analyse de vivacité des registres. L'implémentation prototype initiale du polymorphisme de code (section 2.2.1.1) exploitait un algorithme glouton pour l'allocation de registres pendant la génération de code. Pour chaque nouveau registre, l'allocateur choisit aléatoirement parmi les registres disponibles. Cet allocateur permet de produire du code de qualité acceptable en minimisant l'impact du temps de calcul de l'allocation de registres pendant l'exécution des SGPC.

Nous avons ici utilisé une autre approche, permettant de réduire le coût au runtime de la sélection aléatoire de registres. L'idée est d'instrumenter le code des SGPC, pendant la phase de compilation statique, pour préparer la sélection aléatoire de registres pendant la génération de code dynamique. La phase de compilation statique (`src-to-src`, figure 2.5) produit des

informations de vivacité des registres. Ces informations sont exploitées pour l'insertion d'instructions supplémentaires (insertion d'instructions factices et utilisation de variants sémantiques), qui peuvent ainsi accéder aux registres disponibles sans passer par un calcul d'allocation au runtime. L'aléa sur les registres vivants est introduit par une permutation sur une partie des registres disponibles. On réduit grandement la variabilité dans les registres exploités par chaque instance polymorphe, en contrepartie d'un gain sur le temps de génération de code au runtime. Cette technique réduit aussi la variabilité sur le temps d'exécution des instances polymorphes, puisque le code de sauvegarde et de restauration du contexte d'exécution, qui dépend des registres alloués, est constant pour toutes les instances polymorphes d'une fonction cible.

2.2.1.3 Évaluation expérimentale

Je présente ici quelques résultats d'évaluation, en particulier en collaboration avec des équipes spécialisées à la pointe sur les attaques par canaux auxiliaires, puis une synthèse de l'impact du polymorphisme de code sur la performance.

Les conditions expérimentales ont été détaillées dans les publications présentant ces travaux. On peut simplement noter ici que le polymorphisme de code est évalué sur des implémentations logicielles de primitives de sécurité (e.g., AES) dépourvues d'autres contre-mesures, sur des systèmes embarqués de la famille STM32 de STMicroelectronics. Nous avons choisi, dans le porfolio STM32, des modèles contraints comme ceux de la série F1, et dépourvus d'autres contre-mesures matérielles ou logicielles.

Évaluation en sécurité : attaques verticales. Le polymorphisme de code apporte une forme de dissimulation si la génération de nouvelles instances polymorphes intervient lorsqu'un attaquant mesure l'activité du système ciblé. En premier ordre, le polymorphisme de code apporte une forme de désynchronisation temporelle, un moyen efficace d'augmenter la difficulté de réaliser des attaques verticales simples. En effet, la plupart des transformations de code ont pour effet de faire varier le temps d'exécution des instances polymorphes, et en particulier de faire varier le délai entre un point de synchronisation potentiellement exploité par un attaquant (e.g., le démarrage de l'exécution du programme attaqué) et l'instant où l'activité du processeur produit une information exploitable par l'attaquant. Cet effet est illustré en figure 2.7, où on effectue une CPA sur un AES polymorphe dont le code de l'instance polymorphe est renouvelé après 200 exécutions. On observe une chute importante de la valeur de corrélation à partir du moment où, dans la phase d'analyse de l'attaque, les traces d'exécution de la deuxième instance polymorphe sont mélangées à celles de la première. Dans l'exemple illustré en figure 2.7, l'attaque reste triviale puisque l'intervalle de régénération de code est plus grand que le nombre de traces (donc d'exécutions) nécessaires à l'extraction de la valeur de la clé secrète. La fréquence de génération des instances polymorphes est donc un facteur important dans le niveau de sécurité apporté face à des attaques verticales simples [Bel+19, Figure 6].

L'effet du polymorphisme de code ne se limite pas à de la désynchronisation temporelle : le fait de changer régulièrement de registres, d'opérateurs, de séquences d'instructions, etc., a pour effet d'apporter une grande variabilité comportementale, ce qui augmente significativement le coût de réalisation des attaques. En combinant les transformations de code et en choisissant judicieusement leur configuration, il est possible d'obtenir des résultats intéressants. L'évaluation en sécurité de Odo publiée dans [Bel+19] exploite plusieurs configurations arbitrairement choisies (*low*, *mid*, *high*) apportant des degrés de résistance variables à des attaques verticales. La figure 2.8 illustre le taux de succès d'une attaque verticale, par CPA, contre l'implémentation AES la plus faiblement protégée : le nombre de traces nécessaires pour l'exploitation de l'attaque augmente d'un facteur 13 000 par rapport à une attaque sur le même AES non protégé. L'ap-

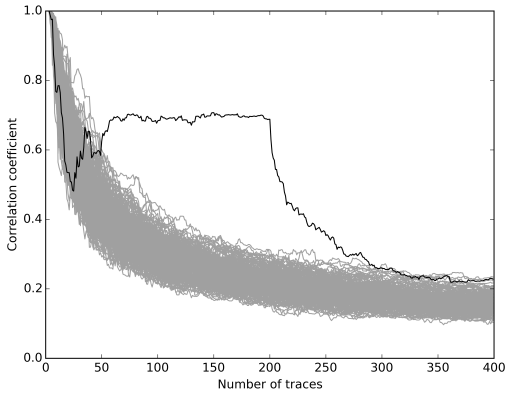


FIGURE 2.7 – CPA sur un AES polymorphe pour lequel l'instance polymorphe est régénérée toutes les 200 exécutions. Extrait de [Cou+16b].

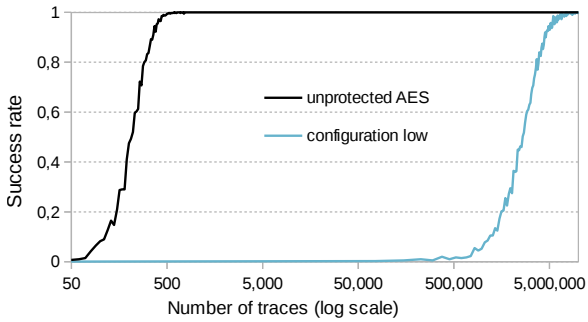


FIGURE 2.8 – Taux de succès d'une CPA sur une implémentation de AES sans protection (*unprotected AES*) et avec du polymorphisme de code dans une configuration faible (*configuration low*). Extrait de [Bel+19].

proche est très configurable et supporte donc un grand nombre de combinaisons possibles dans la mise en œuvre du polymorphisme de code, chaque combinaison apportant un compromis surcoût-protection différent. La figure 2.9 illustre cette variabilité, où chaque configuration présente un niveau de fuite d'information différente dans une évaluation de t-test. Dans les configurations les plus poussées, la fuite d'information n'est plus détectée par le t-test. Par expérience, nous avons aussi observé que l'effet de chaque transformation de code sur une attaque varie suivant la nature du système cible : son architecture, la technologie utilisée pour la réalisation du circuit, etc. En pratique, l'application du polymorphisme de code, comme de toute autre contre-mesure, suppose donc une caractérisation du système cible et une définition des capacités d'un attaquant.

Dans le cadre d'une collaboration avec une équipe du LHS à l'INRIA, nous montrons que les attaques par *template* sont efficaces contre des implémentations de la fonction d'authentification verifyPin [Le +16a]. Il s'agit d'un cas rare de l'usage des attaques par canal auxiliaire en dehors de la cryptographie pour l'extraction de données secrètes. L'originalité de cette étude réside dans le nombre limité d'exécutions sur cible qu'un attaquant peut exploiter pour extraire un secret. Dans ce contexte, les attaques profilées se révèlent particulièrement adaptées. Une implémentation polymorphe de cette fonction augmente la difficulté de réaliser l'attaque.

Dans le cadre d'une collaboration avec le laboratoire SAS de l'EMSE, nous évaluons une implémentation restreinte de l'application du polymorphisme de code sur une implémentation de AES [Abd+17]. Une nouvelle technique de pré-traitement des traces permet d'augmenter l'efficacité d'une attaque en CPA (*Correlation Power Analysis*) : les traces sont automatiquement alignées sur des motifs sélectionnés manuellement au préalable (e.g., motif d'accès mémoire). Cette technique de resynchronisation des traces permet d'attaquer une technique de dissimulation de l'état de l'art de CORON et al. [CK09 ; CK10] avec un nombre de traces du même ordre de grandeur qu'une implémentation non protégée, et permet d'améliorer l'efficacité de l'attaque en CPA sur une configuration simplifiée du polymorphisme de code. Cependant, la solution proposée repose sur une sélection manuelle des motifs de resynchronisation. Dans le cadre expérimental, cette solution était évaluée en boîte blanche (notamment, accès complet au code source et au code machine de la cible). On fait donc l'hypothèse d'un modèle attaquant idéal dans la phase préparatoire de l'attaque. Comme nous le verrons dans le chapitre suivant (section 3.1), cette hypothèse n'est pas toujours justifiée puisque cette phase préparatoire peut demander la plus

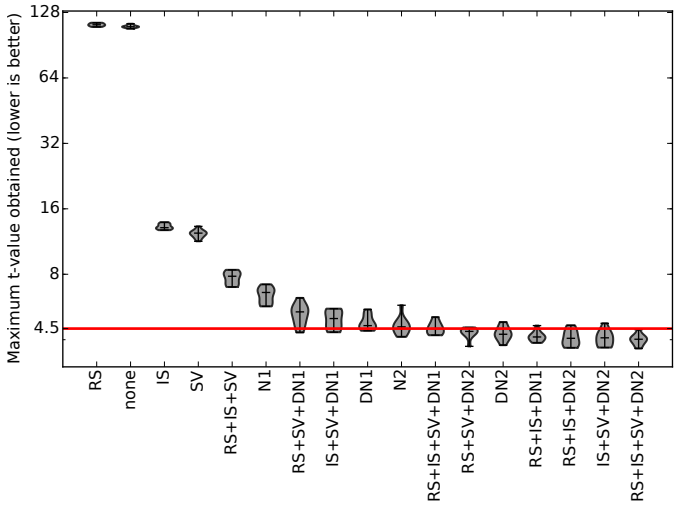


FIGURE 2.9 – Distribution des valeurs maximales de t-test pour 17 configurations différentes de polymorphisme de code (abscisse) appliquées sur une implémentation de AES. Chaque configuration est évaluée sur 10 exécutions (donnant 10 instances polymorphes différentes), et la valeur maximale du t-test est mesurée à chaque fois. Extrait de [Bel+19].

grande partie des efforts dans la réalisation d'une attaque.

Évaluation en sécurité : attaques avancées. Enfin, dans le cadre d'une collaboration avec le CESTI du CEA-Leti, nous réalisons une évaluation pratique du gain en sécurité apporté par le polymorphisme de code seul, utilisé sur un composant sur étagère inadapté aux implémentations sécurisées, et contre un attaquant à l'état de l'art [Mas+20].

Les implémentations polymorphes de AES évaluées correspondent aux configurations de niveau de sécurité le plus élevé publiées dans [Bel+19]. En pratique, il est possible d'utiliser des configurations du polymorphisme de code utilisant une variabilité comportementale encore plus forte, mais les configurations utilisées dans cette étude correspondaient à ce que nous jugions être un bon compromis entre le gain espéré en sécurité et le surcoût en performance.

On se place dans le modèle d'attaquant le plus puissant, à l'état de l'art. Les évaluateurs disposent d'un accès en boîte blanche à la cible, disposant d'un budget en temps conséquent, et d'un équipement à l'état de l'art. Concrètement, nous avons collaboré avec les évaluateurs pour détailler le fonctionnement de la contre-mesure et ses biais potentiels. L'évaluation envisage plusieurs modèles d'attaquants de force progressive [Mas+20, Sec. 2.5] : mise en œuvre automatisée d'attaques verticales ; resynchronisation des traces, à partir de motifs identifiés par un expert, pour la réalisation d'attaques verticales et d'attaques profilées ; attaque à l'aide des techniques d'analyse basées sur des réseaux de neurones convolutionnels, où la structure et les paramètres du réseau sont déterminés par la connaissance détaillée de la cible et du fonctionnement du polymorphisme de code.

Du point de vue de l'attaquant, ce travail aborde deux problèmes : (1) la mise à l'échelle des modèles de réseaux de l'état de l'art qui sont inadaptés en l'état pour analyser une quantité d'information aussi grande ; (2) la capacité des techniques d'analyse à identifier une fuite d'information de premier ordre dans des traces présentant une très grande variabilité.

Du point de vue de la sécurisation, cette évaluation évalue le potentiel du polymorphisme de code à améliorer le niveau de sécurité de systèmes autrement extrêmement vulnérables. Le poly-

morphisme de code étant utilisé seul, des fuites d'information de premier ordre sont exploitables en de nombreux points de chaque observation d'exécution. Cependant, les fuites d'information sont étalées sur une large fenêtre temporelle, ce qui du point de vue de l'attaquant suppose des moyens importants pour la mesure, le stockage et l'analyse des traces side-channel. Aussi, les conditions d'évaluations sont pessimistes, puisque les implémentations sont évaluées sur des composants sur étagère typiques de produits embarqués, mais dépourvus de contre-mesures logicielles ou matérielles autres que le polymorphisme de code, donc inadaptés à des produits de sécurité. En particulier, la fuite side-channel intrinsèque est importante et aisément identifiable, ce qui rend ces composants inadaptés pour des implémentations sécurisées [BS20].

Les résultats expérimentaux confirment nos résultats précédents concernant les attaques verticales [Mas+20, Tab. 1], à savoir que le polymorphisme de code apporte un gain de sécurité réel contre des attaques verticales automatisées. Dans l'évaluation réalisée sur deux implémentations logicielles de AES, cette attaque échoue avec le nombre maximal de traces exploitées, 10^5 traces. Les techniques de resynchronisation peuvent améliorer l'efficacité d'attaques verticales ou d'attaques profilées, mais leur efficacité est variable. Dans cette évaluation, la resynchronisation est effectuée à partir de motifs sélectionnés manuellement par un spécialiste, à partir d'un accès en boîte blanche à l'implémentation. Pour l'une des implémentations, l'attaque échoue en 10^5 traces, alors que pour l'autre implémentation évaluée, l'attaque réussit en moins de 10^3 traces. Enfin, cette évaluation confirme le potentiel des attaques par réseaux de neurones, puisque dans chaque cas la fuite d'information peut être exploitée en moins de 20 traces, ce qui correspond à l'ordre de grandeur du nombre de traces nécessaires pour exploiter la fuite d'information, par une attaque verticale simple, sur les mêmes implémentations non protégées. En conclusion, le polymorphisme de code, utilisé seul comme technique de dissimulation, n'est pas suffisant contre un attaquant à l'état de l'art, s'il est par ailleurs appliqué sur un composant sur étagère qui n'est pas conçu pour une application à la sécurité, mais il reste une solution intéressante pour augmenter significativement la difficulté d'exploitation des attaques par canal auxiliaire.

Impact sur les performances. Notre approche permet une haute flexibilité de la mise en œuvre de la contre-mesure, comme déjà suggéré plus haut dans la figure 2.9, ce qui implique une variabilité importante dans l'impact de la contre-mesure sur les performances. Cet impact est amplifié par la variabilité comportementale induite par la génération dynamique de code : chaque instance polymorphe présente une mise en œuvre différente de la fonction cible dont les effets sur le temps d'exécution et l'empreinte mémoire sont hautement variables.

La figure 2.10 illustre le surcoût sur le temps d'exécution, dans trois configurations choisies arbitrairement, en comparaison à la génération de code machine sans polymorphisme de code (configuration *none*) : le surcoût est très variable, il dépend à la fois de la configuration choisie et de la nature de la fonction cible choisie. Dans la figure 2.10, la configuration *none* illustre l'utilisation de la génération de code dynamique sans l'application de transformations polymorphes. En toute rigueur, dans ce cas, le code des instances polymorphes devrait être identique au code produit statiquement. Cependant, dans la mise en œuvre utilisée pour Odo dans [Bel+19], le code de référence, statique, est produit par GNU GCC (`arm-none-eabi`) alors que les instances polymorphes sont produites par le backend ARM de LLVM 3.8.0. Cette pénalité peut aller jusqu'à un facteur de ralentissement supérieur à $1,5$, qui impacte aussi la mesure de surcoût entre le code original et les configurations polymorphes choisies.

L'utilisation de la génération dynamique de code induit aussi un surcoût notable sur le temps d'exécution pendant la génération du code de la fonction cible. La mise en œuvre de la génération de code exploite intensivement des techniques de spécialisation du code des SGPC pendant la phase de compilation statique, ce qui nous permet d'obtenir un coût de la génération de code au runtime beaucoup plus faible que celui rencontré dans les techniques JIT

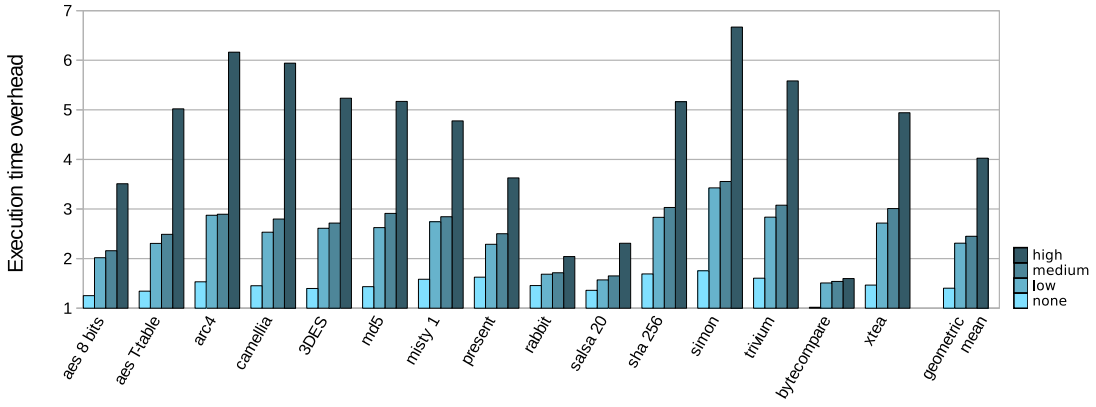


FIGURE 2.10 – Facteur de surcoût à l'exécution des instances polymorphes, en comparaison à une version de référence statique. Extrait de [Bel+19].

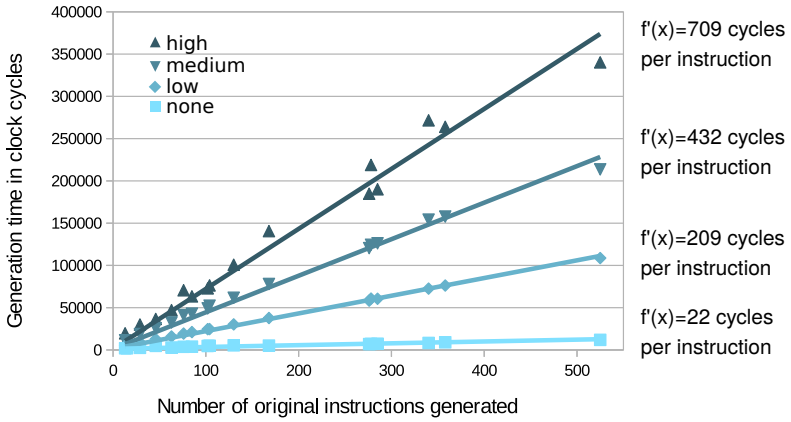


FIGURE 2.11 – Temps de génération de code en fonction de la configuration du polymorphisme de code et du nombre d'instructions originales dans le programme. Extrait de [Bel+19].

traditionnelles [Cha+14b ; ECC15b]. La figure 2.10 illustre l'influence de la configuration de polymorphisme de code choisie sur le temps de génération de code. On peut y noter également que le temps de génération de code est globalement linéaire avec le nombre d'instructions machine dans la fonction originale cible, quelque soit la configuration choisie. On peut noter enfin que la génération de code peut se faire en temps masqué (e.g., génération de code dans un buffer double) de sorte que le délai de traitement des données soit seulement impacté par le surcoût sur le temps d'exécution des instances polymorphes, et pas par le temps de génération de code.

2.2.1.4 Ouverture

Le polymorphisme de code est une technique logicielle de dissimulation puissante. Dans le domaine du side-channel, le principe de variabilité à l'exécution a été repris dans des implémentations matérielles intégrées au processeur [ABP21 ; Pha+21 ; LSB22]. Les techniques matérielles permettent de réduire fortement des surcoûts sur le temps d'exécution, mais impliquent l'utilisation d'un circuit spécifique, ce qui n'est pas possible dans tous les domaines d'application. Les

implémentations matérielles de l'état de l'art n'utilisent pas de génération dynamique de code, et en conséquence la variabilité apportée est intrinsèquement limitée aux instructions en vol dans le processeur. Cependant, l'effet cumulatif peut être important sur une trace d'exécution complète. Il serait intéressant d'envisager la combinaison de ces techniques matérielles avec des techniques logicielles travaillant à plus large échelle, par exemple à l'échelle du flot de contrôle complet plutôt qu'à l'échelle du bloc de base comme dans notre mise en œuvre actuelle.

Ce principe de variabilité est utilisé dans d'autres domaines d'application, par exemple l'obfuscation de code pour l'automobile [HSE22], ou la diversification de binaires WebAssembly pour limiter le potentiel d'exploitation d'une *payload* [Cab+22]. Il serait intéressant d'envisager l'apport potentiel de notre implémentation du polymorphisme de code pour ces autres usages.

Pour l'évaluation du polymorphisme de code, nous nous sommes placés volontairement dans les conditions les plus défavorables : nous avons visé le déploiement sur un système cible aux ressources limitées, dont la mesure de consommation électrique ou la signature électromagnétique est aisément exploitable. Récemment, on a vu les techniques d'attaques par canaux auxiliaires se déporter sur des cibles plus complexes comme des *System on Chip* [Gen+16 ; Lis+21 ; She+21 ; HA22]. Ces plateformes sont peut-être de meilleurs candidats à l'exploitation de techniques comme le polymorphisme de code : les capacités en calcul et en mémoire sont beaucoup plus importantes, donc le surcoût de la contre-mesure aurait un impact moindre relativement à l'ensemble de la pile logicielle, et les systèmes d'exploitation utilisés permettent plus facilement d'exécuter les SGPC en temps masqué.

Jusqu'ici, nous nous sommes intéressés à des micro-architectures simples : pipeline monoscalaire avec exécution dans l'ordre des instructions. Sur des micro-architectures plus complexes, les techniques d'optimisation matérielles comme le renommage de registres ou l'exécution dans le désordre peuvent annuler une partie de la variabilité créée par le polymorphisme de code, notamment par le mélange des instructions. Le portage du polymorphisme de code sur des processeurs plus complexes devra donc être re-pensé en fonction de la micro-architecture ciblée.

Un autre aspect important est la complexité de mise en œuvre de la contre-mesure : contrairement au déploiement habituel de code statique, qui utilise une seule étape de compilation, notre mise en œuvre nécessite trois phases successives : compilation source à source, puis compilation du code machine vers l'architecture cible, et enfin génération des instances polymorphes au runtime (figure 2.5). La mise en œuvre de cette technique de protection dans un produit demanderait donc des modifications importantes sur un flot de production logicielle habituel. Pour rejoindre la discussion dans le paragraphe précédent, une mise en œuvre sur des systèmes moins contraints, comme des systèmes d'exploitation de smartphones, pourrait s'appuyer sur les moteurs JIT qui y sont déjà présents (ou sur le moteur AOT — *Ahead Of Time* — de Android), donc réduire l'impact sur la chaîne logicielle existante. En outre, notre mise en œuvre rend le débogage plus complexe à cause des phases successives de génération de code. Il faut noter que la génération de code au runtime, par les SGPC, comporte une part d'aléa, mais qu'elle reste prédictible : pour une séquence d'aléa donnée par le PRNG, les instances polymorphes produites seront toujours identiques. Cette caractéristique est importante, en particulier pour le test. Concernant tous les points évoqués dans ce paragraphe, les techniques de *multi-versionning* comme MEET [Ago+15], basées sur une seule passe de compilation sans génération de code au runtime, permettent une intégration système plus simple.

Concernant la *dissimulation* comme technique de durcissement contre les analyses side-channel, l'impact de la dispersion temporelle (horizontale) de la fuite d'information sur les analyses verticales est bien formalisé dans la littérature [CCD00 ; MOP07]. Cependant, le polymorphisme de code a d'autres conséquences sur la fuite d'information : outre la dispersion temporelle, plusieurs transformations de code peuvent également produire d'autres effets variables dans la fuite d'information, e.g. par l'utilisation de registres architecturaux différents,

ou par l'introduction de motifs de fuite variables dus aux différentes séquences d'instructions utilisées pour réaliser une opération. Ces techniques, utilisées seules, ne sont pas suffisantes pour empêcher l'exploitation d'une fuite d'information, mais contribuent à augmenter la variabilité observée par side-channel, de la fuite d'information et de son contexte. Belleville et al. ont montré récemment que des techniques de dissimulation à large échelle, en particulier le polymorphisme de code, empêchent l'exploitation de la fuite d'information même par des analyses basées sur des réseaux profonds [BM24]. Ces résultats viennent mitiger les conclusions de notre étude précédente publiée à ESORICS [Mas+20]. Cela illustre le fait que les techniques d'analyse par apprentissage, très puissantes, sont susceptibles à de nombreux effets encore mal connus. En outre, chaque plateforme cible présente des caractéristiques différentes quant aux analyses side-channel. La formalisation de l'impact du polymorphisme de code, comme technique de dissimulation contre les analyses side-channel, est donc une question ouverte qui mêle des aspects théoriques et des aspects expérimentaux.

2.2.2 Masquage

Pendant la thèse de Nicolas Belleville, nous avons également travaillé sur l'application automatisée d'une contre-mesure de masquage Booléen au premier ordre, mise en œuvre dans une chaîne de compilation prototype basée sur LLVM, appelée *Maskara* [Bel+20a]. Cette approche permet d'ouvrir deux verrous dans les prototypes d'outils pour la mise en œuvre du masquage qui existent alors dans l'état de l'art :

— Maskara n'implique pas de limitation sur les structures de flot de contrôle des programmes à compiler. A contrario, les outils de l'état de l'art nécessitent souvent d'aplatir le flot de contrôle avant l'application du masquage (e.g., par déroulage de boucle), ou bien permettent d'exprimer des structures de flot de contrôle mais produisent du code sans flot de contrôle (e.g., [Mos+12 ; Bar+16]). Le support de structures de flot de contrôle implique des modifications profondes dans la manière de traiter les analyses de masquage, qui sont brièvement présentées plus loin dans cette section.

— Maskara permet de masquer automatiquement les fonctions non-linéaires implémentées sous la forme le tables de correspondance (*lookup tables* ou LUT), qui sont typiquement utilisées pour les implémentations des S-box en cryptographie symétrique. Les approches de l'état de l'art supposent que l'utilisateur identifie au préalable la S-box, ou une stratégie de masquage spécifique comme par exemple [RP10] qui exploite les propriétés de la fonction SubBytes de AES. Pour assurer une sécurité de la mise en œuvre, il est en théorie nécessaire de rafraîchir toute la table masquée après chaque accès à son contenu, ce qui n'est pas toujours fait en pratique en raison du coût en performance que cela implique [HOM06]. Dans Maskara, les accès à la table associative dépendant d'une variable secrète sont exprimés sous la forme d'une évaluation masquée d'une interpolation polynomiale de la table. Chaque évaluation de la table associative masquée est coûteux, mais permet d'éviter le rafraîchissement utilisé dans la technique conventionnelle où la trace d'exécution du remasquage peut être exploitée pour retrouver la valeur du masque [TWO14]. Maskara propose une approche outillée et automatisée de la technique proposée par Coron, Roy et Vivek [CRV14], en y apportant des améliorations visant à réduire le coût de mise en œuvre. Cette méthode de masquage a été choisie parce qu'elle n'impose pas de pré-requis sur la nature de la table associative à masquer, et parce qu'elle supporte un ordre de masquage arbitraire.

La vue d'ensemble du fonctionnement de Maskara est illustrée en figure 2.12. La mise en œuvre du masquage requiert plusieurs étapes d'analyses de code et de transformations, mises en œuvre dans une passe dédiée insérée à la fin du middle-end de LLVM. En outre, la passe de

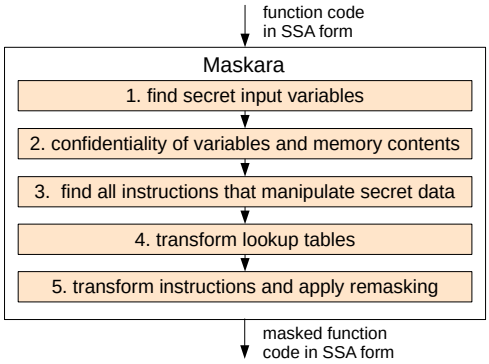


FIGURE 2.12 – Vue d'ensemble des étapes pour l'application du masquage dans Maskara. Extrait de [Bel+20a].

sélection d'instructions du back-end est modifiée pour éviter certaines fuites que nous n'avions pas pu caractériser finement, faute de temps [Bel19, Section 4.6.3]. Depuis les travaux entrepris à la suite de cette thèse (section 2.2.2.7, p 34) , nous savons que ces effets impliquaient probablement des fuites architecturales, par exemple dues à des fuites en transitions les registres, et des effets micro-architecturaux dus à des fuites sur des éléments cachés du logiciel.

2.2.2.1 Analyse de confidentialité

Dans l'implémentation prototype de Maskara, la passe d'analyse fait l'hypothèse que chaque variable masquée d'entrée d'une fonction est représentée sous la forme d'un tableau de shares, et que le code à masquer manipule le premier élément de ce tableau.

L'analyse de confidentialité met en œuvre un système de typage et des règles de propagation qui permettent de déterminer le niveau de confidentialité de chaque variable intermédiaire (i.e., public ou secret) dans une fonction à masquer [Bel+20a]. Le système de typage de confidentialité est similaire au typage utilisé par MOSS et al. [Mos+12], avec la différence que dans Maskara le type *secret* est décliné en plusieurs niveaux de confidentialité associés au nombre de déréférencements d'une variable [Bel+20a, Figure 3 et Figure 4], ce qui permet de suivre la propagation de la confidentialité d'une variable dans une succession d'indirections.

2.2.2.2 Masquage des accès aux tables constantes

La méthode employée est décrite ici de manière succincte. Sa mise en œuvre, complexe, est détaillée dans la publication à la conférence CASES 2020 [Bel+20a] et illustrée par des exemples dans le manuscrit de thèse de Nicolas Belleville [Bel19].

Les tables constantes (LUT, *Lookup Tables*) sont identifiées par les instructions de chargement mémoire dont l'adresse de base est constante et dont l'index est secret. Pour le masquage des tables, en particulier les S-box tabulées utilisées en cryptographie symétrique, la méthode la plus couramment employée consiste à utiliser une table dont le contenu est masqué [HOM06]. En théorie, après chaque accès à la table masquée, cette technique implique de rafraîchir l'intégralité du contenu de la table, ce qui implique un coût prohibitif. En pratique, la table est donc rafraîchie moins fréquemment, typiquement à chaque ronde ou à chaque exécution de la primitive cryptographique. TUNSTALL, WHITNALL et OSWALD ont également montré qu'une seule trace d'exécution du remasquage de la table est suffisante pour révéler le secret [TWO14].

La technique de masquage utilisée dans Maskara est une application de la méthode publiée par Coron, Roy et Vivek [CRV14]. Cette méthode s'applique à n'importe quelle table constante même si elle sera particulièrement utile pour le masquage des S-boxes en cryptographie symétrique. L'idée est d'identifier un polynôme interpolateur dans $\mathbb{F}_{2^n}$ du contenu de la table [Car+12]. L'accès à la table masquée est remplacé par une évaluation masquée de ce polynôme interpolateur. Contrairement aux méthodes qui consistent à calculer une nouvelle table masquée, l'utilisation d'un polynôme interpolateur ne souffre pas de la vulnérabilité publiée dans [TWO14], et supporte naturellement le masquage à l'ordre supérieur. En revanche, le polynôme interpolateur aura le plus souvent un degré égal à la taille de la table à masquer, ce qui suppose une évaluation coûteuse, même si en pratique elle reste moins coûteuse que la méthode de remasquage des tables.

La méthode publiée par Coron, Roy et Vivek cherche à minimiser le nombre de multiplications utilisées dans l'évaluation du polynôme interpolateur, puisque ce sont des opérations non linéaires coûteuses à réaliser dans un schéma de masquage Booléen [CRV14]. En particulier, dans le calcul des termes du polynôme, un certain nombre de multiplications est évité en favorisant les élévations au carré qui, contrairement à la multiplication, sont peu coûteuses [RP10]. Les autres opérations de l'opération polynomiale (additions et fonctions affines) sont également simples à réaliser dans un schéma de masquage Booléen. Le masquage de toutes les opérations impliquées dans l'évaluation du polynôme interpolateur de la table consiste à appliquer les techniques proposées par Rivain et Prouff [RP10].

À cela, Nicolas Belleville a proposé les améliorations suivantes :

— La multiplication dans $\mathbb{F}_{2^n}$ est évaluée à l'aide de tables de log et alog [RP10]. La représentation de ces tables est étendue de sorte à éviter des tests de valeur nulle des paramètres de la fonction log, et de sorte à éviter le calcul du modulo$(2^n - 1)$. Ces optimisations permettent de réduire significativement le nombre d'opérations machine pour chaque multiplication, au prix d'une plus grande utilisation de la mémoire (figure 2.13, *Field mul zero*, et *Field mul modulo*).

— La technique d'interpolation polynomiale de Coron, Roy et Vivek consiste à choisir arbitrairement un certain nombre de monômes (les critères de choix sont détaillés par exemple dans [Bel19]), puis, par réduction, à identifier les autres termes polynomiaux impliqués dans le polynôme interpolateur [CRV14]. Il existe un grand nombre d'ensemble de monômes permettant d'aboutir à un polynôme interpolateur de la table ciblée. Nicolas Belleville propose d'itérer sur le choix des ensembles de monômes candidats afin de sélectionner les ensembles comprenant des monômes nuls, de sorte que l'évaluation du polynôme interpolateur soit moins coûteuse. Cette recherche d'ensembles particuliers peut nécessiter un temps de calcul important mais néanmoins accessible (de l'ordre de plusieurs heures sur une machine de développement standard), variable suivant la taille de la table à interpoler (donc de la taille du polynôme interpolateur résultant) et des contraintes sur le nombre de monômes nuls. Ce temps de calcul est cependant payé une fois pour toutes, à la compilation, et les résultats de cette recherche peuvent être mémorisés pour un ré-emploi dans des masquages ultérieurs de tables de même taille [CRV14] (figure 2.13, q_i *coefficients*).

— Enfin, l'ordre d'évaluation des monômes exploités dans le polynôme interpolateur est judicieusement choisi de sorte à maximiser le ré-emploi des termes intermédiaires (figure 2.13, intégrée à la mesure de référence *Poly reference*).

2.2.2.3 Rafraîchissement des masques (remasquage)

L'analyse de remasquage associe à chaque variable secrète un ensemble de listes de masques (i.e., plutôt qu'une liste de masques). En effet, le contenu d'une variable peut dépendre du

chemin d'exécution pris, et de la même manière, la liste des masques appliqués à une variable secrète peut dépendre du chemin d'exécution pris. L'analyse applique un remasquage dès qu'un démasquage est détecté dans une des listes de masques. Le remasquage consiste à injecter un nouveau masque (i.e., une valeur aléatoire) dans chaque share d'une variable secrète, typiquement [RP10, Algorithme 4]. L'injection de valeurs aléatoires est traitée par l'appel à une fonction spécifiée par l'utilisateur à la compilation.

L'analyse de remasquage doit maintenir plusieurs ensembles de listes de masques : un ensemble dit *word-level* considérant une analyse à la granularité de la variable ainsi qu'un ensemble pour l'analyse séparée de chaque bit de la variable. Les ensembles de masques au niveau bit se sont avérés nécessaires pour l'analyse de certaines opérations bit-à-bit, comme les rotations ou les décalages du contenu d'un registre, où une analyse word-level n'est pas capable d'identifier certains démasquages potentiels (e.g., [Bel19, Algorithme 7, lignes 11-15]). En théorie, il suffirait d'exploiter un ensemble pour chaque bit, mais la manipulation d'ensembles word-level s'avère plus efficace en l'absence d'opérations bit-à-bit.

L'analyse de confidentialité (section 2.2.2.1) exploite une approche statique, basée uniquement sur la connaissance des masques exploités et du nombre d'opérandes secrets d'une instruction. Cette analyse est moins précise que des analyses de dépendance statistique comme SELA [MPH23] ou LeakageVerif [MOH20], mais elle est sûre, en appliquant un remasquage dès que la combinaison de variables masquées pourrait donner lieu au démasquage du secret.

2.2.2.4 Gestion des boucles

Les structures de flot de contrôle posent une difficulté particulière dès que des variables secrètes y interviennent. La difficulté vient de l'analyse de la propagation des variables masquées et de leurs masques dans les boucles, qui peut dépendre du chemin pris à l'exécution. Dans Maskara, on apporte des solutions pour deux points : la jointure de chemins d'exécution impliquant des secrets (e.g., en sortie d'une structure `if-then-else`), et la propagation des secrets dans les itérations de boucles.

Jointure de chemins d'exécution impliquant des secrets. Maskara travaille dans le middle-end de LLVM, donc sur une forme SSA où les résolutions de variables aux jointures de chemins d'exécution sont traitées par des nœuds ϕ. Lorsqu'un nœud ϕ a au moins une variable secrète pour opérande, on procède comme pour le masquage d'opérations linéaires :

— les nœuds ϕ ayant un opérande secret `s` et un opérande public `p` sont dupliqués pour chaque share de `s` : ϕ`(s, p);` devient ϕ`(s0, p);` ϕ`(s1, e);`, où `e` est l'élément neutre de la fonction d'encodage des shares pour le schéma de masquage ciblé (e.g., 0 pour le masquage Booléen quand les shares sont calculés par la fonction `xor`).

— les nœuds ϕ à deux opérandes secrets, `x` et `y`, sont exprimés sous la forme : ϕ`(x0, y0);` ϕ`(x1, y1);`.

Propagation des secrets dans les itérations de boucles. Une analyse de propagation spécifique est nécessaire en présence de boucles, puisque chaque itération peut modifier la liste des masques associée à une variable secrète, et donc donner lieu à un démasquage. L'analyse de remasquage simule la propagation des masques au fil des itérations de boucle jusqu'à ce qu'un point de convergence soit identifié [Bel+20a, Algorithme 2]. L'analyse s'interrompt en cas de démasquage avéré, ou après un nombre d'itérations borné par l'utilisateur en cas de divergence. Dans ces conditions, un remasquage est systématiquement appliqué avant le nœud ϕ.

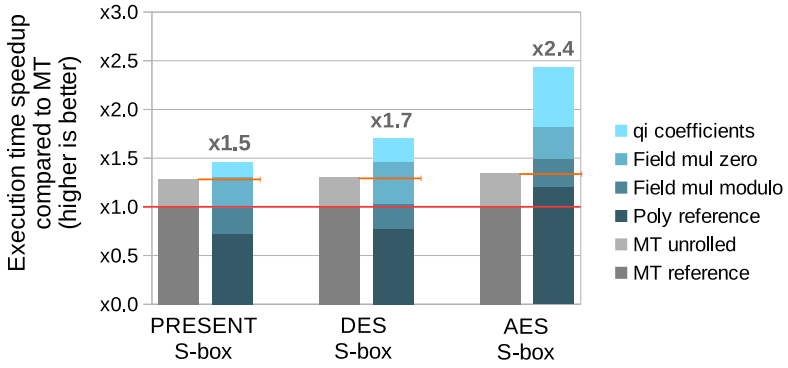


FIGURE 2.13 – Gain sur le temps d'exécution de l'accès aux tables masquées dans la mise en œuvre utilisée dans Maskara, en comparaison avec l'approche par masquage des tables (*MT*), où on tient compte du temps d'accès à la table puis du remasquage de la table. Les valeurs *Poly reference* correspondent à l'application de la technique originale dans [CRV14], et les valeurs q_i *coefficients*, *Field mul zero*, et *Field mul modulo* correspondent à l'application d'optimisations supplémentaires sur l'approche originale. Extrait de [Bel+20a].

TABLEAU 2.3 – Nature des opérations masqués utilisées dans AES, nombre d'utilisations de type d'opération et temps d'exécution de leur implémentation masquée. Les lignes MT indiquent le temps d'exécution pour l'utilisation d'une table masquée dont le code est déroulé (version la plus rapide). Extrait de [Bel+20a].

	Nature of the operations manipulating a secret							**Original implementation**	**Masked implementation**	
AES subfunctions	xor	load	store	sign ext.	shift	and imm.	S-box access	time (cycles)	time (cycles)	overhead
AddRoundKey	16	32	16	0	0	0	0	174	354	x2.0
SubBytes	0	16	16	0	0	0	16	134	17509	x130.7
SubBytes (MT)									32533	x242.8
ShiftRows	0	16	16	0	0	0	0	38	55	x1.4
MixColumns	68	16	16	16	32	16	0	260	454	x1.7
KeySchedule	17	16	16	0	0	0	4	90	4533	x50.4
KeySchedule (MT)									8327	x92.2

2.2.2.5 Évaluation en performance

La figure 2.13 illustre le coût important du masquage des tables, qui s'avère néanmoins plus faible que l'approche par masquage des tables quand le contenu de la table masquée est rafraîchi après chaque accès en lecture. On note aussi que les optimisations ajoutées dans Maskara apportent une réduction significative du surcoût lié au masquage par rapport à la méthode publiée dans [CRV14] (mesures *Poly reference* dans figure 2.13). Le tableau 2.3 détaille le surcoût pour chaque opération utilisée dans l'implémentation de l'AES, illustrant que c'est le masquage de la S-box qui est à l'origine de la plus grande partie du surcoût. À titre d'illustration, le masquage de la fonction de chiffrement Simon [Bea+15], qui utilise seulement les opérations *et booléen*, *rotation*, et *ou exclusif*, et qui ne contient pas de S-box, induit un surcoût de $\times 1,5$ seulement [Bel19]).

2.2.2.6 Analyse de sécurité du masquage au niveau binaire

Dans le cadre d'une collaboration avec l'équipe de Karine Heydemann au LIP6, la protection des codes compilés par Maskara est vérifiée formellement par une approche symbolique [Ben+19 ; Ben21]. Cette vérification a permis d'identifier et de corriger quelques problèmes dans l'implé-

mentation de Maskara, et permet de garantir l'absence de fuite d'information dans un modèle de fuite en valeur.

Dans la thèse de Nicolas Belleville, on procède à une mesure des fuites réelles sur une implémentation de AES masquée par l'approche présentée ici, réalisée sur un cœur ARM Cortex-M3 dans une carte STM32F1. L'implémentation est vérifiée formellement comme correcte dans un modèle de fuite en valeur. Cependant, un t-test non-spécifique, sur des mesures de rayonnement électro-magnétique, révèle plusieurs points de fuite d'information, ce qui implique que le modèle de fuite en valeur n'est pas suffisant pour représenter correctement la fuite d'information même sur une plateforme aussi simple que le STM32F1. Dans l'hypothèse de fuites en transition de sources diverses, le back-end du compilateur est modifié pour insérer des instructions factices visant à casser des transitions hypothétiques. On procède à une recherche empirique des sources possibles de fuite d'information, en appliquant des motifs permettant de casser certaines transitions supposées puis en ré-itérant une vérification par t-test. En fin de compte, sur la plateforme utilisée pour l'évaluation et pour l'implémentation AES testée, on constate qu'il est nécessaire de combiner plusieurs motifs ciblant plusieurs sources de fuites en transition, en particulier liées aux accès à la mémoire et aux opération de calcul dans l'ALU.

2.2.2.7 Ouverture

La technique de masquage mise en œuvre dans Maskara est générique puisqu'elle n'impose pas de restrictions sur le flot de contrôle et sur les tables associatives. Elle permet donc de couvrir une grande palette d'implémentations. À notre connaissance, il s'agit de la seule approche d'application automatisée de masquage qui ne fait pas appel à des techniques spécialisées pour les fonctions linéaires : en général, les outils de masquage (e.g. maskComp [Bar+16]) supportent la S-box de l'AES seulement lorsqu'elle n'est pas tabulée, e.g. exprimée comme une transformation affine de l'inverse multiplicative.

Le fait de conserver les structures de contrôle rend l'analyse de masquage plus difficile puisqu'il est nécessaire de conserver une liste de masques associée à chaque variable secrète pour chaque chemin d'exécution. Lorsqu'une instruction combine deux variables secrètes, l'analyse actuelle considère toutes les recombinaisons possibles entre chaque liste de masques des deux opérandes. Cette approche est sûre mais imprécise, puisque cela revient à considérer toutes les combinaisons de chemins d'exécution, y compris celles impossibles, ce qui peut donner lieu à des remasquages inutiles. Il serait possible de rendre cette analyse plus précise en faisant une évaluation (éventuellement symbolique) des conditions d'exécution de chaque chemin d'exécution. Aussi, dans les boucles, on simule la propagation des masques à chaque itération mais sans considérer les bornes de boucles. Sachant que les implémentations cryptographiques utilisent des bornes de boucles fixes, qu'il est possible de déterminer par une analyse statique, la connaissance des bornes de boucles pourrait rendre l'analyse de masquage plus précise.

Notre évaluation en sécurité a mis en évidence des fuites d'information résiduelles. La vérification formelle du code au niveau binaire, donc tel qu'il est exécuté sur la plateforme cible, permet a priori d'écarter des problèmes de mise en œuvre du schéma de masquage. On s'appuie ici sur un modèle de fuite en valeur, on peut donc supposer que les fuites observées correspondent à d'autres modèles de fuite. Cette observation est cohérente avec les hypothèses de fuite en transition dont les effets sur les implémentations logicielles sont, à l'époque de ces travaux, déjà montrés dans la littérature [Bal+15 ; de +17], même si les effets de fuite dus à la micro-architecture ne sont pas bien connus. Face à ces constats, Karine Heydemann et moi lançons courant 2019 l'idée d'un nouveau projet de recherche pour mieux caractériser les effets de fuite d'information induits par la micro-architecture des processeurs. Le projet IDROMEL démarre début 2021, coordonné par Vincent Migliore au LAAS. Entre temps, la communauté scientifique aborde aussi ce sujet, avec

des publications marquantes comme *MIRACLE* [MPW22] ou *Power Contracts* [Blo+22].

ARMISTICE [dHM22] constitue une avancée par rapport à l'approche de vérification formelle utilisée pendant la thèse de Nicolas Belleville, en basant l'analyse de fuite d'information sur un modèle de la micro-architecture du processeur cible. Cette analyse aurait permis d'identifier précisément les sources de fuites que nous avons observées en pratique (et probablement d'autres). En revanche, la mise en œuvre automatisée d'un schéma de masquage robuste à ces sources de fuites suppose une approche plus élaborée que les heuristiques employées jusqu'ici dans le back-end de Maskara. Cette question de recherche a été abordée dans la thèse de Lorenzo Casalino, récemment soutenue [Cas24]. La thèse montre premièrement comment intégrer un modèle précis de la micro-architecture du processeur cible dans le compilateur. Le modèle est exploité dans le back-end du compilateur par l'allocation de registres et par l'ordonnancement d'instructions pour produire un code optimisé et robuste aux fuites architecturales et micro-architecturales. Cette approche suppose de modéliser précisément la micro-architecture cible, ce qui n'est pas toujours possible en pratique, par exemple pour des raisons de propriété intellectuelle. Aussi, cette approche rend le code compilé dépendant de l'implémentation micro-architecturale du processeur cible, ce qui n'est pas toujours souhaitable, par exemple lorsqu'on souhaite produire des bibliothèques portables. La thèse de Lorenzo Casalino aborde donc une deuxième approche, orthogonale, qui consiste à étudier la résistance intrinsèque de différents schémas de masquage aux effets de fuite d'information induits par la micro-architecture des processeurs. On envisage l'application de schémas de masquage, comme l'*Inner Product* [BFG15], qui supportent des modèles de fuites plus élaborés que la fuite par valeur, ces schémas étant donc potentiellement robustes aux fuites dues à la micro-architecture des processeurs. On constate cependant que les implémentations évaluées, vérifiées dans un modèle de fuite en valeur, sont toutes vulnérables à exploitations de fuites de premier ordre. Ces résultats nous laissent supposer qu'il est nécessaire de considérer à la fois des schémas de masquage intrinsèquement robustes dans des modèles de fuites complexes, et la sécurité de leur implémentation sur une micro-architecture ciblée.

2.3 Conclusion

Dans ce chapitre, on s'intéresse au potentiel du compilateur comme outil d'aide à l'application de contre-mesures logicielles contre les attaques matérielles.

La première section s'attache à montrer comment exploiter un compilateur pour l'application de contre-mesures contre les attaques par injection de fautes : le compilateur permet une application automatisée de la contre-mesure, et peut apporter des leviers d'optimisation et de configuration supplémentaires dans l'implantation de la contre-mesure. La section 2.1.1 montre comment automatiser l'application d'un schéma de tolérance aux sauts d'instructions. Nous montrons comment exploiter les stratégies d'optimisation du back-end du compilateur pour réduire le surcoût de la mise en œuvre originale du schéma de protection. Le modèle d'attaquant couvert par le schéma de protection initial (une faute permettant de sauter une instruction) est élargi dans notre mise en œuvre pour couvrir des fautes multiples permettant de sauter un nombre borné (mais paramétrable) d'instructions consécutives. L'exploitation d'un outil comme le compilateur pour l'application d'une contre-mesure simplifie l'utilisation d'un modèle de protection paramétrable, avec lequel il est possible de tester rapidement différentes implémentations de la contre-mesure pour évaluer les impacts en sécurité et en performance.

La section 2.1.2 présente l'application automatisée d'un mécanisme de traceur, qui vise la protection du flot de contrôle. La contre-mesure est implantée au niveau du code machine afin d'étendre la couverture du schéma de protection original : en contrôlant l'entrelacement des instructions du programme cible avec les instructions des signatures, et en apportant la protection

des branchements conditionnels. En outre, le compilateur permet de placer judicieusement les instruction de vérification du flot de contrôle, en exploitant l'alignement des instructions machines sur des demi-mots permise par l'architecture ARMv7-M. De la sorte, il devient possible de détecter des fautes sur le chargement de mots d'instruction. Ces techniques de mise en œuvre, qui exploitent les détails de l'architecture cible, ARMv7-M, supposent une analyse minutieuse du code machine du programme et de son entrelacement avec le code de la contre-mesure, qu'il serait coûteux de mettre en œuvre sans le support d'un outil comme le compilateur. Nous montrons enfin comment l'utilisateur du compilateur peut activer sélectivement les portions du code à protéger, et peut choisir un niveau de protection adapté au modèle de menace à couvrir. Dans nos évaluations expérimentales, le code obtenu présente un surcoût systématiquement inférieur à la mise en œuvre originale de la contre-mesure sans support du compilateur.

La seconde section du chapitre traite de l'exploitation du compilateur pour l'application de contre-mesures contre les attaques par canaux auxiliaires. Nous présentons la mise en œuvre de deux contre-mesures : une contre-mesure de dissimulation, appelée polymorphisme de code, qui apporte une variabilité comportementale à l'exécution, et une contre-mesure de masquage.

Le polymorphisme de code est appliqué automatiquement, à partir du code source, par une chaîne de compilation spécifique en plusieurs phases. La dernière phase de génération de code se déroule au runtime, sur le système cible, qui est souvent un système embarqué contraint en ressources de calcul et en mémoire. La re-génération de code exploite une part d'aléa de manière à produire une succession d'implémentations toutes différentes mais toutes fonctionnellement équivalentes, appelées instances polymorphes. Notre mise en œuvre permet de tenir compte de ces contraintes en exploitant une technique de génération de code rapide et à faible empreinte mémoire. En outre, nous réduisons la surface d'attaque liée à l'utilisation de la génération de code au runtime par divers mécanismes, entre autres par un contrôle fin de l'activation des droits en écriture sur la mémoire programme, par le calcul automatisé de la taille des buffers recevant le code généré, et par le contrôle des débordements de la mémoire programme. La contre-mesure de polymorphisme de code est flexible, ce qui permet à son utilisateur d'explorer divers compromis performance-sécurité dans un espace de configuration vaste. La mise en œuvre de la contre-mesure travaille sur un modèle de programme qui tient compte de l'architecture du processeur ciblé : elle vise à apporter une très grande variabilité dans la génération des instances polymorphes en exploitant plusieurs stratégies d'implémentations sur l'architecture cible sans porter atteinte aux propriétés fonctionnelles des portions de code ainsi protégées. De la sorte, la contre-mesure peut être appliquée à tout type de programme, i.e. pas seulement aux implémentations cryptographiques qui sont la cible privilégiée des attaques par canaux auxiliaires. Nous réalisations plusieurs évaluations de sécurité, qui montrent que le polymorphisme de code augmente significativement la difficulté de réaliser les attaques dites verticales. Dans un modèle d'attaquant avancé qui exploite des techniques par apprentissage à l'aide de réseaux profonds, la difficulté majeure pour l'attaquant consiste à identifier la structure et la configuration du réseau qui permettront une bonne caractérisation de la fuite d'information. Une fois l'apprentissage réalisé, la fuite d'information, de premier ordre, peut être exploitée. Il faut noter toutefois que toutes nos évaluations expérimentales se sont déroulées sur des processeurs sur étagère, qui ne sont pas conçus pour résister aux attaques matérielles. Sur ces cibles, la fuite d'information intrinsèque à la plateforme est aisément exploitable, ce qui met l'attaquant dans une position favorable. Il aurait été intéressant de mesurer l'apport du polymorphisme de code sur des cibles où la fuite d'information est intrinsèquement plus difficile à exploiter.

La fin de ce chapitre présente la compilation d'un schéma de masquage Booléen au premier ordre. La technique de masquage mise en œuvre est générique puisqu'elle n'impose pas de restrictions sur le flot de contrôle et sur les tables associatives. De la sorte, notre mise en œuvre ne suppose pas de prérequis et n'impose pas de limitations sur les implémentations à protéger, elle

permet donc de couvrir une grande palette d'implémentations cibles. Aussi, on peut noter que plusieurs aspects de cette contre-mesure donnent lieu à une combinatoire complexe qu'il serait difficile à analyser sans l'aide d'un outil comme le compilateur : les analyses de propagation des masques, en combinaison avec la structure de contrôle du programme cible ; le calcul des interpolations polynomiales de tables associatives et la génération du code d'évaluation masquée des interpolations polynomiales. Nous combinons l'application outillée de la contre-mesure avec une méthode de vérification formelle appliquée après la compilation sur le code machine tel qu'il serait exploité dans le système cible. Cette approche permet de séparer les problématiques de l'application de la contre-mesure et de sa vérification sécuritaire tout en garantissant la conformité de la mise en œuvre du schéma de protection. Notre évaluation de sécurité sur une implémentation concrète montre néanmoins des fuites d'information résiduelles qui ne sont pas prises en compte dans le modèle de sécurité initial. Ces fuites d'informations sont dues à des éléments matériels qui ne sont pas modélisés dans le modèle de sécurité utilisé pour la conception de la contre-mesure, ce qui ouvre la question d'une articulation plus fine entre les modèles du logiciel et du matériel.

Dans ce chapitre, nous montrons comment mécaniser l'application de contre-mesures contre les attaques matérielles à l'aide d'un compilateur. Les contre-mesures ciblent des composants génériques, sur étagère, qui à l'origine ne sont pas dédiés aux implémentations de sécurité. De fait, nos évaluations de sécurité illustrent la difficulté d'atteindre un niveau élevé de sécurité face à des modèles d'attaquants puissants sans s'appuyer sur des architectures matérielles dédiées. Dans le chapitre suivant, nous abordons cette question, à savoir une meilleure articulation du matériel et du logiciel dans la conception de contre-mesures, pour atteindre des implémentations robustes à des modèles d'attaquants puissants.

Chapitre 3

Sécurisation par association logiciel-matériel

Nos premiers travaux sur la sécurisation contre les attaques matérielles ont montré qu'il est possible d'améliorer la robustesse d'un composant embarqué sur étagère par l'application de protections logicielles. Cependant, l'état de l'art des attaques évolue rapidement. D'une part, les techniques d'attaques se démocratisent, avec des outils accessibles sur catalogue pour un coût modéré [OC14], et avec la publication d'outils d'analyse aussi exploitables dans des attaques [CB23 ; SSG23]. Aussi, les attaques à l'état de l'art sont de plus en plus puissantes, notamment dans le domaine des attaques en fautes pour leur capacité à cibler précisément des éléments d'un circuit, comme un bit en mémoire [Col+19] ou un transistor [Anc+17]. D'autre part, les attaques matérielles, qui privilégiaient historiquement le domaine restreint de la carte à puce, peuvent aujourd'hui cibler à la fois des produits grand public de l'Internet des objets [CH17 ; Ron+17 ; MC19a], et des systèmes sur puce complexes comme ceux qu'on trouve dans un smartphone [Tro+21 ; Fan+22]. Par conséquent, on observe à la fois une démocratisation des moyens d'attaque, une montée en puissance du modèle d'attaquant à l'état de l'art, et la possibilité de cibler une grande part des systèmes numériques aujourd'hui en usage. Tout ceci suppose des moyens de protection à large couverture et capables en même temps de prévenir des exploitations très localisées.

Dans cet axe de recherche, je me suis intéressé à la conception de contre-mesures s'appuyant sur une meilleure association matériel-logiciel pour apporter des solutions de sécurisation face à un modèle d'attaquant à l'état de l'art ou au-delà. Dans la continuité des travaux précédents, on vise des mécanismes de protection génériques dont la composante logicielle est applicable de manière outillée sans imposer de restrictions sur les programmes à protéger. La première section de ce chapitre, section 3.1, traite des attaques par canal auxiliaire, où nous nous sommes intéressés à la combinaison de contre-mesures et à l'élargissement du périmètre de protection pour couvrir un scénario complet d'attaque. La section 3.2 présente ensuite une technique de durcissement des processeurs contre les attaques par injection de fautes dans un modèle d'attaquant puissant, au-delà de l'état de l'art.

3.1 Exécution polymorphe de code chiffré

Principaux éléments contextuels de cet axe de recherche

Collaborations	Lionel Morel (CEA List, CITI/INSA Lyon), Thomas Hiscock (CEA Leti), Olivier Savry (CEA Leti)
Publications	[CHS17 ; MC19a ; MCH21]
Projets liés	Projet SERENE-IoT, projet CLAPs

La première contre-mesure étudiée, le polymorphisme de code (section 2.2.1), est une contre-mesure de dissimulation contre les attaques par canal auxiliaire qui se prête naturellement à une combinaison avec des contre-mesures de masquage. Cependant, ce type de combinaison pose un certain nombre de problèmes dont nous discuterons en section 4.2, page 60. D'autre part, de par son aspect dynamique, le polymorphisme de code présente des caractéristiques intéressantes pour le durcissement d'implémentations par obfuscation. Nous n'avons pas étudié l'intérêt du polymorphisme de code sous cet angle, mais il est probable que des mécanismes avancés exploitant de la génération dynamique de code présentent de l'intérêt pour le durcissement contre la rétro-ingénierie. C'est dans le cadre de ces réflexions que je me suis rapproché de Thomas Hiscock et d'Olivier Savry, au CEA Leti, qui travaillaient sur un support matériel pour l'exécution de code chiffré [HSG19]. Nous avons envisagé comment articuler le polymorphisme de code avec du chiffrement de code. L'idée a d'abord été brevetée [CHS17], puis dans le cadre d'un séjour de Lionel Morel au CEA, nous avons développé une preuve de concept complète appelée PolEn [MCH21]. L'idée est d'articuler deux contre-mesures, le polymorphisme de code et le chiffrement de code, pour qu'elles se protègent mutuellement et que leur combinaison apporte une protection globale plus large.

3.1.1 Problématique et motivation

On considère qu'une attaque par canal auxiliaire se décompose en deux grandes phases distinctes : (i) une phase d'*analyse*, de prise de connaissance de la cible afin d'identifier des vulnérabilités potentielles et les vecteurs d'attaques les plus efficaces ; (ii) une phase d'*exploitation,* mettant en application une technique d'attaque particulière et le plus souvent connue. La communauté scientifique s'intéresse essentiellement à la question de la faisabilité de la phase d'exploitation.

La phase d'analyse est une phase préparatoire de prise de connaissance de la cible. Elle peut s'avérer triviale pour des produits non sécurisés, mais notre thèse est que, dans la pratique, il s'agit d'une étape incontournable qui peut représenter la majeure partie de l'effort d'un attaquant. Ces aspects sont cependant négligés par la communauté de la sécurité matérielle : la rétro-ingénierie couvre un grand nombre de techniques artisanales et les techniques de durcissement manquent de métriques objectives pour en mesurer l'efficacité. D'autre part la recherche puis la publication de vulnérabilités sur des produits sont très encadrées du point de vue juridique, limitant de telles initiatives même si les choses évoluent progressivement. La rétro-ingénierie est donc considérée comme présentant peu d'intérêt pour la communauté scientifique [BS20]. On trouve cependant dans la littérature quelques travaux illustrant l'importance de la phase d'analyse. Oswald et al. extraient la clé secrète d'un verrou numérique par analyse des émissions électro-magnétiques [Osw+13]. Le papier détaille clairement que les efforts les plus importants se sont portés sur la rétro-ingénierie du matériel, puis de l'implémentation logicielle du firmware. L'extraction du firmware, son analyse, puis son instrumentation pour identifier la fenêtre temporelle adéquate pour les mesures électromagnétiques sont décrits par Oswald et al. comme une étape essentielle avant la phase d'exploitation, qui elle est triviale : une attaque verticale par CPA est capable de retrouver la clé en 150 traces. Lomné et Roche ont publié une vulnérabilité

dans la clé matérielle d'authentification Google Titan [LR21]. La phase d'exploitation impliquait une cryptanalyse avancée de l'implémentation, mais Lomné et Roche détaillent toute la phase d'analyse de la cible, et soulignent ici aussi l'importance des efforts déployés pendant la phase d'analyse pour identifier une vulnérabilité.

Dans la phase d'exploitation, on considère comme un fait acquis la possibilité de faire l'acquisition de mesures physiques ciblées, sur une cible dont le principe de fonctionnement est connu (e.g., l'exécution du chiffrement AES), et la difficulté réside dans l'analyse des mesures physiques pour identifier un secret comme une clé de chiffrement.

Dans ce cadre, nous nous intéressons à la combinaison de deux contre-mesures, le chiffrement de code et le polymorphisme de code, pour couvrir un scénario d'attaque complet par canal auxiliaire. Le chiffrement de code apporte une protection contre la phase d'analyse, en empêchant la rétro-ingénierie statique de code après une extraction du firmware [Gou+23]. Le polymorphisme de code augmente la difficulté de réaliser la phase d'exploitation de l'attaque. En outre, les contre-mesures se protègent mutuellement, en comblant leur vulnérabilités respectives. Le chiffrement de code, seul, n'apporte pas de protection contre les attaques par canal auxiliaire, et similairement, le polymorphisme de code est vulnérable à la rétro-ingénierie. On peut noter toutefois que la rétro-ingénierie d'un programme polymorphe sera plus délicate que celle d'un programme statique habituel, puisque l'extraction statique de code donne seulement accès au code du générateur de code. Nous n'avons pas évalué la faisabilité de faire une rétro-ingénierie complète d'un composant logiciel protégé par polymorphisme de code, mais on peut raisonnablement faire l'hypothèse qu'elle est accessible à un attaquant à l'état de l'art. Si l'attaquant possède un accès logique à la cible, il lui est possible d'extraire le code d'une instance polymorphe une fois générée par son SGPC. Si l'attaquant possède seulement un accès physique à la cible, des techniques de rétro-ingénierie dynamique par l'analyse de mesures physiques seraient également envisageables [CLH19]. Ce point est d'ailleurs un angle mort de notre approche : le chiffrement de code apporte une protection efficace contre la phase d'analyse, mais nous n'avons pas évalué son efficacité contre les techniques dynamiques d'extraction et d'analyse de code.

Enfin, le chiffrement de code tel qu'il est mis en œuvre ici supporte naturellement tout agencement statique de code, donc en particulier toutes les contre-mesures à la phase d'exploitation qu'il est possible de mettre en œuvre avec des constructions statiques, comme le masquage. A contrario, le polymorphisme de code suppose, dans notre mise en œuvre, l'utilisation de la génération dynamique de code, dont l'articulation avec le chiffrement de code n'est pas triviale. Ce point constituait une raison supplémentaire pour étudier l'articulation du chiffrement de code et du polymorphisme de code.

3.1.2 Vue d'ensemble de PolEn

Les contre-mesures sont intégrées dans une chaîne de compilation et une architecture processeur dédiée, appelées PolEn [MCH21], basées sur l'architecture RISC-V RV32IM et sur la chaîne de compilation LLVM. On s'appuie sur le backend RISC-V de LLVM version 7. Ultérieurement à la publication dans DTRAP [MCH21], Étienne Louboutin a réalisé le portage de PolEn vers LLVM version 12. Le fonctionnement de la chaîne de compilation et du runtime sont illustrés en figure 3.1.

Les deux contre-mesures sont appliquées automatiquement lors de la compilation du programme à sécuriser. Le polymorphisme de code est d'abord appliqué sur les parties du programme sélectionnées par l'utilisateur. Comme dans le cas de Odo (section 2.2.1), un back-end dédié du compilateur, cette fois pour l'architecture RISC-V (`llvm_polen -poly`, figure 3.1), génère le code source des SGPC dans une implémentation en C. Un deuxième compilateur génère ensuite le code objet destiné à être chiffré (`llvm_polen -encrypt`, figure 3.1). Enfin, après l'édition des liens, un

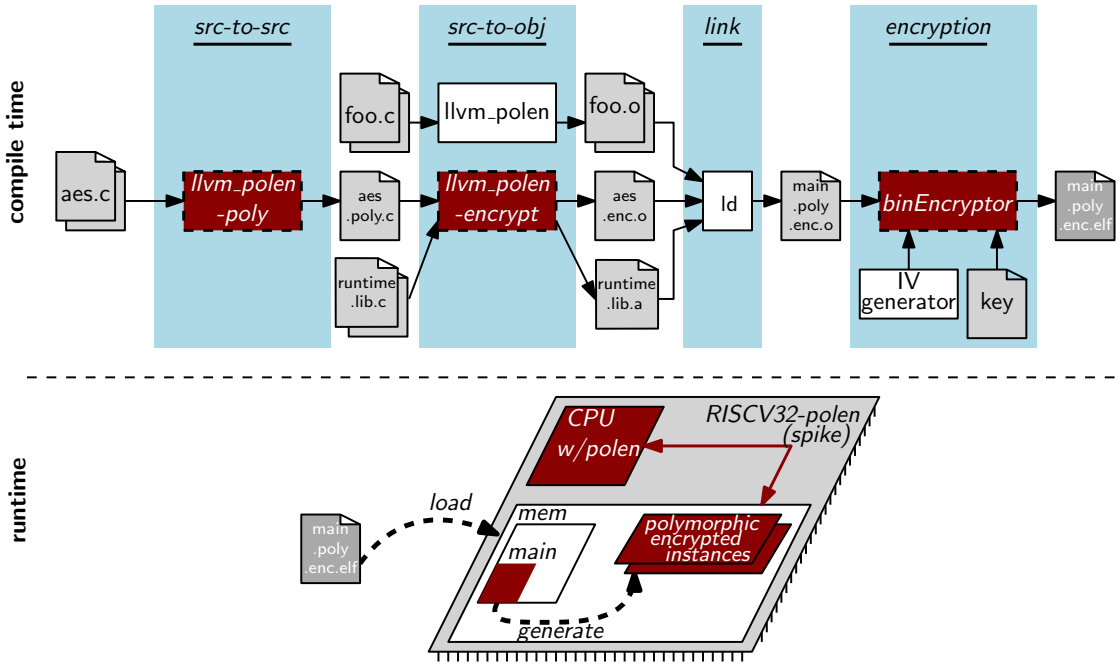


FIGURE 3.1 – Vue d'ensemble de POLEN : chaîne de compilation et exécution au runtime. Le fichier d'entrée `aes.c` représente la partie du programme à protéger, et le fichier `foo.c` représente le reste de l'application originale qui n'est pas protégé. Extrait de [MCH21].

dernier outil *ad hoc* applique le chiffrement de code : initialisation des vecteurs d'initialisation (IVs) et chiffrement des instructions des blocs de base concernés (`binEncryptor`, figure 3.1). Au runtime, le déchiffrement de code est opéré dans la micro-architecture du processeur immédiatement avant le décodage des instructions (section 3.1.3). La chaîne de compilation est conçue pour pouvoir appliquer indépendamment chaque contre-mesure, ce qui permet d'obtenir toutes les combinaisons possibles : polymorphisme de code seul, chiffrement de code seul, ou combinaison des deux contre-mesures. Enfin, lors de l'exécution d'un SGPC (que son code soit chiffré ou non), l'émission de code peut produire des instances polymorphes elle-même chiffrées (bas de la figure 3.1).

3.1.3 Chiffrement de code

La technique de chiffrement de code et d'exécution de code chiffré est issue des travaux de thèse de Thomas Hiscock [His17 ; HSG19]. L'implémentation originale ciblait un processeur MIPS 32 bits, et le travail dans ce cadre a consisté en un portage sur l'architecture RISC-V RV32IM : portage dans le simulateur Spike pour la partie matérielle, et portage dans le backend RISC-V de LLVM pour la partie logicielle.

La méthode de chiffrement du code repose sur un chiffrement par flot (*stream cipher*), et on utilise la primitive de chiffrement Trivium, qui permet des implémentations matérielles efficaces [DP08]. Au prix de quelques aménagements mineurs, il est également possible d'utiliser une primitive de chiffrement par bloc [MCH21, section 5.3].

Chaque branchement vers du code chiffré suppose qu'un vecteur d'initialisation (IV) est présent avant la première instruction du bloc de base cible, à l'adresse de la cible du branchement.

L'IV est utilisé pour initialiser la suite chiffrante du module de déchiffrement des instructions.

Concernant le support matériel, on ajoute :

— une instance de Trivium dans l'étage *fetch* du processeur pour le déchiffrement des mots d'instructions avant leur décodage (illustrée en Figure 3.2(a)) ;

— une instance de Trivium, pour le chiffrement de données avant leur écriture en mémoire (illustrée en Figure 3.2(b)). Cette instance, optionnelle, peut être placée dans l'étage *execute* ou dans l'étage *memory* du processeur suivant l'organisation de sa micro-architecture. Cette instance est utilisée en particulier pour l'émission d'instructions machine au runtime, dans les SGPC de programmes polymorphes.

— Quatre instructions dédiées à l'émission de code chiffré et au basculement entre l'exécution de code chiffré et l'exécution de code en clair [MCH21, Table 1].

En procédant ainsi, le code n'est jamais accessible en clair dans les mémoires du processeur, à l'exception des instructions générées par les SGPC qui sont accessibles en clair pendant quelques cycles dans les registres architecturaux du processeur avant leur émission sous une forme chiffrée en mémoire programme.

Concernant le support logiciel, plusieurs passes (détaillées dans [MCH21, section 3.3.2]) sont ajoutées dans le backend pour traiter les points suivants :

— La fusion de blocs de base pour éliminer certains branchements, donc réduire le nombre de réinitialisations de la suite chiffrante lors de l'exécution du code chiffré. Cette passe permet de réduire le surcoût de l'exécution de code chiffré.

— L'ajout de branchements vers un bloc de base *fallthrough* lorsque celui-ci possède plusieurs prédécesseurs. Cette passe permet l'exécution correcte du saut d'un bloc de base sans branchement de sortie vers un *fallthrough*.

— Enfin, une passe insère les emplacements en tête de blocs de base destinés à recevoir les valeurs d'IVs.

Ce travail ne traite pas de la génération, du partage et du stockage des clés pour le chiffrement et le déchiffrement du code, pour lesquels il est possible d'utiliser les techniques habituellement en usage, sans restriction ou contrainte particulière.

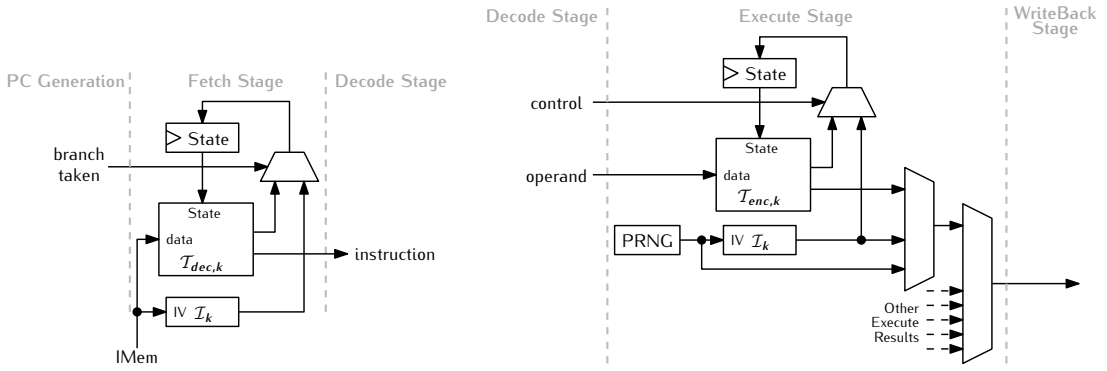
3.1.4 Polymorphisme de code

La mise en œuvre du polymorphisme de code est une application immédiate du travail réalisé dans Odo (section 2.2.1), dans un portage pour l'architecture RISC-V RV32IM. Dans l'objectif de diminuer l'effort du travail de portage, cette implémentation du polymorphisme de code supporte seulement l'insertion d'instructions factices (page 18). Cette mise en œuvre simplifiée nous permet cependant d'aborder tous les verrous liés à l'articulation du chiffrement de code avec le polymorphisme de code.

3.1.5 Exécution polymorphe de code chiffré

L'émission de code d'un SGPC diffère selon que le code des instances polymorphes est chiffré ou non. Dans les deux cas, le fonctionnement du SGPC reste le même, mais la nature des opérations effectuées est différente pour l'émission de code chiffré. Dans le cas où le code des instances polymorphes est chiffré, quelques aménagements sont apportés :

— lors de l'émission du code d'un nouveau bloc de base, une instruction dédiée permet d'écrire un IV en mémoire et d'initialiser l'état du module de chiffrement avec la même valeur d'IV ;



(a) Insertion d'une primitive de chiffrement par flot (State) dans l'étage *fetch* du pipeline du processeur pour le déchiffrement d'instructions.

(b) Insertion d'une primitive de chiffrement par flot (State) dans l'étage *execute* du processeur pour l'émission d'instructions machine chiffrées.

Figure 3.2 – Aménagements de la micro-architecture du processeur pour le support matériel de PolEn. Extrait de [MCH21].

— une instruction dédiée, sémantiquement similaire à une écriture en mémoire, permet l'écriture en mémoire d'un mot chiffré à partir d'un mot en clair stocké dans un registre du processeur.

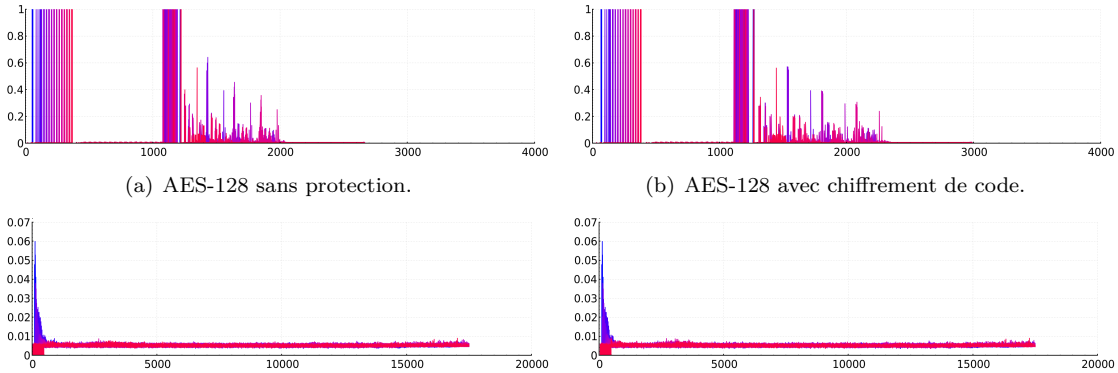
Les SGPC sont construits de sorte que l'analyse de code et les transformations pour la mise en œuvre du polymorphisme de code sont calculées en une seule passe, permettant l'émission directe du code machine sans passer par une représentation intermédiaire. Cette propriété est importante pour que la génération dynamique de code soit légère et rapide. Cependant, cette manière de procéder ne permet pas l'émission d'instructions de branchement en avant, pour lesquelles l'adresse cible n'est pas connue au moment du calcul de leur encodage. Dans ce cas, l'instruction de branchement est émise avec une adresse cible nulle, et l'encodage de l'instruction est finalisé une fois que la valeur de l'adresse cible est connue. Si l'instruction est chiffrée, cette manière de procéder est également possible en exploitant l'homomorphisme de Trivium au ou exclusif : $\mathrm{Enc}(\mathrm{insn} \oplus \mathrm{addr}) = \mathrm{Enc}(\mathrm{insn}) \oplus \mathrm{Enc}(\mathrm{addr})$.

3.1.6 Évaluation expérimentale

L'évaluation expérimentale est envisagée sous deux angles : le gain potentiel en sécurité, et l'impact sur les performances.

Évaluation en sécurité. On évalue l'exploitabilité de points d'intérêts potentiels sur des analyses verticales à l'aide de la *Normalized Inter-Class Variance* (NICV) [Bha+14], une métrique proche du SNR, dont la valeur est normalisée dans l'intervalle $[0; 1]$. Dans notre cas, la NICV est plus adaptée que le SNR habituellement utilisé puisque l'analyse est appliquée sur des traces obtenues par simulation, donc dépourvues de bruit.

Les résultats sont illustrés en figure 3.3. Ils confirment le fait que le chiffrement de code n'apporte aucune protection contre une analyse par canal auxiliaire qui exploite de la fuite d'information sur les données manipulées (Figure 3.3(b)), puisque les valeurs de NICV sont similaires à celles obtenues sur le code de référence sans protection. Le polymorphisme de code, même dans une version amoindrie basée sur la seule insertion d'instructions factices, permet



(a) AES-128 sans protection.

(b) AES-128 avec chiffrement de code.

(c) AES-128 avec polymorphisme de code.

(d) AES-128 avec exécution polymorphe de code chiffré.

FIGURE 3.3 – Analyse NICV sur 50000 traces simulées de la première ronde d'un chiffrement AES, pour tous les octets de clé, de l'octet 0 en bleu à l'octet 15 en rouge. Extrait de [MCH21].

bien de supprimer des points d'intérêt dans une analyse verticale (Figures 3.3(c) et 3.3(d)). Une analyse en CPA au premier ordre, qu'il n'est pas nécessaire de détailler ici, donne des conclusions similaires [MCH21, section 4.2.2 et Figure 10].

Impact sur les performances. Globalement, l'impact sur les performances est important, puisqu'il résulte des surcoûts additionnés du chiffrement de code et du polymorphisme de code. La figure 3.4 illustre l'impact en performance sur l'exécution des instances polymorphes. L'analyse est détaillée dans [MCH21], et en synthèse on peut retenir les points suivants :

Impact sur la taille de code. Le chiffrement de code avec Trivium implique l'insertion d'un IV de 80 bits, pour toute adresse cible d'un branchement vers du code chiffré. Pour notre cible, cela signifie l'ajout de 3 mots de 32 bits en début de chaque bloc de base (i.e. 80 bits et un bourrage de 16 bits pour conserver l'alignement sur des mots de 32 bits). Il est possible d'utiliser une autre primitive de chiffrement (section 3.1.7) pour modérer l'impact la taille des IV sur le surcoût en taille de code. En revanche, l'utilisation des IVs en tête de chaque bloc de base est indissociable, en l'état, de notre technique de chiffrement de code. Le nombre d'IVs étant lié au nombre de blocs de base dans le programme protégé, il est possible d'en réduire le nombre en utilisant des techniques plus agressives d'inlining de code ou de fusion de blocs, ces techniques ayant pour conséquence d'augmenter la taille de code.

Le polymorphisme de code implique les mêmes surcoûts que ceux de l'implémentation Odo (section 2.2.1) : l'ajout d'un générateur de code machine pour chaque SGPC, et l'augmentation de la taille des instances polymorphes en particulier pour l'utilisation d'insertion d'instructions factices.

Impact sur le temps d'exécution. Le chiffrement de code implique, lors de chaque branchement vers du code chiffré, la ré-initialisation de la suite chiffrante dans l'instance Trivium utilisée pour le déchiffrement de code. Pour les autres instructions, le chiffrement de code n'a pas d'impact sur le temps d'exécution. En d'autres termes, le surcoût du chiffrement de code sur le temps d'exécution est lié au nombre de branchements pris vers du code chiffré. La figure 3.4 illustre cette tendance, où le surcoût du temps d'exécution de code suit la même tendance que

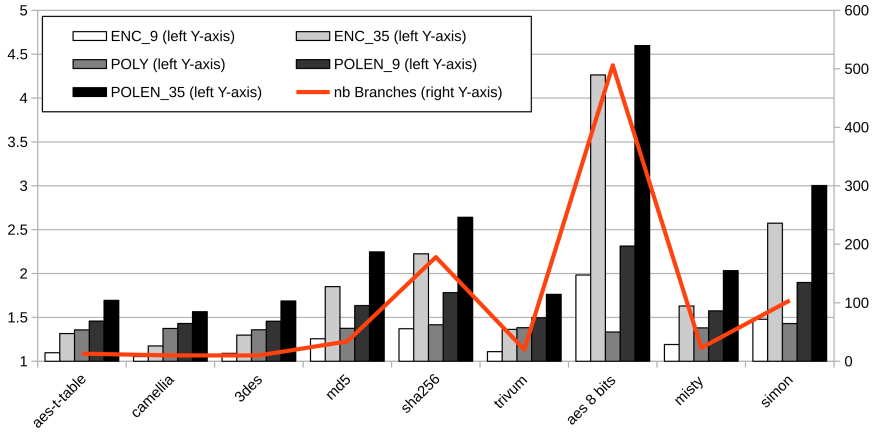


FIGURE 3.4 – Surcoût de POLEN à l'exécution de code chiffré (ENC), d'instances polymorphes (POLY), ou d'instances polymorphes chiffrées (POLEN), en comparaison au temps d'exécution du même programme sans protection, pour 9 benchmarks (en abscisse). La simulation considère deux durées différentes pour l'initialisation de Trivium : 9 cycles (e.g., ENC_9) et 35 cycles (e.g. ENC_35). La courbe en rouge mesure le nombre de branchements exécutés pour chaque benchmark. Extrait de [MCH21].

le nombre de branchements pris. L'utilisation du polymorphisme de code implique les mêmes surcoûts que ceux mesurés pour Odo (section 2.2.1) : la génération de code au runtime a un temps d'exécution qui est globalement linéaire par rapport au nombre d'instructions générées, et le coût de génération de chaque instruction augmente avec le nombre de transformations de code exploitées et de leur complexité. En outre, si les instances polymorphes sont chiffrées, la génération de code machine chiffré implique un temps supplémentaire pour l'initialisation de la suite chiffrante à chaque insertion d'IV. On peut noter que la suite chiffrante est ré-initialisée pour chaque IV *généré*, alors qu'à l'exécution de code chiffré le coût d'initialisation de la suite chiffrante est payé pour l'*exécution* de chaque branchement vers du code chiffré (i.e. vers un IV). En revanche, le temps d'écriture en mémoire d'une instruction machine chiffrée peut être identique à celui d'une instruction en clair si le bloc de chiffrement est suffisamment rapide (3.2(b)).

3.1.7 Ouverture

La question de l'articulation de contre-mesures ouvre un large champ potentiel pour leur application outillée. Nous avons montré dans le chapitre 2 comment le compilateur peut être exploité pour supporter un modèle de contre-mesure plus flexible, en élargissant l'espace de configuration et en apportant du support pour la mise en œuvre de toutes les combinaisons possibles. C'est le cas également ici, POLEN est supporté dans plusieurs configurations différentes : chiffrement de code seul ; polymorphisme de code seul ; polymorphisme de code avec chiffrement du code des SGPC mais génération d'instances chiffrement en clair ; polymorphisme de code avec génération d'instances polymorphes chiffrées. Ces configurations sont mises en œuvre automatiquement par la chaîne de compilation et d'exécution de code, à partir d'une déclaration choisie par l'utilisateur.

De manière similaire, l'implémentation matérielle du support d'exécution de code chiffré de POLEN supporte de nombreuses configurations possibles, qui auraient également pu faire l'objet

d'une exploration des impacts en performance et en sécurité. Par exemple, pour des implémentations matérielles très contraintes en surface, il est envisageable de mutualiser une partie des instances de Trivium pour le chiffrement (génération de code chiffré en mémoire) et pour le déchiffrement (exécution de code chiffré). Aussi, Trivium peut être implémenté de plusieurs manières, qui présentent des compromis différents entre la surface occupée et de temps d'exécution (notamment le temps d'initialisation de la suite chiffrante). Enfin, dans un modèle d'attaquant moins contraint, on peut envisager des implémentations logicielles de Trivium, voire la combinaison d'une implémentation matérielle (pour l'exécution de code chiffré) et d'une implémentation logicielle (pour la régénération occasionnelle de petites parties de programmes). Cette question de l'articulation de mécanismes matériels et logiciels est également sous-jacente à la conception de MAFIA, présentée ci-dessous.

3.2 Intégrité des signaux de contrôle d'un processeur

Principaux éléments contextuels de cet axe de recherche

Collaborations	Thomas Chamelot (CEA List), Karine Heydemann (LIP6)
Publications	[Cou17 ; CCH22a ; CCH22c ; CCH23]
Projets liés	Thèse de Thomas Chamelot [Cha22], projet COFFI

Nos premiers travaux sur la protection contre les attaques par injection de faute (section 2.1) illustrent la difficulté d'obtenir un niveau de protection suffisant lorsque l'attaquant a les moyens de perturber, par des injections précises, le fonctionnement du matériel sur lequel reposent les contre-mesures logicielles. Le logiciel a seulement accès à une vision architecturale du processeur (ISA), et n'a pas accès aux détails de son implémentation micro-architecturale. Cependant, une injection de faute est susceptible de perturber le fonctionnement micro-architectural d'un processeur, dont il est difficile de se prémunir par des moyens logiciels uniquement. Dans cette optique, la réalisation d'un schéma de protection logiciel dont la couverture est suffisante est peut-être impossible, et un tel schéma, si tant est qu'il doit possible de le concevoir, impliquerait un surcoût en temps d'exécution ou en empreinte mémoire prohibitifs. On fait donc l'hypothèse que la sécurisation contre un modèle d'attaquant puissant passe par la conception de solutions matérielles dédiées. C'est la pratique habituelle pour les produits de sécurité : depuis plusieurs dizaines d'années dans la carte à puce, et plus récemment dans les solutions de sécurisation pour les plateformes mobiles et pour l'informatique grand public. L'utilisation de solutions matérielles spécialisées a bien entendu un impact important sur tout le cycle de production de produits sécurisés : temps de développement, coût des produits. Cependant, ce coût est compensé par la grande taille des marchés adressés ; par exemple, les volumes de fabrication de carte à puce atteignent aujourd'hui encore plusieurs milliards d'unités par an.

Dans cet axe de recherche, on s'intéresse à améliorer la protection des programmes contre les injections de fautes en maîtrisant les surcoûts en performance liés à la sécurisation, en autorisant des modifications dans l'implémentation du matériel et en s'appuyant sur une meilleure articulation du logiciel et du matériel. L'objectif était également de fournir un schéma de protection qui couvre toute la micro-architecture du processeur ciblé, ce qui à notre connaissance n'existait pas dans l'état de l'art. Enfin, nous souhaitions fournir un prototype complet, incluant la partie matérielle, et toute la chaîne de compilation logicielle. Les travaux de développement de la solution ont été entrepris par Thomas Chamelot durant sa thèse, dans le cadre du projet COFFI. Le principe de fonctionnement, appelé SCI-FI [CCH22a], apporte l'intégrité ou l'authenticité du code, l'intégrité du flot de contrôle, et l'intégrité des signaux de contrôle de la micro-architecture du processeur. Les aménagements logiciels sont supportés par une chaîne de compilation dédiée.

Une version étendue, appelée MAFIA, reprend les principes de SCI-FI et ajoute le support de la prédiction de branchement, des interruptions et des branchements indirects [CCH23].

3.2.1 Un nouveau besoin : la protection de la micro-architecture

De nombreux travaux sur la caractérisation des fautes montrent qu'il est possible d'obtenir des effets variés sur l'exécution d'un programme. Concernant l'injection de fautes sur des processeurs, on modélise traditionnellement l'effet des fautes par des modèles de niveau ISA, par exemple : saut d'instructions [BGV11 ; Mor+13 ; Riv+15], rejeu d'une instruction ou modification de son opcode [BGV11], rejeu ou saut de paquets d'instructions [Riv+15].

Yuce et al. sont les premiers, à notre connaissance, à s'intéresser à l'impact des fautes sur le fonctionnement micro-architectural du processeur [Yuc+16]. La thèse de Johann Laurent étudie en profondeur l'impact des fautes dans la micro-architecture d'un processeur, au niveau RTL [Lau+19b ; Lau20 ; Lau+21]. Une faute (e.g., un bit flip) injectée sur des éléments ciblés de la micro-architecture peut permettre d'obtenir les modèles de fautes de niveau ISA déjà connus, mais surtout donner lieu à des effets inattendus qui sortent des modèles de fautes habituels. Aussi, ces travaux montrent que les contre-mesures de l'état de l'art ne couvrent pas ces nouveaux modèles de fautes, puisqu'elles ne couvrent pas l'intégrité de signaux internes à la micro-architecture des processeurs.

3.2.2 Modèle d'attaquant

On se place dans un modèle d'attaquant par injection de fautes typique [CCH23, Sec. III.A.]. On vise à détecter toute injection de fautes ciblant les signaux de contrôles dans la micro-architecture du processeur. Par extension, MAFIA peut détecter toute faute en dehors du processeur impactant le chemin d'instructions. Dans notre mise en œuvre, il est possible de détecter toute injection ciblant précisément au plus 8 bit flips, et une mise en œuvre de la fonction de signature reposant sur une primitive cryptographique peut augmenter la capacité de détection de MAFIA (cf. infra). MAFIA ne couvre pas les fautes sur le chemin de données du processeur (e.g., fautes sur les registres architecturaux ou certaines fautes dans la mémoire).

3.2.3 Principe de fonctionnement

MAFIA s'appuie sur deux principes de protection utilisés conjointement pour apporter une couverture complète de la protection des signaux de contrôle. Son fonctionnement s'appuie sur deux modules dont l'intégration à la micro-architecture du processeur est illustrée en figure 3.5. Une signature d'intégrité, mise à jour à chaque cycle du processeur, est calculée dans le module CACFI (*Code Authenticity and Control-Flow Integrity*) à partir de la valeur courante d'une sélection de signaux de contrôle, appelée *pipeline state*. L'utilisation d'une signature d'intégrité dont la valeur est propagée et mise à jour pendant l'exécution du programme protégé est une approche courante pour la détection des perturbations induites par les injections de fautes [OSM02 ; WWM15]. L'originalité de notre approche tient au fait de calculer la valeur courante de la signature d'intégrité à partir de la valeur d'une sélection de signaux de contrôle du processeur. L'intégration de la valeur de signaux de contrôle au calcul d'une signature d'intégrité a été utilisé dans le domaine de la sûreté de fonctionnement [KS01], mais cette mise en œuvre est totalement différente et ne couvrirait pas le modèle d'attaquant que nous visons.

Dans MAFIA, ce principe de signature permet d'assurer l'intégrité (CI) ou l'authenticité (CA) du code, l'intégrité du flot de contrôle (CFI), et l'intégrité des signaux de contrôle (CSI) en amont du pipeline state. En aval du pipeline state, l'intégrité des signaux de contrôle est assurée

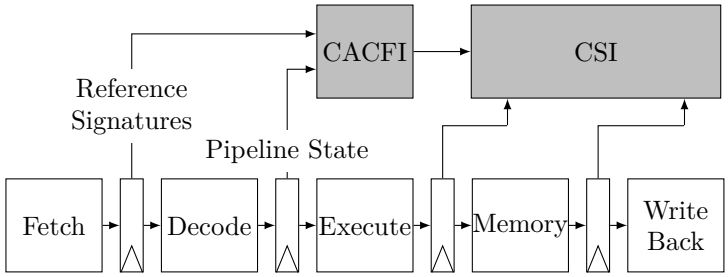


FIGURE 3.5 – Illustration d'un pipeline de processeur in-order à 5 étages étendu avec MAFIA (modules CACFI et CSI, en gris). Extrait de [CCH23].

par le module CSI (*Control Signal Integrity*), qui vérifie l'intégrité des signaux issus du pipeline state en comparant leur valeur courante à la valeur d'une copie qui a été vérifiée par le module CACFI.

Intégrité et authenticité du code (CI, CA). Une signature d'intégrité est calculée à chaque cycle du processeur par une *fonction de signature* à partir de la valeur courante du *pipeline state*. Le pipeline state est une sélection de signaux de contrôle émis par l'étage de decode du pipeline du processeur.

On distingue deux types de signaux de contrôle : les signaux *statiques*, dont la valeur dépend uniquement de l'instruction décodée ; les signaux *dynamiques*, dont la valeur dépend de l'instruction décodée et du contexte d'exécution, en raison de dépendances de données ou de dépendances avec les autres instructions en vol dans le pipeline du processeur. Le pipeline state permet d'assurer l'intégrité du code puisque la valeur des signaux de contrôle statiques émis par l'étage decode est déterministe, et qu'elle dépend de la valeur de l'instruction décodée. L'intégrité du code est assurée à la condition que la couverture du pipeline state soit suffisante. Nous y reviendrons en section 3.2.5.

Pour assurer une intégrité d'exécution complète, en dehors de l'intégrité des données qui n'est pas couverte par MAFIA (section 3.2.2), il est nécessaire de protéger aussi les signaux de contrôles dynamiques. Il est possible d'inclure dans le pipeline state des signaux dynamiques dont la valeur est fonction des dépendances entre instructions. Cela nécessite une analyse de dépendance entre instructions qui tient compte de l'implémentation de la micro-architecture du processeur, afin de pouvoir calculer leur valeur courante à chaque cycle du processeur. Par exemple, le contrôle du mécanisme de forwarding est un signal dynamique, dont la valeur est fonction des dépendances de données entre instructions [Lau+19a ; Tol+22c]. Ce point est détaillé page 51 en section 3.2.4. En revanche, il n'est pas possible d'inclure dans le pipeline state des signaux dynamiques dont la valeur dépend des données, comme des signaux de contrôle issus de la prédiction de branchement, puisque leur valeur n'est pas déterminable statiquement. Pour le support de la prédiction de branchement, nous avons proposé une solution basée sur un mécanisme de *rollback* [CCH23, Sec. IV.F.].

Le choix des signaux de contrôle inclus dans le pipeline state est un compromis entre le coût de l'implémentation et sa capacité à détecter des fautes (couverture). Concernant le coût, la taille de l'entrée de la fonction de signature est déterminée par la taille du pipeline state. Des fonctions cryptographiques légères supportent des entrées de 32 ou 64 bits. Des codes détecteurs d'erreur comme des CRC supportent des messages d'entrée de taille variable, mais dans le cas d'une implémentation matérielle la taille de l'entrée influe sur le coût de l'implémentation. La taille du pipeline state impacte aussi directement la taille du module CSI. Concernant la couverture,

la capacité de détection des fautes dépend de la fonction de signature, et du choix des signaux de contrôle inclus dans de pipeline state. Concernant la sélection des signaux de contrôle, on souhaite détecter toute faute injectée en amont du pipeline state, ce qui implique potentiellement d'inclure dans le pipeline state un grand nombre de signaux. Cependant, dans l'objectif de réduire l'impact en surface de la protection, il est souhaitable d'identifier la sélection minimale de signaux nécessaires au pipeline state. Dans le cadre de nos travaux, la sélection des signaux a été faite manuellement, par une analyse minutieuse du code RTL du processeur cible, mais l'implémentation résultante a été vérifiée formellement (section 3.2.5).

Le choix de la fonction de signature est déterminé par les objectifs de sécurité, et doit satisfaire un certain nombre de critères [WWM15], [CCH23, Sec. IV.C.]. Un code détecteur d'erreur ayant une capacité de détection suffisante sera un candidat léger pour l'intégrité du code (CI). Une fonction de signature cryptographique à clé secrète peut en outre assurer l'authenticité du code (CA). Nous présentons dans la section 3.2.4 deux implémentations de MAFIA : avec une fonction CRC et avec une fonction CBC-MAC basée sur la primitive cryptographique Prince.

Intégrité du flot de contrôle (CFI). Dans un modèle d'attaquant typique de la sécurité logicielle, on considère que l'attaquant peut contrôler les données (e.g., par l'exploitation d'un *buffer overflow*) mais qu'il ne peut pas modifier les instructions du programme, parce que la mémoire programme n'est pas, en général, accessible en écriture. Dans ce contexte, les techniques de CFI protègent uniquement les branchements indirects [Aba+09 ; Bur+17], puisque ce sont les seuls instruction de flot de contrôle dont la cible peut être contrôlée par un attaquant pouvant altérer les données du programme. Aussi, les blocs de base d'un programme ne sont pas protégés par de telles contre-mesures.

Dans un modèle d'attaquant qui inclut la possibilité d'injection de fautes, il est nécessaire en revanche de protéger toutes les instructions du programme, puisqu'une faute peut être exploitée pour sauter ou rejouer des instructions [Mor+13 ; Riv+15]. Dans ce contexte, l'intégrité du flot de contrôle doit donc couvrir toutes les instructions du programme, y compris l'exécution d'une séquence d'instructions dépourvue de branchements, comme dans un bloc de base. La plupart des solutions de CFI articulent deux mécanismes de protection [De +17 ; Dan+18] : l'un pour le flot de contrôle dans les blocs de base, l'autre pour le flot de contrôle entre blocs de base et entre fonctions. Cette manière de procéder a pour inconvénient de nécessiter la vérification de la signature à la fin de chaque bloc de base. Sachant qu'un bloc de base a une taille moyenne relativement faible (quelques instructions seulement pour certains programmes), le coût de la protection peut devenir important.

Notre approche s'inspire de la méthode GPSA (*Generalized Path Signature Analysis*) [WS90], également employée par Werner et al. [WWM15 ; Wer+18]. La méthode consiste à propager une valeur de signature tout le long du chemin d'exécution du programme. En cas de divergence des chemins d'exécution, chaque chemin propage une valeur de signature différente en fonction des instructions rencontrées. Aux points de jonction des chemins d'exécution, des *patches* sont combinés à la signature courante à l'aide d'une *fonction de patch*, de sorte qu'une seule valeur de signature ne soit possible après la jointure. Si la fonction de signature a la propriété de préserver les erreurs (i.e., à la suite d'une injection de faute, la signature courante ne peut pas retrouver la valeur correspondant à l'exécution de la même séquence d'instructions sans faute), il devient possible de vérifier la valeur de signature en n'importe quel point du programme, par exemple une seule fois en fin de programme. La nombre de vérifications opérées devient alors un compromis entre le surcoût engendré par les vérifications et le délai de détection entre les points potentiels d'injection de fautes et les points de vérification.

Dans les travaux antérieurs basés sur GPSA [WS90 ; WWM15 ; Wer+18], la valeur de la signature dépend de l'encodage machine des instructions exécutées. Dans MAFIA, la valeur de

la signature dépend de la valeur du pipeline state, donc de la valeur de signaux de contrôles, et en particulier de signaux dynamiques. Il est possible que la valeur d'un signal dynamique dépende du chemin d'exécution pris : aux jointures de chemins d'exécution, chaque instruction précédant la jointure est différente, et la valeur de signaux dynamiques dépendant des dépendances entre instructions peut donc elle aussi varier. Étant donné que la valeur de la signature d'intégrité est strictement déterminée par la valeur du pipeline state, il est nécessaire que la valeur du pipeline state soit unique pour chaque instruction du programme. Nous appelons cette propriété *unicité du pipeline state*. En pratique, dans notre implémentation de MAFIA, elle est appliquée par des aménagements logiciels pendant la compilation (section 3.2.4).

Dans l'approche initiale de Wilken et al. [WS90], l'intégrité d'exécution sur les blocs de base est complétée par l'association de tags d'intégrité à chaque instruction, appelés *horizontal signatures*, dans le but de réduire le délai de détection, mais ces tags permettent aussi de renforcer le CFI en détectant par exemple les inversions d'instructions. Dans notre approche, on assure l'ordre d'exécution des instructions sans surcoût en s'assurant que la fonction de signature n'est pas associative [WWM15], [CCH23, Sec. IV.C.], puisque dans ce cas l'exécution d'instructions dans le désordre donne des signatures de valeurs différentes.

Intégrité du flot de contrôle indirect. Enfin, MAFIA propose une solution permettant de réduire la surface d'attaque liée aux branchements indirects. Dans le contexte des attaques par injection de fautes, le flot de contrôle indirect nécessite un traitement particulier, qui fait appel à des techniques de protections différentes de la protection du flot de contrôle direct présentée ci-dessus.

La protection des branchements indirects (autres que les retours de fonctions) passe par la protection des données impliquées dans le calcul de l'adresse cible. Or, MAFIA ne protège pas le chemin de données du processeur. Dans l'état de l'art des protections CFI contre les attaques par injection de fautes, la protection des branchements indirects est rarement traitée, et quand elle est traitée, la solution passe par l'utilisation d'un point fixe. Par exemple, Werner et al. utilisent un état cryptographique constant (équivalent à la signature utilisée dans MAFIA) accessible par tout branchement indirect [Wer+18]. Dans ce cas, une modification du flot de contrôle est donc possible permettant de passer de toute source d'un branchement indirect à toute destination d'un branchement indirect. Certaines solutions font appel à des protections plus précises, autorisant uniquement le branchement entre une source et une destination précises, e.g. [SEH20], mais ces solutions supposent de connaître précisément toutes les cibles de branchement indirect. Cette analyse dépend des données manipulées par le programme. Il s'agit d'un problème d'analyse difficile à traiter, et en pratique on fait appel à des sur-approximations du graphe d'appels [Bur+17], ce qui rend inopérantes les solutions de protection les plus précises.

La solution mise en œuvre dans MAFIA est la suivante :

— Chaque appel de fonction indirect est remplacé par un appel direct vers un trampoline (*dispatcher*), qui est une une séquence d'instructions faisant uniquement appel à des branchements et des appels de fonctions directs, et permettant de diriger le flot de contrôle vers la ou les cibles de l'appel indirect. Un trampoline vérifie d'abord l'adresse cible en la comparant successivement à toutes les cibles d'appels autorisées. Dès que l'adresse cible est trouvée parmi les adresses autorisées, le trampoline renvoie le flot d'exécution vers la fonction cible à l'aide d'un appel de fonction direct. Par ailleurs, chaque branchement indirect est remplacé par un branchement direct, à l'aide d'une option dédiée du compilateur. L'utilisation exclusive d'appels et de branchements directs permet de prolonger le schéma d'intégrité à base de signatures de MAFIA dans les trampolines, ce qui permet d'avoir une couverture de la contre-mesure sans discontinuités.

— La liste des cibles d'appels de fonctions autorisées dans les trampolines est construite pour chaque site d'appel en faisant une analyse de *classes d'équivalence* [Bur+17]. Chaque classe d'équivalence contient la liste de toutes les fonctions appelables ayant le même prototype. On considère que chaque appel de fonction indirect peut uniquement cibler une fonction ayant un prototype identique au prototype visé par le site d'appel. Chaque classe d'équivalence est construite de manière conservative (i.e., elle peut contenir des fonctions qui ne sont jamais effectivement appelées), mais il est possible de faire cette analyse statiquement, lors de la compilation, sans avoir recours à des techniques d'analyses plus avancées et plus coûteuses. La signature d'intégrité étant propagée tout au long de l'exécution des trampolines, toute déviation du flot de contrôle est détectable si elle ne cible pas une fonction de la même classe d'équivalence, ce qui limite considérablement les opportunités de déviation du flot de contrôle. En pratique, dans les benchmarks étudiés, les classes d'équivalence contiennent au plus 9 fonctions [CCH23, Table I].

— On fait l'hypothèse qu'une *shadow stack* matérielle, e.g. [Ozd+06], est implémentée dans le processeur pour la protection des retours des appels de fonctions (*backward edges*). La shadow stack n'est pas implémentée dans l'évaluation ci-dessous. Il serait également possible d'étendre la solution de trampolines à la protection des retours de fonctions, mais ces trampolines seraient probablement très volumineux puisque tous les retours de fonctions appartiennent à la même classe de fonctions. Ces trampolines induiraient également un temps d'exécution important pour chaque retour de fonction.

Intégrité des signaux contrôle (CSI). Le module CSI vérifie que les signaux du pipeline state sont propagés correctement jusqu'à leur point d'utilisation dans les étages suivants du pipeline. Les signaux du pipeline state sont dupliqués dans le module CSI à la sortie de l'étage decode. Ensuite, le module CSI compare les signaux originaux aux signaux dupliqués entre chaque étage. Ainsi, le module CSI peut détecter n'importe quelle faute qui modifierait un signal du pipeline state après l'étage décode. L'intégrité de l'ensemble du chemin d'exécution est assurée par la combinaison des modules CACFI et CSI : le module CACFI assure l'intégrité du pipeline state et le module CSI assure l'intégrité de la propagation des signaux dans la micro-architecture.

3.2.4 Implémentation

Thomas Chamelot a réalisé durant sa thèse une implémentation prototype de MAFIA. La partie matérielle, illustrée en figure 3.5, est intégrée au processeur RISC-V CV32E40P, un processeur 32 bits composé d'un pipeline à 4 étages qui implémente le jeu d'instructions RV32IMC. La partie logicielle, illustrée en figure 3.6, est typique des systèmes embarqués contraints. Elle s'articule autour de la chaîne de compilation LLVM et de la librairie C Newlib.

Impact sur l'implémentation matérielle. L'intégration de MAFIA dans le processeur CV32E40P consiste en deux modifications majeures : l'implémentation des modules CACFI et CSI, et l'extension du jeu d'instructions (ISA).

Deux implémentations de MAFIA ont été réalisées avec les fonctions de signature suivantes :

— une fonction CRC, assurant CI, dont l'implémentation est légère et capable de détecter jusqu'à 8 bit flips dans le pipeline state ;

— une fonction CBC-MAC encapsulant la primitive Prince, assurant CA et capable de détecter toute altération du pipeline state.

Le jeu d'instructions RV32IMC est étendu comme il suit :

— Une variante de chaque instruction de flot de contrôle est créée, avec pour effet supplémentaire de déclencher une vérification de la valeur de signature courante avant de traiter l'opération sur le flot de contrôle. Ces instructions attendent une valeur de référence de la signature, placée dans la mémoire programme à la suite de l'instruction. Chaque instruction de vérification insérée dans le programme induit donc un surcoût d'un mot de la taille de la signature, c'est-à-dire 32 ou 64 bits, respectivement pour CRC et CBC-MAC.

— Une nouvelle instruction, `MAFIA.ldp`, permet d'injecter une valeur de patch dans la signature courante. Lors de son exécution, cette instruction charge une valeur de patch depuis une section mémoire dédiée (`.patches`) et la place dans un registre dédié. Lors de l'exécution de l'instruction de flot de contrôle suivante, une fonction de patch est appliquée à la signature de courante, avec pour paramètre la valeur du registre de patch. Après chaque branchement, le registre de patch est réinitialisé avec l'élément neutre de la fonction de patch, ce qui permet d'appliquer la fonction de patch systématiquement, lors de l'exécution de chaque instruction de flot de contrôle. Chaque instruction `MAFIA.ldp` insérée dans le programme induit un overhead de deux mots mémoire : un pour l'instruction, et un pour la valeur de patch.

Impact sur la chaîne logicielle. Le support de MAFIA implique des aménagements dans la chaîne de compilation, qui est illustrée en figure 3.6.

L'originalité de notre approche vient du fait d'exploiter la valeur de signaux de contrôle dans le calcul d'une signature d'intégrité, que nous exploitons en outre pour assurer CA ou CI et CFI. Dans notre implémentation, le seul signal de contrôle dynamique intégré au pipeline state est un signal de forwarding, sur 4 bits, calculé dans l'étage de decode.

— Afin de préserver l'unicité du pipeline state, lors de la compilation des fichiers objets (*clang*, figure 3.6), une analyse de dépendance vérifie que la valeur des signaux dynamiques est constante aux jointures de chemins d'exécution. Si ce n'est pas le cas, des instructions sans effet sont ajoutées (e.g., `nop`) pour casser la dépendance dynamique.

— Le calcul des signatures de référence nécessite un modèle micro-architectural permettant de calculer la valeur des signaux exploités dans le pipeline state en fonction des instructions exécutées (*Signature Generator*, figure 3.6). Dans notre implémentation, un modèle de l'étage decode du processeur CV32E40P suffit.

L'insertion d'instructions pour le placement des patches nécessite une analyse du graphe de flot de contrôle. À chaque jointure de chemin d'exécution, des patches sont appliqués sur tous les chemins sauf un.

Enfin, le support du flot de contrôle indirect est traité de la façon suivante. Lors de la compilation (*clang*, figure 3.6), les appels de fonctions indirects sont remplacés par un appel direct à un trampoline, et le code des trampolines est inséré ensuite par un outil dédié (*Dispatcher Generator*, figure 3.6). Les branchements indirects sont remplacés par des branchements directs par une option dédiée du compilateur clang ; cette solution permet de réduire le nombre de trampolines utilisés.

Estimation de l'impact sur les performances. L'évaluation du coût en surface est réalisée à l'aide d'une synthèse ASIC, dans la technologie GF-22FDX FDSOI à 400 MHz, illustrée dans le tableau 3.1. On se compare par rapport à la synthèse du cœur CV32E40P seul, réalisée dans les mêmes conditions. Dans notre implémentation du module CSI, les signaux de contrôle sont protégés par copie simple, ce qui résulte en une implémentation légère, dont l'impact global sur la surface est faible. La mise en œuvre de schémas de redondance apportant un meilleur niveau

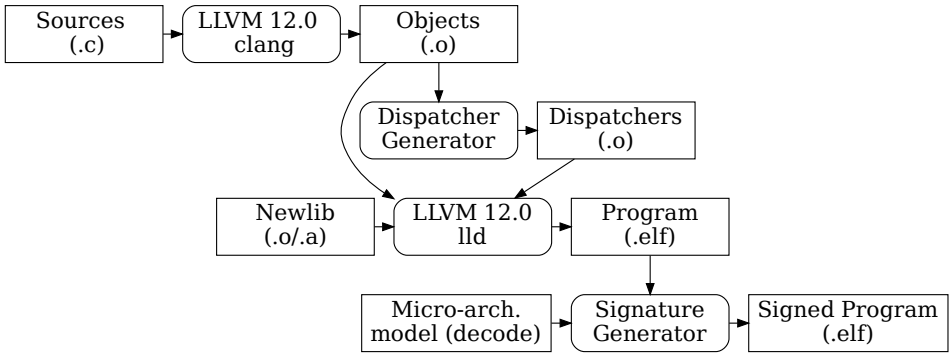


FIGURE 3.6 – Chaîne de compilation typique des systèmes embarqués, étendue avec le support de MAFIA. Adapté de [CCH23].

TABLEAU 3.1 – Évaluation du surcoût matériel de MAFIA, en GF-22FDX FDSOI à 400 MHz. Extrait de [CCH23].

Implémentation	surface (kGE)	surcoût (%)
Circuit de référence – CV32E40P	51,8	–
MAFIA – CRC32	55,2	6,5
MAFIA – CBC-MAC	64,2	23,8

de résistance aux injections de fautes est possible, sans impact sur le reste de l'implémentation matérielle. Le module CACFI intègre la fonction de signature, qui représente la majeure partie du surcoût matériel. L'implémentation CRC32 de MAFIA induit un surcoût de 3,4 kGE par rapport à l'implémentation de référence, et l'implémentation CBC-MAC/Prince induit un surcoût de 12,4 kGE. Relativement à la surface du cœur de référence, le surcoût peut sembler significatif (6,5 et 23,8 % respectivement), mais il faut noter qu'un système complet aurait une surface plus importante en raison de l'ajout de mémoires et de périphériques. Sur ces composants périphériques au processeur, MAFIA n'induit aucun surcoût supplémentaire et le surcoût relatif à la surface de l'implémentation de référence serait donc plus faible que celle mesurée dans notre évaluation.

Le tableau 3.2 donne les résultats des mesures de performance menées sur la suite de benchmarks Embech-IoT, pour l'implémentation de MAFIA basée sur la fonction de signature CRC. Ici, les instructions de vérification sont utilisées systématiquement, pour chaque instruction de flot de contrôle, dans la fonction principale de chaque benchmark. En revanche, le reste du code ainsi que la Newlib C sont seulement instrumentées avec des instructions de patches pour assurer la continuité de la signature. Aussi, pour donner une estimation précise du surcoût en taille de code, tout le code inutile est supprimé à la compilation [Cha22, Sec. 4.6.3]. Enfin, 3 des 19 benchmarks évalués utilisent des appels indirects de fonctions (`picojpeg`, `sglib-combined`, `wikisort`).

La suite Embench-IoT présente un panel de programmes représentatifs d'applicatifs exécutés sur des systèmes embarqués dépourvus de systèmes d'exploitation. Les profils d'applications sont variés, avec des temps d'exécution pouvant dépasser plusieurs dizaines de millions de cycles processeur. Les surcoûts observés sont variables : de 7 % à 47 % (moyenne 30 %) pour la taille de code, et de 3 % à 49 % (moyenne 19 %) pour le temps d'exécution. Pour réduire le surcoût logiciel, il serait possible d'utiliser une seule instruction de vérification par benchmark ; par

Tableau 3.2 – Mesures de performances à partir de la suite Embench-IoT pour les niveaux de compilation `O2` et `Os`, pour l'implémentation de MAFIA avec CRC32 : taille de code (en octets, et facteur de surcoût par rapport à la version non protégée), nombre de signatures et de patchs insérés, temps d'exécution (en cycles CPU, et facteur de surcoût par rapport à la version non protégée). Les surcoûts en taille de code et en temps d'exécution sont mesurés par rapport au même programme, compilé avec le même niveau de compilation, sans le support de MAFIA. Extrait de [CCH23].

Programme	O2				Os			
	Taille	Signatures	Patches	Temps d'exéc.	Taille	Signatures	Patches	Temps d'exéc.
aha-mont64	6204 (×1.30)	124	106	75335 (×1.38)	4720 (×1.28)	92	70	67461 (×1.18)
crc32	824 (×1.34)	8	21	380997 (×1.21)	856 (×1.35)	9	21	381006 (×1.21)
cubic	97460 (×1.16)	98	1611	14787617 (×1.16)	96920 (×1.16)	95	1608	14802517 (×1.16)
edn	3564 (×1.29)	53	63	1157814 (×1.22)	3944 (×1.25)	53	64	1157585 (×1.22)
huffbench	4028 (×1.39)	90	90	382404 (×1.20)	3548 (×1.33)	61	73	383163 (×1.16)
matmult-int	3376 (×1.26)	29	70	1119255 (×1.21)	2588 (×1.23)	10	54	1200057 (×1.21)
minver	21472 (×1.24)	56	489	151685 (×1.18)	21404 (×1.24)	59	474	178843 (×1.17)
nbody	20576 (×1.20)	52	404	55767175 (×1.18)	19944 (×1.20)	42	387	55780098 (×1.18)
nettle-aes	5600 (×1.14)	46	63	136279 (×1.10)	5512 (×1.14)	44	59	136447 (×1.10)
nettle-sha256	7864 (×1.07)	39	44	9573 (×1.03)	7720 (×1.08)	46	46	10239 (×1.03)
nsichneu	25204 (×1.45)	654	565	3693 (×1.44)	25152 (×1.45)	648	546	3685 (×1.44)
qrduino	21840 (×1.39)	554	455	1605197 (×1.18)	18304 (×1.37)	448	357	1610103 (×1.16)
slre	7604 (×1.55)	242	211	0	6760 (×1.52)	214	168	45480 (×1.17)
st	21964 (×1.20)	58	420	6618952 (×1.17)	21764 (×1.19)	44	409	6626187 (×1.17)
statemate	8956 (×1.29)	203	141	1573 (×1.09)	9116 (×1.28)	198	138	1672 (×1.08)
ud	3456 (×1.25)	39	62	23674 (×1.16)	3232 (×1.27)	37	62	24125 (×1.16)
picojpeg	31116 (×1.47)	881	665	2403701 (×1.21)	22548 (×1.49)	642	485	2424312 (×1.16)
sglib-combined	8332 (×1.47)	197	219	409637 (×1.18)	8736 (×1.46)	206	214	466871 (×1.31)
wikisort	32340 (×1.30)	354	647	28465027 (×1.38)	31956 (×1.29)	318	625	24817089 (×1.20)
Moyennes	**17462 (×1.30)**	**99**	**334**	**5973662 (×1.20)**	**16564 (×1.29)**	**86**	**308**	**5795628 (×1.18)**

exemple, l'utilisation d'une seule vérification (signature) pour `nsichneu` permettrait de réduire les surcoûts en taille de code et en temps d'exécution à environ 12 % et 17 % respectivement. Les deux seules applications de cryptographie, `nettle-aes` et `nettle-sha256`, montrent un profil de performance intéressant : le surcoût en taille et en temps d'exécution sont modérés, (au plus, respectivement 14 % et 8 %) et sont inférieurs aux moyennes sur l'ensemble des benchmarks.

3.2.5 Analyse de sécurité

Le pipeline state est une des pierres angulaires de la protection apportée par MAFIA. Dans l'implémentation actuelle, les signaux de contrôle inclus dans le pipeline state sont sélectionnés manuellement, en visant le meilleur compromis entre couverture et coût (section 3.2.3, page 47). La construction du pipeline state est vérifiée formellement par l'approche développée dans la thèse de Simon Tollec [Tol+22c] : on s'assure que toute faute injectée en amont du pipeline state est capturée par une modification de la valeur courante du pipeline state. L'analyse de sécurité de MAFIA montre que la fonction de signature est capable de détecter toute modification de valeur du pipeline state (dans le cadre du modèle d'attaquant défini) [CCH23, Sec. VI.A], ce qui apporte la garantie d'une couverture suffisante du pipeline state.

Les fonctions CRC donnent lieu à des implémentations légères, mais leur capacité de détection des erreurs dépend de leur polynôme générateur. Thomas Chamelot a réalisé une étude exhaustive de la capacité de détection de fonctions CRC32 connues pour leur capacité de détection [Koo02]. Il s'agit de déterminer le poids de Hamming minimum des vecteurs de fautes par inversions de bits susceptibles de créer des collisions sur la signature, en considérant des séquences de 1 à 30 valeurs de pipeline state, qui correspondraient dans notre cas d'usage à des séquences de 1 à 30 instructions contiguës. Par simulation numérique, on évalue que le CRC32 0xFA567D89 permet

de détecter 8 inversions de bit pour des séquences allant de 1 à 30 pipelines states, ce qui permet de se protéger contre un attaquant capable de réaliser des injections de fautes multiples sur 8 bits.

Enfin, une analyse théorique de sécurité [Cha22 ; CCH23] permet d'assurer aussi que la conception de MAFIA est robuste à des attaques ciblant la fonction de signature CBC-MAC/Prince, l'application de valeurs de patches, ou la vérification des signatures.

3.2.6 Ouverture

Intégrité des données. Dans le paysage des contre-mesures de processeurs contre les attaques par injection de fautes, MAFIA apporte une solution pour la protection des signaux de contrôle de la micro-architecture, articulée avec l'authenticité ou l'intégrité du code (CA/CI) et l'intégrité du flot de contrôle (CFI). Pour apporter une couverture complète contre les injections de fautes, il est nécessaire de combiner cette approche avec l'intégrité des données (DI). À notre connaissance, il n'existe pas, dans l'état de l'art, d'approches proposant une protection combinée des données et du code. En revanche, Stefan Mangard et son équipe ont fait une étude systématique des solutions de sécurisation des systèmes embarqués dans des modèles d'attaquants qui incluent les attaques matérielles, et traitent en particulier l'impact des injections de fautes sur le chemin de données. Ces travaux couvrent plusieurs problèmes de sécurité embarquée et nous ne citons ici que ceux couvrant les attaques par injection de fautes. SCB [SWM18] propose une protection des branchements conditionnels qui protège aussi les valeurs d'opérandes des instructions de branchement. SecWalk [Sch+18 ; Sch+21] propose une protection de la mémoire virtuelle contre les injections de fautes. Par extension, SecWalk apporte une protection des pointeurs et de l'arithmétique de pointeurs, mais ne protège pas les autres données ou les autres opérations effectuées dans le processeur. SCFI [Nas+23] propose une protection du flot de contrôle des implémentations matérielles de machines à états, couvrant à la fois la logique de contrôle et les registres d'état.

On note enfin que le processeur CV32E40S [Gro23], basé sur le même processeur CV32E40P que celui utilisé pour le prototype de MAFIA, propose un jeu de contre-mesures contre les attaques par canaux cachés et par injection de fautes. Ces contre-mesures sont également exploitées dans le cœur Ibex utilisé dans OpenTitan [Joh+18]. Ces contre-mesures couvrent une partie du chemin de données (e.g., ECC sur le banc de registres) et peuvent limiter l'exploitation d'attaques sur le flot de contrôle (détection d'incohérences entre les états du registre PC dans les étages de fetch et de decode). Cependant, il n'existe pas d'analyse publique de la sécurité apportée par ces contre-mesures, et on peut supposer que leur couverture reste insuffisante contre un attaquant à l'état de l'art.

Une autre approche intéressante de protection des données consiste à dupliquer les calculs. La protection peut être implémentée par logiciel [Rei+05 ; Bar+10]. Manssour et al. présentent un support matériel pour le rejeu d'instructions, toujours dans le but de assurer l'intégrité des données [Man+22]. Une combinaison avec MAFIA serait en mesure d'apporter une couverture complète du processeur.

Chiffrement du programme. Le chiffrement est un moyen indirect d'empêcher l'exploitation de fautes dans la hiérarchie mémoire [UWM19 ; SEH20 ; Nas+21]. En présence de chiffrement, l'effet de l'injection est impossible à caractériser car l'attaquant n'a pas la connaissance des instructions exécutées. Cette technique permet de se passer d'un mécanisme de détection et d'alerte, car une faute injectée sur une instruction chiffrée peut donner un mot d'instruction qui n'est pas décodable, ou bien entraîner l'exécution d'une séquence d'instructions décodables mais menant rapidement à un état de programme illicite, le processeur levant alors dans chaque cas

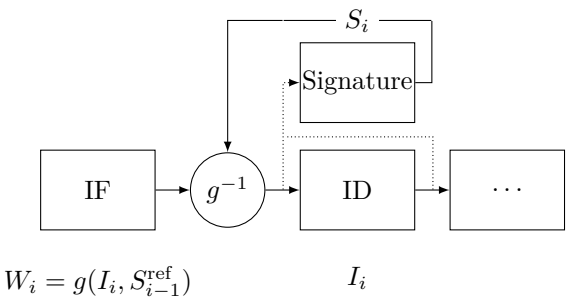


FIGURE 3.7 – Combinaison de la confidentialité du code avec la protection des signaux de contrôle par calcul d'une signature d'intégrité. La fonction g est la primitive de chiffrement, pour produire les mots d'instructions chiffrés W_i à partir des instructions en clair I_i et des signatures de référence S_{i-1}^{ref}. Figure adaptée de [CCH22c].

une exception. Dans la même veine, SBCF [Wer+18] propose une solution de chiffrement des instructions machine d'un programme, mais cette fois les instructions sont déchiffrées juste avant leur décodage. Cette solution apporte indirectement une protection du flot de contrôle, y compris intra-bloc de base, contre les injections de fautes, mais ne protège ni les données ni les signaux de contrôle dans la micro-architecture.

L'inconvénient majeur de l'utilisation du chiffrement des instructions machine comme protection contre les injections de fautes est la difficulté de caractériser le délai de détection de la faute, qui dépend de la densité du jeu d'instructions utilisé, c'est-à-dire de la probabilité que la faute entraîne le déchiffrement d'un mot d'instruction non décodable. Cet inconvénient est à mettre en regard des avantages apportés : outre la confidentialité du code, le chiffrement permet de réduire le surcoût lié à l'utilisation de vérifications explicites. En effet, avec les techniques d'intégrité, même dans les cas où la vérification n'est pas déclenchée explicitement par des instructions ajoutées au programme protégé, les valeurs d'intégrité de référence doivent être déclarées [WS90 ; WWM15], ce qui induit un surcoût non négligeable sur l'utilisation de la mémoire. A contrario, si les mots d'instruction chiffrés ont la même taille que les mots d'instruction en clair, le surcoût sur la taille du programme est nul.

À la fin de la thèse de Thomas Chamelot, nous avons proposé un mécanisme permettant de combiner le chiffrement des instructions avec un mécanisme de signature protégeant la valeur des signaux de contrôle tel que le module CACFI de MAFIA [CCH22c]. Le schéma de prinicipe est illustré en figure 3.7. L'idée est d'exploiter la valeur de la signature de référence précédente dans le déchiffrement de l'instruction courante. Dans la mise en œuvre illustrée ici, la fonction de signature est calculée dans l'étage de décode (ID), ce qui peut imposer des contraintes fortes sur le chemin critique du processeur. Il est possible de retarder le calcul de la fonction de signature, e.g. dans l'étage suivant le decode, mais cela implique de déchiffrer l'instruction courante avec la signature de référence associée à l'instruction $n-2$: on relâche les contraintes sur le chemin critique du processeur, mais on augmente la complexité de la génération de code.

Protection des signaux de contrôle dans des architectures plus complexes. MAFIA a été prototypée pour une micro-architecture simple avec un pipeline in-order à 4 étages. Cette micro-architecture soulève déjà des problèmes intéressants, dont la protection de signaux de contrôle dynamiques comme le forwarding. Étant donné que MAFIA demande une intégration minutieuse à la micro-architecture, on peut supposer que son application à des micro-architectures

plus complexes demanderait un effort important.

Concernant les micro-architectures in-order, le point d'intégration le plus difficile est probablement la protection des signaux de contrôle dynamiques. Les signaux dynamiques qui dépendent des données ne peuvent pas être inclus dans le pipeline state (section 3.2.3) et doivent donc être protégés par d'autres stratégies, par exemple avec de la redondance dans le module CSI et en protégeant également la logique de calcul de la valeur de ces signaux. Les signaux dynamiques qui ne dépendent pas des données, comme le forwarding, peuvent être inclus dans le pipeline state, mais la logique de calcul du forwarding (*Forwarding Unit*) est souvent placée dans l'étage execute du processeur, donc après que le pipeline state ait été calculé dans notre schéma original (figure 3.5). Aussi, la mise en œuvre du forwarding dans le CV32E40P est plus simple que celles qu'on peut trouver dans les micro-architectures conventionnelles de processeur, qui par exemple exploitent un bypass des accès mémoire.

Enfin, la protection des signaux de contrôle des micro-architectures out-of-order suppose de repenser complètement la solution envisagée dans MAFIA. Il s'agit d'une question de recherche ouverte.

Exploitation de fautes dans la micro-architecture. La littérature ne présente pas encore d'exploitation d'injections de fautes ciblées dans la micro-architecture d'un processeur, mais on peut néanmoins supposer que ce type d'exploitation est réaliste pour un attaquant puissant, doté à la fois d'outils d'injection précis et de temps pour une caractérisation poussée. Ce type d'attaque suppose, d'une part, la capacité d'injecter une faute dans un registre spécifique sur un circuit, ce qui est une hypothèse assez réaliste puisque les derniers travaux sur les injections de fautes dans les mémoires montrent qu'il est possible de cibler un bit mémoire spécifique, de manière reproductible [Col+19 ; Col+22]. Outre l'injection laser, on peut citer l'utilisation, pour l'instant expérimentale, de rayons X qui améliore encore la précision spatiale de l'injection [Anc+17]. D'autre part, le mouvement vers des implémentations matérielles ouvertes, illustré par exemple par la croissance de l'*OpenHW group*[1] ou la plateforme de conception ouverte *Efabless*[2], faciliterait encore la faisabilité d'une caractérisation poussée des effets des fautes dans la micro-architecture dans la mesure où il devient possible d'avoir accès au modèle RTL et au *layout* du circuit ciblé. Ce type d'exploitation semble donc réaliste pour un attaquant puissant.

3.3 Conclusion

Ce chapitre présente deux approches de durcissement de systèmes embarqués contre les attaques matérielles, en proposant une articulation logiciel-matériel pour aller vers des modèles d'attaquants plus puissants. La première approche, implémentée dans la chaîne de compilation PolEn, articule une contre-mesure matérielle, le chiffrement de code, et une contre-mesure logicielle, le polymorphisme de code, chaque contre-mesure se protégeant mutuellement pour couvrir complètement un scénario d'attaque de bout en bout par canaux auxiliaires, de la phase de reverse du système à l'exploitation de traces. La seconde approche, appelée MAFIA, présente un schéma d'intégrité des signaux de contrôle de la micro-architecture du processeur ciblé. La protection matérielle assure aussi l'authenticité du code et l'intégrité de flot de contrôle, et peut également se combiner avec du chiffrement de code pour assurer la confidentialité du code. La protection matérielle se combine avec un schéma logiciel de protection des branchements indirects pour apporter une couverture complète de la partie contrôle des programmes exécutés.

1. https://www.openhwgroup.org
2. https://efabless.com

Chapitre 4

Conclusion

Lors du démarrage de mes travaux de recherche sur la sécurité matérielle, la carte à puce était encore un objet emblématique : des fonctionnalités et des capacités en calcul limitées, pour une surface d'attaque réduite, mais faisant l'objet de recherches et de développement intenses pour atteindre un haut niveau de sécurité. Dix ans plus tard, la carte à puce reste emblématique du marché de la sécurité matérielle : ce marché reste conséquent mais ne voit plus de croissance forte, même s'il est encore soutenu par un accroissement de l'usage des technologies du numérique dans certains domaines (e.g., santé). L'usage de la carte à puce tend aujourd'hui à être complété ou remplacé par d'autres solutions, logicielles ou matérielles, qui exploitent des architectures variées. La demande, toujours plus forte, en capacités de calcul dans les objets numériques pousse à une intégration accrue, entre autres des fonctions de sécurité qui peuvent prendre des formes différentes suivant le système hôte. On assiste donc à une diversification des solutions de sécurisation et de leurs implémentations, sur des architectures matérielles elles aussi de plus en plus diverses. La communauté scientifique a aussi progressé sur de nombreux axes. Les techniques d'apprentissage automatique se sont massivement répandues, entraînant de nouvelles techniques d'attaque mais aussi de nouveaux besoins de sécurisation. Enfin, le domaine de la cryptographie est en train de vivre un tournant majeur avec la mise en usage imminente de primitives post-quantiques. Face à cette diversité accrue, il me semble d'autant plus nécessaire de fournir des outils d'aide à la sécurisation aux spécialistes en charge des implémentations. Ce mémoire fait état de mes travaux sur l'application outillée de contre-mesures contre les attaques matérielles.

4.1 Bilan des travaux abordés dans le mémoire

Mon travail de recherche s'inscrit dans le domaine du logiciel embarqué au sens large, dans une articulation fine avec l'architecture matérielle sous-jacente.

Depuis mon arrivée au CEA en 2011 jusqu'à 2016 environ, j'ai travaillé sur la performance de cœurs de calculs implantés sur des systèmes embarqués, dans l'objectif d'en réduire le temps d'exécution (e.g., [LC12 ; AC13 ; CLC13 ; Cha+14b ; ECC14 ; ECC15b ; QCC15 ; CQC16]) et l'empreinte énergétique (e.g., [End15 ; ECC15a ; ECC16]). En particulier, les réalisations autour de la génération de code dynamique ont constitué le socle technique pour les implémentations ultérieures de polymorphisme de code.

À partir de 2013, et jusqu'à présent, je me suis intéressé aux protections contre les attaques matérielles, avec l'objectif de les appliquer automatiquement, à la compilation. Ce mémoire couvre cette partie de mes travaux de recherche. Le chapitre 2 présente mes travaux sur l'application automatisée de contre-mesures logicielles contre les attaques par injection de fautes et par

canaux auxiliaires. La première partie de ce chapitre présente l'application de contre-mesures logicielles à des systèmes embarqués « sur étagère », dépourvus d'autres contre-mesures, notamment matérielles. Dans cette optique, le compilateur est un outil puissant mais dangereux. Puissant parce qu'il devient possible d'exploiter des techniques d'optimisations de code lors de l'application des contre-mesure, et d'apporter plus de flexibilité aux schémas de protection en élargissant l'espace de configuration de la contre-mesure. De cette manière, on apporte à l'utilisateur du compilateur une grande flexibilité d'utilisation de jouer sur le pouvoir de protection de chaque contre-mesure et sur le surcoût induit. Dangereux parce que les stratégies de transformations de code appliquées par les compilateurs considèrent essentiellement les propriétés fonctionnelles du programme cible, parfois contradictoires avec la préservation de propriétés non fonctionnelles comme des propriétés de sécurité. Dans ces travaux, j'ai également veillé à ne pas imposer de restrictions sur les programmes à durcir, de sorte que les contre-mesures puissent être appliquées le plus largement possible.

La deuxième partie du chapitre 2 présente l'application à la compilation de contre-mesures contre les attaques par injection de fautes et contre les attaques par canaux auxiliaires. Les contre-mesures sont appliquées et évaluées sur des composants sur étagère représentatifs des systèmes embarqués à faible coût et par ailleurs dépourvus de protections, notamment matérielles. Dans nos évaluations, on observe que chaque contre-mesure est susceptible d'apporter un gain en sécurité contre un modèle d'attaquant de puissance modeste, mais que ce gain ne suffit pas contre les modèles d'attaquants à l'état de l'art. On conjecture deux raisons principales à cela. En premier lieu, la couverture apportée par chaque contre-mesure est plus faible que les moyens d'attaques utilisés pour l'évaluation. En second lieu, les moyens d'attaque utilisés ici sont habituellement exploités contre des systèmes dédiés à la sécurité, faisant l'objet d'une conception matérielle spécifique impliquant des protections de natures diverses, sur lesquels sont appliqués également des contre-mesures logicielles visant à parer les exploitations non couvertes par les protections matérielles.

Dans l'objectif d'adresser des modèles d'attaquants plus puissants, il m'a semblé nécessaire de repenser l'articulation avec le support d'exécution matériel pour maîtriser les surcoûts induits par les protections. Le chapitre 3 présente deux approches dans ce sens. Premièrement, la combinaison d'une contre-mesure matérielle, pour l'exécution de code chiffré, avec une contre-mesure logicielle, le polymorphisme de code, pour le durcissement contre les attaques par canaux auxiliaires. Chaque contre-mesure pallie aux faiblesses de l'autre contre-mesure, et la combinaison permet d'adresser un scénario d'attaque de bout en bout, depuis la phase de rétro-conception du système pour l'identification de failles exploitables jusqu'à l'exploitation de mesures physiques dans une cryptanalyse. Deuxièmement, la modification de la micro-architecture d'un processeur pour protéger l'intégrité de ses signaux de contrôle. La nature de la contre-mesure matérielle implique de repenser son interaction avec le logiciel. La contre-mesure matérielle est donc supportée par des aménagements dans le logiciel, automatiquement mis en œuvre à la compilation. En outre, la couverture de la contre-mesure matérielle est étendue par des modifications logicielles, aussi automatiquement mises en œuvre à la compilation : protection des branchements indirects, support des interruptions.

4.2 Perspectives de recherche

Mes travaux se situent à l'interface entre le logiciel et le matériel, et cherchent à exploiter les outils existants comme des leviers pour la sécurisation du logiciel, en coopération avec le matériel. Dans le futur, j'imagine poursuivre ces travaux dans les directions suivantes :

— poursuivre le développement d'outils d'aide à la sécurisation ;

— repenser l'articulation logiciel-matériel pour un meilleur niveau de sécurité sans compromettre la performance.

Préservation de propriétés de sécurité à la compilation Aborder la question de la sécurisation de programmes au travers d'outils met en lumière l'antagonisme entre les visions fonctionnelle et non fonctionnelle des programmes manipulés. Cet antagonisme est particulièrement vrai dans les approches traditionnelles de compilation, qui par construction s'intéressent (presque) exclusivement aux propriétés fonctionnelles des programmes. Il s'agit d'une difficulté fondamentale dès qu'on souhaite manipuler des propriétés non fonctionnelles, comme des propriétés de sécurité, pendant le processus de compilation. Cette problématique est également connue, dans une certaine mesure, de la communauté de la sécurité matérielle (voir par exemple [Bal+15]). J'ai déjà effleuré cette question en introduction de la section 2.1, et la problématique a été plus longuement exposée dans la thèse de Thierno Barry [Bar17] puis dans [Bel+18b]. Cette question a eu un impact concret dans chacune des réalisations techniques décrites dans ce mémoire : nous nous sommes systématiquement posés la question de la place adéquate, dans l'architecture du compilateur, des passes nécessaires à l'application d'une contre-mesure. Si ces passes travaillent trop en amont, la contre-mesure est susceptible d'être défaite par les passes d'optimisation plus aval. Trop en aval, l'application de la contre-mesure est souvent plus complexe parce que le niveau d'abstraction n'est pas adapté. Cependant, cette question n'a pas été traitée frontalement, jusqu'à présent, dans mes travaux.

Jusqu'à très récemment, dans la littérature, cette question était contournée de différentes manières, par exemple par des outils dédiés aux implémentations cryptographiques. Moss et al. proposent un outil *ad hoc* qui remplace le compilateur, permettant de produire un code assembleur protégé par masquage Booléen à partir d'une implémentation exprimée dans un DSL [Mos+12]. Il existe plusieurs approches source à source, entre autres maskComp [Bar+16] qui produit des implémentations masqués en C à partir d'une implémentation de référence en C, ou Usuba [MD19] qui génère des implémentations *bitslicées* à partir d'une implémentation décrite dans un DSL. Dans ces cas, la génération du code machine est reléguée au compilateur de la plateforme cible, ce qui ne permet pas d'avoir une garantie sur la préservation des propriétés de sécurité dans le code exécuté. Enfin, Jasmin [Alm+17 ; Alm+20] est un autre projet notable, qui cible les implémentations cryptographiques, en apportant un DSL et un compilateur dédié dans lequel les transformations de code à la compilation ou leurs résultats sont prouvés robustes aux attaques par observation de propriétés temporelles (*timing attacks*).

Vu et al. [Vu+20 ; Tua21 ; Vu+21] ont ouvert la question du transport et de la préservation de propriétés, en particulier non fonctionnelles, dans le pipeline de passes d'un compilateur. L'approche proposée consiste à opacifier les propriétés ciblées ou certains éléments permettant de supporter ces propriétés. Cette opacification permet de protéger les propriétés des transformations appliquées dans les passes de compilation, ce qui laisse le champ libre à l'application de compilations agressives sur le reste du code. Cette approche est originale puisqu'elle est agnostique des propriétés de sécurité à protéger. Elle apporte aussi une certaine flexibilité dans les pratiques de sécurisation puisqu'elle permet d'exprimer des mécanismes de protection dans le code source sans modifier le compilateur pour chaque nouvelle contre-mesure. On pourrait également imaginer exploiter cette approche dans le compilateur, pour que les propriétés de sécurité, apportées par des passes d'application de contre-mesures, soient préservées dans les passes de compilation en aval. Il s'agit d'une approche qui s'articule naturellement avec les travaux présentés dans ce mémoire, et qui ouvre de nombreuses perspectives pour la sécurisation de code.

Combinaison de contre-mesures Jusqu'à présent, nous avons étudié des contre-mesures individuellement, à l'exception d'une exploration de l'apport du schéma de tolérance aux fautes pour durcir les points de vérification du schéma de traceur (section 2.1.2.6, page 15), et des travaux sur l'exécution polymorphe de code chiffré qui montrent comment deux protections peuvent mutuellement se protéger (section 3.1).

Dans le cadre des attaques matérielles, il est naturel de combiner plusieurs vecteurs d'attaques dans un modèle d'attaquant. Par exemple, l'observation du fonctionnement de la cible permet d'évaluer l'effet d'une injection de faute. A contrario, une injection de faute est susceptible d'affecter l'efficacité d'une contre-mesure en side-channel, par exemple en dégradant la qualité d'un aléa. En conséquence, la protection contre ces attaques passe par la combinaison de plusieurs principes de protection : par exemple, contre les fautes, par la combinaison de la détection de perturbations et de la redondance ; contre le side-channel, par la combinaison de techniques de dissimulation et de masquage.

La question de la combinaison de contre-mesures est un enjeu majeur pour l'application outillée de contre-mesures : la conception d'un système sécurisé contre les attaques matérielles doit envisager plusieurs vecteurs d'attaque, exploités seuls ou en combinaison. Du point de vue scientifique, cet axe de recherche aborde des problématiques peu traitées dans l'état de l'art : une contre-mesure ne doit pas nuire aux propriétés fonctionnelles du code à laquelle elle est appliquée (préserver la fonction originelle), et elle ne doit pas non plus nuire aux propriétés non fonctionnelles (robustesse) des autres contre-mesures. Cette question peut également s'inscrire dans une forme de généralisation du point développé ci-dessus concernant la préservation de propriétés de sécurité à la compilation : s'il existe un moyen d'exprimer et de manipuler des propriétés de programme, qu'elles soient fonctionnelles ou non fonctionnelles, on peut alors faire en sorte que la manipulation de chaque propriété soit « étanche » à la manipulation des autres propriétés du programme. Il est également possible de repenser l'articulation de propriétés de sécurité, de sorte qu'une contre-mesure couvre plusieurs vecteurs d'attaques. On trouve dans la littérature plusieurs approches théoriques de cette question, mais à ma connaissance leur mise en œuvre dans des outils n'a pas encore été abordée.

Enfin, un autre axe de recherche intéressant est la combinaison de contre-mesures matérielles et logicielles, dans la perspective des travaux évoqués au chapitre 3. La technique de vérification matérielle récemment développée dans la thèse de Simon Tollec [Tol+24] permet d'analyser la robustesse d'un processeur aux injection de fautes. Cette technique permet de mettre en évidence les injections de fautes qui ne sont pas capturées par les contre-mesures matérielles, puis de vérifier si ces vulnérabilités potentielles sont exploitables lors de l'exécution d'un programme. Par exemple, l'analyse de robustesse de OpenTitan a montré que le banc de registres du processeur Secure Ibex est vulnérable à un bit-flip. En revanche, cette vulnérabilité n'est pas exploitable pendant l'exécution du *secure boot* de OpenTitan, car l'exploitation de la vulnérabilité est (fortuitement) protégée par des contre-mesures logicielles. Cette étude illustre le potentiel d'analyses conjointes du matériel et du logiciel. Ces techniques peuvent être utiles pour l'application de contre-mesures logicielles sur un design matériel existant, déjà mis en production, pour lequel de nouvelles vulnérabilité sont identifiées. À plus long terme, il serait intéressant d'envisager la conception de contre-mesures matérielles dont la couverture est maîtrisée mais réduite pour amoindrir le surcoût de la sécurisation, cette couverture étant complétée par des contre-mesures logicielles ciblées.

Cette perspective de recherche est en cours d'investigation dans le projet FlexSecurity, dans le cadre de la combinaison de contre-mesures logicielles contre les attaques par canaux auxiliaires.

Application automatisée de contre-mesures matérielles Les techniques de compilation logicielle ont de nombreuses similitudes avec les techniques de synthèse matérielle. Il est donc

naturel d'envisager l'application de contre-mesures matérielles avec une approche similaire. Dans le domaine de la sûreté par exemple, certaines techniques de durcissement, comme la redondance, peuvent être automatiquement prises en charge par le flot de synthèse. Steven Derrien et son équipe, à l'IRISIA, développent un flot pour la synthèse automatisée de micro-architectures de processeurs à partir d'une description fonctionnelle de l'architecture ciblée [GRD22]. Cette approche, originale, soulève de nombreuses questions, en particulier pour l'intégration de contre-mesures matérielles. À court terme, on souhaite en particulier utiliser MAFIA (section 3.2) comme cas d'étude : certains blocs, comme le module CACFI, se prêtent bien à une description architecturale, et la synthèse micro-architecturale pourrait être automatisée. En revanche l'intégration de ce module dans la micro-architecture du processeur suppose une analyse fine, notamment pour la construction du pipeline state, et est donc plus complexe à traiter automatiquement.

Cette perspective de recherche est en cours d'investigation dans le projet ANR LOTR (2023-2027) coordonné par Steven Derrien, IRISA.

Vérification de conformité L'application outillée de contre-mesures doit aller de pair avec des méthodes, et si possible des outils, permettant de valider la conformité des composants logiciels produits. La communauté scientifique travaille depuis longtemps sur des moyens pratiques d'évaluer la sécurité, en rejouant des attaques connues soit directement sur le système, soit dans un environnement de simulation. Les méthodes formelles peuvent compléter cette estimation pratique de la robustesse, en apportant une garantie de conformité à un modèle de sécurité. Elles présentent aussi l'avantage de pouvoir être intégrées tôt dans un processus de conception, avant que le système final ne soit disponible, et donc avant que des tests de sécurité *in situ* soient réalisables.

Vérification de propriétés de sécurité. L'automatisation de l'application de contre-mesures est un moyen de limiter l'intervention humaine là où elle a une valeur ajoutée plus faible, donc de réduire les risques d'erreurs pouvant donner lieu à des vulnérabilités. Cependant, les implémentations d'outils, comme les compilateurs, sont elles aussi susceptibles de contenir des bugs. Dès le projet PROSECCO en 2015, nous avons voulu augmenter le niveau de confiance d'une chaîne de sécurisation logicielle par l'articulation du compilateur avec un outil de vérification des propriétés de sécurité [Bel+21]. L'objectif était de fournir une chaîne de sécurisation complète, englobant le compilateur et des outils de vérification de sécurité au niveau binaire. Les outils de vérification, développés par le LIP6, ont été exploités pour vérifier la mise en œuvre correcte de contre-mesures décrites dans ce mémoire (section 2.1.1.3, section 2.2.2.6).

À court et moyen terme, ce sous-axe cible deux verrous liés à l'utilisation de techniques formelles de vérification de propriétés de sécurité : (a) caractériser les conditions d'exploitation d'une vulnérabilité, et (b) intégrer dans l'analyse de robustesse des modèles matériels plus précis.

(a) Caractérisation des conditions d'exploitation d'une vulnérabilité. Les méthodes formelles permettent de garantir l'absence de vulnérabilité d'un (modèle de) système à un modèle d'attaquant, et dans certains cas peuvent fournir un contre-exemple illustrant comment une propriété (e.g., de sécurité) est invalidée. Ce premier verrou est lié au fait que, lorsque la preuve de sécurité n'est pas satisfaite, ces méthodes ne permettent pas de *quantifier* le degré d'exploitabilité de la vulnérabilité identifiée. C'est-à-dire que le contre-exemple retourné par l'analyse n'est pas suffisant pour caractériser comment la propriété analysée n'est pas satisfaite. Il est donc important d'aider l'utilisateur à caractériser chaque vulnérabilité potentielle représentée par un contre-exemple : pour faire le tri entre les vulnérabilités présentant un réel risque et les vulnérabilités nécessitant des conditions d'exploitation inatteignables en pratique ; pour mesurer le

risque représenté par chaque vulnérabilité, et le mettre éventuellement dans la perspective d'un modèle d'attaquant.

Nous avons récemment réalisé une première approche permettant de construire une formule logique représentant les conditions d'exploitation de vulnérabilités [Sel+24]. Lorsque l'analyse aboutit, elle formule les conditions nécessaires et suffisantes d'exploitation de la vulnérabilité, permettant la caractérisation précise du risque associé à chaque vulnérabilité. Nous avons aussi montré que cette approche est beaucoup plus efficace que les approches par énumération (e.g., simulation sur chaque état initial possible du système) pour la caractérisation d'un grand espace d'états.

Cependant, cette approche suppose que l'utilisateur formalise un langage (composé de formules logiques de premier ordre) qui est exploité pour l'inférence de la formule représentant les conditions d'exploitation de la vulnérabilité. Trop simple, le langage permet une exploration rapide mais imprécise (car une caractérisation plus précise n'est pas formulable dans ce langage). Trop complexe, le langage ne permet pas à l'exploration d'aboutir dans un temps raisonnable, ou bien donne des formules logiques insolvables.

Enfin, nous avons montré comment cette approche permet de caractériser les conditions initiales permettant l'exploitation d'une instance de faute sur un programme, ce qui peut être utile pour déterminer les points d'injection de fautes les plus facilement exploitables, et donc les contre-mesures à déployer en priorité. En d'autres termes, l'analyse de robustesse aux fautes à l'aide de cette approche suppose toujours de faire l'analyse de chaque (instance de) faute séparément, ce qui suppose d'itérer sur chaque modèle de faute et sur chaque point d'injection pour avoir un résultat d'analyse de robustesse complet. Il serait intéressant de voir comment étendre l'approche proposée dans [Sel+24] à la caractérisation du modèle de faute.

(b) Vers des modèles de fautes plus précis. Le second verrou concerne l'application des méthodes formelles à des modèles de sécurité incluant les attaques matérielles. L'application de méthodes formelles à la sécurité matérielle suppose d'affiner les modèles employés, afin de représenter précisément les interactions subtiles entre le matériel et le logiciel. La communauté scientifique est active sur ce sujet récent. On peut par exemple citer [dHM22 ; Nas+22 ; Ric+22 ; Bat+23] comme diverses manières d'aborder l'usage de techniques formelles pour la sécurité matérielle, dans un modèle d'attaquant variable.

Dans le cadre de la thèse de Simon Tollec, nous avons montré comment réaliser une analyse formelle de la robustesse de programmes aux fautes en tenant compte de l'impact potentiel des fautes dans la micro-architecture du processeur [Tol+22c ; Tol+23 ; Tol+24]. Ces approches permettent d'identifier si une faute, appliquée dans la micro-architecture d'un processeur, permet d'invalider une propriété de sécurité (ou de satisfaire les conditions de réussite d'une attaque). Cette analyse de robustesse considère non seulement l'implémentation micro-architecturale du processeur ciblé, mais aussi le programme exécuté, car l'exploitation d'une faute dépend de l'état initial de la mémoire ou d'une séquence particulière d'instructions. Cependant, le fait d'exploiter des modèles micro-architecturaux dans l'analyse de robustesse mène rapidement à des modèles formels complexes, ce qui rend l'analyse de satisfiabilité difficile à résoudre. Cette complexité est exacerbée par la présence de fautes qui entraîne une explosion combinatoire du nombre d'états atteignables. En l'état, les techniques actuelles ne permettent pas de faire l'analyse de robustesse d'implémentations micro-architecturales complexes (pipelines avec un grand nombre d'étages, superscalaires, etc.), de programmes de grande taille, ou de tenir compte de modèles d'attaquants avancés (e.g. fautes multiples).

Cette perspective de recherche est en cours d'investigation dans la thèse de Simon Tollec, et se poursuivra dans le cadre d'une collaboration avec l'ANSSI et dans le cadre d'autres projets collaboratifs en cours de montage.

Vérification fonctionnelle. Il est également nécessaire d'assurer que la sécurisation (qu'elle soit outillée ou non) n'altère pas les propriétés fonctionnelles du système. Dans ce sous-axe, on trouve deux familles d'approches.

D'une part, les compilateurs certifiés (e.g., CompCert [Ler09], Jasmin [Alm+17 ; Alm+20]) apportent une preuve d'équivalence fonctionnelle entre le code source et le code assembleur produit. Actuellement, la sémantique de CompCert ne permet pas de traiter naturellement du code binaire, et ne supporte pas l'analyse de propriétés non fonctionnelles, nécessaire pour la sécurité, ce qui ouvre des questions de recherche importantes (voir par exemple [Mon24]). Ce type d'approche est monolithique, et nécessite un effort d'ingénierie et des temps de développement considérables.

D'autre part, l'analyse de l'équivalence fonctionnelle de programmes est un sujet peu exploré, mais qui peut naturellement s'appliquer à l'analyse de code binaire. Cette approche a l'avantage d'être plus modulaire puisqu'elle est agnostique des autres outils d'une chaîne de production logicielle, comme le compilateur. On souhaite généraliser la technique utilisée par Daniel et al. pour la vérification d'exécution en temps constant [DBR20]. Cette technique utilise une analyse relationnelle de couples de traces d'exécution, et on souhaite la généraliser à l'analyse d'équivalence de programmes. Cela permettrait de faire la vérification des propriétés fonctionnelles d'un programme dans lequel des contre-mesures ont été ajoutées, en comparant deux versions du même programme compilé avec ou sans contre-mesures. Outre la vérification de programmes sécurisés, cette perspective ouvre un vaste champ d'applications potentielles à d'autres techniques de transformation de programmes.

Cette perspective de recherche sera abordée dans une thèse à démarrer à la rentrée universitaire 2024-2025, en collaboration avec Frederic Recoules et Sébastien Bardin (CEA-List).

Chapitre 5

CV détaillé

5.1 Notice bibliographique

État civil

	Damien Couroussé
Né le	24 mai 1979
Adresse	Commissariat à l'énergie atomique et aux énergies alternatives
	Institut List | Minatec Campus
	17 avenue des Martyrs | 38054 Grenoble Cedex | France
Téléphone	+33 (0)4 38 78 04 66
E-mail	damien.courousse@cea.fr

Parcours professionnel et situation actuelle

juil. 2003 – sept. 2004	Ingénieur d'études, ACROE. *Développement d'un driver pour une carte d'acquisition A.N. et interfaçage avec un moteur de simulation d'objets physiques. Intégration à une plateforme de simulation multisensorielle en temps réel.*
sept. 2004 – juin 2008	Moniteur puis ATER au département Informatique de l'IUT2, UPMF Grenoble. *64 à 92h équiv. TD par an. Cours enseignés : architecture des ordinateurs, systèmes d'exploitation, architecture des réseaux, analyse et modélisation des systèmes d'information, algorithmique.*
oct. 2004 – oct. 2008	Ingénieur-Chercheur contractuel et Doctorant, laboratoire ICA, Grenoble. *Dans le cadre du projet européen Enactive Interfaces (25 laboratoires en Europe et Amérique du Nord). Développement logiciel pour une plateforme de simulation multisensorielle (geste, image, son) pour la Réalité Virtuelle. Participation à un projet de création d'entreprise résultant des travaux de recherche du laboratoire : ERGOS Technologies ; livraison des développements logiciels, formation et support client, mise en place de démonstrateurs.*
jan. 2009 – déc. 2010	Expert Technique, Logica, Grenoble, centre d'expertise *Machine to Machine. Encadrement technique de l'équipe* informatique embarquée *(3 à 5 personnes) : réponse aux appels d'offre, formation, pilotage technique des projets. STM, division HED : prototypage d'un gestionnaire de ressources, portage d'un environnement de tests vidéo pour un middleware de set-top-box. Participation aux projets collaboratifs européens en cours d'instruction.*
jan. 2011 –	Ingénieur de Recherche, CEA-List, Grenoble.

Formation

2002	Diplôme d'ingénieur de l'INSA de Rennes, spécialité Électronique et Systèmes de Communication.
2003	DEA de l'INP-Grenoble, spécialité Sciences Cognitives.
2008	Doctorat de l'INP-Grenoble. École Doctorale Ingénierie pour la Santé, la Cognition et l'Environnement (EDISCE), Spécialité Ingénierie de la Cognition, de l'Interaction, de l'Apprentissage et de la Création. Titre de la thèse : « *Haptic Processor Unit : vers une plateforme transportable pour la simulation temps-réel synchrone multisensorielle* » Thème de recherches : simulation multisensorielle de phénomènes physiques dynamiques pour le jeu instrumental Direction de thèse : Claude Cadoz, I.R. du Ministère de la Culture, laboratoire ICA & ACROE. Co-encadrement : Jean-Loup Florens, I.R. de l'ACROE ; Annie Luciani, I.R. du Ministère de la Culture & laboratoire ICA. Rapporteurs : Indira Thouvenin, Professeure à l'université de Compiègne ; Antoine Ferreira, Professeur à l'INSA Centre Val de Loire

5.2 Activités de recherche

La figure 1.1 page 4 donne un aperçu chronologique des jalons majeurs au cours de mes activités de recherche au CEA. Cette figure reprend : (i) les thèses en cours et déjà soutenues ; (ii) les projets collaboratifs dans lesquels je me suis particulièrement impliqué, à la fois dans les phases de montage puis d'instruction du projet ; (iii) les résultats scientifiques les plus marquants ainsi que les publications scientifiques associées les plus représentatives.

5.2.1 Rayonnement scientifique et contributions à la communauté

5.2.1.1 Participation à des comités de programme et relecture

J'ai participé aux comités de programme des conférences ou workshops suivants : COMPAS 2019, Workshop on Cryptography and Security in Computing Systems (CS2) 2019, DAC 2019 (reviewer externe), DATE 2019 (reviewer externe), EuroLLVM 2019, NEWCAS 2019, SILM 2020-2021, PARMA-DITAM 2023-2024. PROOFS 2019-2020, 2022-2024,

J'ai également fait des revues de soumissions aux journaux suivants : TACO (ACM), Software Quality Journal (Springer), DTRAP (ACM), SoftwareX (Elsevier).

5.2.1.2 Participation à des comités d'organisation de conférence ou de workshop

Conférence Enactive/07 (19-24 Novembre 2007, Grenoble), Journée thématique sur les Attaques par Injection de Fautes (JAIF) 2018-2024 – *general chair* pour JAIF 2019, RISC-V week 2019-2022, Journée 2020 du Groupe de Travail Méthodes Formelles pour la Sécurité du GdR Sécurité Informatique.

5.2.1.3 Participation à des jurys de thèse

Soutenances :

— Thèse de Julien Proy, « Sécurisation systématique d'applications embarquées contre les attaques physiques », Doctorat de l'Université de recherche Paris Sciences et Lettres, soutenue le 17 Juin 2019 à Gardanne. Participation au jury comme examinateur.

— Thèse de Son Tuan Vu, « Optimizing Property-Preserving Compilation », Doctorat de l'Université Sorbonne, soutenue le 2 Avril 2021 à Paris et en visio-conférence. Participation au jury comme examinateur.

— Thèse de Inès Ben El Ouahma, « Analyse de robustesse et sécurisation de codes assembleur contre des attaques physiques », Doctorat de l'Université Sorbonne, soutenue le 7 Avril 2021 à Paris et en visio-conférence. Participation au jury comme examinateur.

— Thèse de Duy-Phuc Pham, « *Leveraging side-channel signals for IoT malware classification and rootkit detection* », Doctorat de l'Université Rennes 1, soutenue le 13 Janvier 2023 à Rennes et en visio-conférence. Participation au jury comme examinateur.

Comités de suivi à tiers- ou mi-parcours :

— Thèse de Duy-Hieu Buy, « An innovative lightweight cryptography system for Internet-of-Things ULP applications », comité à mi-parcours pour l'EEATS de l'Université de Grenoble Alpes, le 17 Octobre 2017.

— Thèse de Son Tuan Vu, « Flot de compilation et propriétés pour l'analyse ou la transformation de code » comité à tiers-parcours pour l'EDITE de l'Université Sorbonne, le 7 Décembre 2019.

5.2.1.4 Expertise d'appels à projets

En interne CEA

— Expert CEA (depuis 2017) sur les thématiques logiciel embarqué, compilation, cyber-sécurité matérielle.

— Expert senior CEA (depuis 2022) sur les thématiques : compilation, cyber-sécurité, systèmes embarqués, parallélisation.

— Membre des groupes de travail *Leti2030* (CEA-Leti, 2020) sur la thématique cyber-sécurité (2020), et *Reveal* sur la thématique cyber-sécurité (CEA-List, 2021).

— Expertise de projets pour le programme Carnot Exploratoire CEA-Leti (2022).

— Participation aux revues de feuille de route du département DSCIN pour la thématique cyber-sécurité. Pilotage et animation de la sous-thématique *sécurité matérielle embarquée* en 2020-2021.

Expertise externe

— Évaluation de projets pour l'appel à projets générique 2024 de l'ANR (CE25 – Sciences et génie du logiciel, CE39 – Sécurité globale, résilience et gestion de crise, cybersécurité) ;

— initiatives de Recherche à Grenoble Alpes (IRGA) de l'Université Grenoble Alpes, pour l'appel 2024.

5.2.2 Encadrement

Depuis 2011, j'ai participé à l'encadrement de 6 thèses, dont 5 ont déjà été soutenues. Une thèse est en cours. J'ai également encadré 9 chercheurs ou ingénieurs, 8 d'entre eux étant docteurs. J'ai encadré 3 étudiants pendant mes deux années passées chez Logica, et 11 étudiants de Master depuis ma prise de fonctions au CEA.

5.2.2.1 Doctorants

Fernando Endo – 2012-2015

Sujet de thèse :	« Génération dynamique de code pour l'optimisation énergétique ». École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII), Université de Grenoble. [End15]
Période :	2012-2015. Thèse soutenue le 18 Septembre 2015.
Projet support :	Financement interne CEA (CTBU)
Équipe d'encadrement :	Henri-Pierre Charles (CEA, direction), Damien Couroussé
Taux d'encadrement :	50 %
Publications communes :	[Cha+14a ; Cha+14b ; ECC14 ; ECC15a ; ECC15b ; ECC16 ; ECC17]
Poste(s) occupé(s) :	Post-doctorat dans l'équipe ALF, IRISA (2015-2016) ; postes divers en qualité de *firmware engineer* ; actuellement *Senior Firmware Enginner* chez XenomatiX, Bruxelles.

Thierno Barry – 2014-2017

Sujet de thèse :	« Sécurisation de l'exécution des applications contre les attaques par injection de fautes par une contre-mesure intégrée au processeur ». École Doctorale Sciences, Ingénierie, Santé (SIS, spécialité Microélectronique), École des Mines de Saint-Étienne. [Bar17]
Période :	2014-2017. Thèse soutenue le 24 Novembre 2017.
Projet support :	Financement interne CEA (CTBU)
Équipe d'encadrement :	Bruno Robisson (CEA, direction), Damien Couroussé
Taux d'encadrement :	50 %
Publications communes :	[Cou+14 ; BCR15 ; Cou+15 ; BC16 ; BCR16 ; Cou+16a ; Le +16a ; Le +16b ; Bar+17 ; BCH17 ; Bel+17 ; Bel+18b ; Cou+18]
Poste(s) occupé(s) :	*Principal Mobile Security Evaluator*, CESTI Thalès (2017-2019) ; *Senior Mobile Security Engineer*, Huawei UK (2019-2021) ; Actuellement *Head of Security Lab*, Mediatek (2021-.).

Nicolas Belleville – 2016-2019

Sujet de thèse :	« Compilation pour l'application de contre-mesures contre les attaques par canal auxiliaire ». École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII), Université de Grenoble. [Bel19]
Période :	2016-2019. Thèse soutenue le 21 Novembre 2019.
Projet support :	projet ANR PROSECCO
Équipe d'encadrement :	Henri-Pierre Charles (CEA, direction), Karine Heydemann (Sorbonne Université, co-direction), Damien Couroussé
Taux d'encadrement :	33 %
Publications communes :	[Bel+17 ; BC18 ; Bel+18a ; Bel+18b ; Cou+18 ; Bel+19 ; Bel+20a ; Bel+20b ; Mas+20]
Poste(s) occupé(s) :	Actuellement Ingénieur de recherche au CEA-List (depuis le doctorat jusqu'à présent)

Thomas Chamelot – 2019-2022

Sujet de thèse :	« Sécurisation par co-design matériel / logiciel contre les attaques par injection de fautes ». École Doctorale Informatique, Télécommunications, Électronique (EDITE), Sorbonne Université. [Cha22]
Période :	2019-2022. Thèse soutenue le 28 Novembre 2022.
Projet support :	projet ANR COFFI
Équipe d'encadrement :	Karine Heydemann (Sorbonne Université, direction), Damien Couroussé
Taux d'encadrement :	50 %
Publications communes :	[CCH21 ; CCH22a ; CCH22b ; CCH22c ; CCH23]
Poste(s) occupé(s) :	Actuellement *Cybersecurity Evaluator* chez Airbus Toulouse

Lorenzo Casalino – 2021-2024

Sujet de thèse :	« (On) The Impact of the Micro-architecture on Countermeasures against Side-Channel Attacks ». École Doctorale Informatique, Télécommunications, Électronique (EDITE), Sorbonne Université.
Période :	Janvier 2021-2024. Thèse soutenue le 30 Janvier 2024.
Projet support :	projet ANR IDROMEL
Équipe d'encadrement :	Karine Heydemann (Sorbonne Université, Thalès, direction), Damien Couroussé, Nicolas Belleville (CEA)
Taux d'encadrement :	33 %
Publications communes :	[Cas+23]

Simon Tollec – 2021-(2024)

Sujet de thèse :	« Modélisation des fautes dans la vérification de contre-mesures à des attaques matérielles par injection de fautes ». École Doctorale Sciences et Technologies de l'Information et de la Communication (STIC), Université Paris-Saclay.
Période :	2021-2024
Projet support :	Financement interne CEA (CTBU)
Équipe d'encadrement :	Mathieu Jan (CEA, direction), Karine Heydemann (Sorbonne Université, Thalès, co-direction), Mihail Asavoae (CEA), Damien Couroussé
Taux d'encadrement :	25 %
Publications communes :	[Tol+22a ; Tol+22b ; Tol+22c ; Tol+23 ; Tol+24]

5.2.2.2 Post-doctorants et ingénieurs de recherche

Hassan Noura – 2014-2014

Thématique :	« Génération de code dynamique appliquée à la cryptographie ». Contrat de post-doctorat.
Période :	Janvier – Décembre 2014
Projet support :	projet ANR Cogito
Équipe d'encadrement :	Damien Couroussé
Publications communes :	[Cou+14 ; NC14 ; NC15a ; NC15b]

Caroline Quéva – 2015-2017

Thématique :	« Génération de code dynamique pour la performance des systèmes embarqués ». Ingénieur de recherche.
Période :	Janvier – Décembre 2014
Projet support :	projet ArrowHead
Équipe d'encadrement :	Damien Couroussé, Henri-Pierre Charles (CEA)
Publications communes :	[CQY15a ; CQY15b ; QCC15 ; CQC16]

Abderhamane Seriai – 2016-2017

Thématique :	« Automatic generation of dynamic code generators from legacy code ». Contrat de post-doctorat.
Période :	Avril 2016 – Juillet 2017
Projet support :	projet ANR Cogito
Équipe d'encadrement :	Damien Couroussé
Publications communes :	[Bel+17 ; Bel+18b]

Saana Herroumi – 2017-2019

Thématique :	« Embedded machine learning for attack detection ». Contrat de post-doctorat.
Période :	Septembre 2017 – Février 2019
Projets support :	projets L4S, CLAPs
Équipe d'encadrement :	Anca Molnos (CEA), Damien Couroussé
Publications communes :	[Ker+18]

Lionel Morel – 2017-2020

Thématique :	« Sécurité embarquée pour les applications médicales ». Contrat d'accueil chercheur, en détachement de l'INSA de Lyon.
Période :	Septembre 2017 – Août 2020
Projets support :	projet Serene-IoT
Équipe d'encadrement :	Damien Couroussé
Publications communes :	[Mor+18a ; Mor+18b ; MC19a ; MC19b ; MCH21]

Nisrine Jafri – 2019-2021

Thématique :	« Vérification de programmes au niveau binaire en présence d'attaques par injection de fautes ». Ingénieur de recherche.
Période :	Novembre 2019 – Janvier 2021
Projets support :	projet CLAPs
Équipe d'encadrement :	Damien Couroussé
Publications communes :	–

Yanis Sellami – 2021-2023

Thématique :	« Méthodes formelles pour la vérification de programmes au niveau binaire en présence d'attaques par injection de fautes ». Ingénieur de recherche.
Période :	Mai 2021 – Décembre 2023
Projets support :	projet Pulse-IA
Équipe d'encadrement :	Damien Couroussé, Sébastien Bardin (CEA)
Publications communes :	[Sel+24]

Cyril Hugounenq – 2022-.

Thématique :	« Compilation de contre-mesures contre les attaques par canal auxiliaire ». Ingénieur de recherche.
Période :	Décembre 2022 – .
Projets support :	projet FlexSecurity
Équipe d'encadrement :	Nicolas Belleville (CEA), Damien Couroussé
Publications communes :	–

5.2.2.3 Ingénieurs de développement

Étienne Louboutin – 2020-2021

Thématique :	Développement et maintenance des outils Cogito.
Période :	Juillet 2020 – Novembre 2021
Projets support :	projets Sarmenti, FlexSecurity
Équipe d'encadrement :	Damien Couroussé, Nicolas Belleville

5.2.2.4 Étudiants de Master

Damien Vallat, 2008-2009 – Licence Professionnelle Informatique Embarquée et Mobile, Université Claude Bernard Lyon 1. Encadrement en collaboration avec Matthieu Milléquant (Logica).

Luc Bastian, 2009 – « Développement logiciel de composants embarqués ». PFE 6 mois, Polytech Grenoble.

Adrien Lequeux, 2010 – « Démonstrateur TV 3D avec point de vue libre ». PFE 6 mois, Université de Rennes 1. Encadrement en collaboration avec Matthieu Milléquant (Logica).

Yacine Fall, 2012 – « Instrumentation pour la mesure énergétique sur MPSoc ARM ». Stage de 4 mois, 4e année Polytech Grenoble.

Théo Deleligne, 2013 – « Cholesky decomposition acceleration on a many-core architecture ». PFE 6 mois, École Centrale de Lyon. Encadrement en collaboration avec Yves Durand (CEA).

Charles Aracil, 2013 – « Génération dynamique de code pour microcontrôleur ». PFE 6 mois, Phelma Grenoble, filière Systèmes et Logiciel Embarqué (SLE).
Ce travail a donné lieu à une publication commune : [AC13]

Thierno Barry, 2014 – « Code Polymorphism to Secure Embedded Devices ». PFE 6 mois, Université de Limoges. Les réalisations de Thierno en stage ont contribué aux publications [Cou+14 ; Cou+16a].

Thibault Cattelani, 2015 – « Code Polymorphism for secured embedded devices ». PFE 6 mois, Master SIC (Système Intelligent et Communicant), Université de Cergy-Pontoise.

Nicolas Belleville, 2016 – « Génération automatisée de code embarqué sécurisé avec LLVM ». PFE 6 mois, ENSIMAG.

Tiago Trevisan Jost, 2017 – « Optimised LLVM code generation for a RISC-V core extended with adequate-precision compute units ». PFE 6 mois, Master of Science in Computer Science at the Federal University of Rio Grande do Sul, Brazil. Encadrement en collaboration avec Anca Molnos (CEA).
Ce travail a donné lieu à une publication commune : [Tre+18]

Irénée Groz, 2018 – « Intégrité et confidentialité des programmes et des données pour les systèmes embarqués ». PFE 6 mois, ENSIMAG. Encadrement en collaboration avec Lionel Morel (CEA). Prototypage du moteur de génération polymorphe de code pour l'architecture RISC-V.

Ziad Khalaf, 2020 – « Extension Of A Masking Compiler Against Side-Channel Attacks ». PFE 6 mois, ESISAR. Encadrement en collaboration avec Nicolas Belleville (CEA).

Juan Suzano Da Fonseca, 2021 – « Integration in an ASIC of a secured RISC-V processor ». PFE 6 mois, CPE Lyon. Encadrement en collaboration avec Thomas Chamelot (CEA) et Mikael Le Coadou (CEA).

Simon Tollec, 2021 – « Verification of dependability properties on formal RISC-V pipeline models ». PFE 6 mois, Telecom Paris. Encadrement en collaboration avec Mathieu Jan (CEA), Karine Heydemann (Sorbonne Université, Thalès), et Mihail Asavoae (CEA).

Simon Baissat-Chavant, 2023 – « Émulation d'attaques physiques avec QEMU ». PFE 6 mois, ENSEIRB Matmeca – INP Bordeaux. Encadrement en collaboration avec Nicolas Belleville (CEA).

5.2.3 Contributions à des projets collaboratifs de recherche

Cette section dresse une synthèse des principaux projets collaboratifs de recherche dans lesquels j'ai été impliqué, par ordre chronologique.

Dans les tables ci-dessous : l'abréviation P.c.i. désigne les *principaux chercheurs impliqués* avec lesquels j'ai collaboré, par ordre alphabétique. Les personnes mentionnées sont affiliées au CEA lorsque leur affiliation n'est pas mentionnée. La description des *moyens financés* couvre mon périmètre de recherches uniquement, sachant que des projets peuvent impliquer plusieurs équipes au CEA. De même, la description des *faits marquants* concerne seulement les résultats marquants en lien avec mes activités de recherche.

iGlance – Interactive Free Viewpoint for 3D TV

Programme :	MEDEA+
Période :	2008-2011
Partenaires :	CEA, Logica, Philips, STMicroelectronics, Task24, TIMA, Verum, ...
P.c.i. :	Michel Imbert (coordinateur, STMicroelectronics), Matthieu Milléquant (Logica), Daniel Ruijters (Philips), Luc Verhaegh (Task24)
Moyens financés :	Ingénieur développement, temps personnel de recherche
Faits marquants :	Réalisation d'une preuve de concept TV-3D sur set-top-box
Implication :	Chef de projet Logica

MIND – Langages de programmation à base de composants adaptés à l'embarqué

Programme :	Pole de compétitivité Minalogic
Période :	2008-2011
Partenaires :	CEA, STMicroelectronics, VERIMAG, Logica, Schneider Electric, ...
P.c.i. :	Mathieu Jan, Matthieu Milléquant (Logica), Maurice Pitel (coordinateur, Schneider Electric), Marc Poulhiès (VERIMAG)
Moyens financés :	Ingénieur développement, temps personnel de recherche
Faits marquants :	Prototypage d'un IDE pour un langage de programmation à composants
Implication :	Contributeur pour Logica, puis pour le CEA

COBRA – Ressources de calcul pour applications haute performance

Programme :	CATRENE
Période :	2010-2014
Partenaires :	CAPS Entreprise, CEA, ST-Ericsson, STMicroelectronics, TU Delft, ...
P.c.i. :	Selma Azaiez, Thierry Goubier, Vincent Lorquet (STMicroelectronics), Laurent Bertaux (CAPS)
Moyens financés :	5 HA Ingénieur de recherche, temps personnel de recherche
Faits marquants :	Prototypage de la chaîne de compilation pour τC, un langage de programmation *data-flow* [GCA14]
Implication :	Chef de projet pour le laboratoire

SMECY – Smart Multicore Embedded Systems

Programme :	Artemis
Période :	2010-2014
Partenaires :	CEA, STMicroelectronics, ACE, TU Delft, UNIBO ...
P.c.i. :	Henri-Pierre Charles, Mathieu Jan, Yves Lhuillier, Julien Mottin, François Pacull (coordinateur), Germain Haugou (STM)
Moyens financés :	Temps personnel de recherche
Faits marquants :	Prototypage de `deGoal`, chaîne de compilation pour la génération dynamique de code appliquée à la performance des systèmes embarqués [Cha+14b] ; spécialisation du code d'un allocateur mémoire pour Linux [LC12]
Implication :	Contributeur

HARP – Heterogeneous Architectures for Parallel Computing

Programme :	MEDEA
Période :	2013-2017
Partenaires :	CEA, STMicroelectronics, UAB, ...
P.c.i. :	David Castells (UAB), Yves Durand, Michel Imbert (STM), Vincent Lorquet (STM)
Moyens financés :	Temps personnel de recherche
Faits marquants :	Génération dynamique de code appliquée à la performance des kernels de calcul dans des architectures many-core
Implication :	Contributeur, chef de projet pour le laboratoire

ESCADE – Essaim de Calculateurs Distribués Embarqués

Programme :	Carnot
Période :	2013-2015
Partenaires :	CEA
P.c.i. :	Marc Duranton (coordinateur), Stéphane Louise, Thierry Goubier
Moyens financés :	Temps personnel de recherche
Faits marquants :	Spécification de l'intégration d'une technique de spécialisation sur les données au runtime dans un langage *data-flow*
Implication :	Contributeur

ArrowHead – Service Interoperability Enabling Collaborative Automation

Programme :	ARTEMIS
Période :	2013-2017
Partenaires :	CEA, Thales, ...
P.c.i. :	Henri-Pierre Charles, Jean-Pierre Krimm, Yves Lhuillier, Maxime Louvel
Moyens financés :	36 HM CDD recherche, temps personnel de recherche
Faits marquants :	Prototypage d'une librairie auto-optimisante pour les applications IoT [QCC15 ; CQC16]
Implication :	Montage de projet, Contributeur

COGITO – Génération de Code au Runtime pour la Sécurisation de Composants

Programme :	ANR, INS 2013
Période :	2013-2017
Partenaires :	CEA, EMSE, INRIA Rennes, XLIM
P.c.i. :	Philippe Jaillon (EMSE), Jean-Louis Lanet (XLIM, INRIA), Olivier Potin (EMSE), Bruno Robisson
Moyens financés :	24 HM CDD, temps personnel de recherche
Faits marquants :	Publication de la première preuve de concept du polymorphisme de code compatible avec les contraintes de systèmes embarqués à faibles ressources [Cou13 ; Cou+14 ; Cou+16a]
Implication :	Coordination du montage et du projet, chef de projet CEA

PROSECCO – Formally Proven Protections for Secured Compiled Code

Programme :	ANR, Appel à projets génériques 2015
Période :	2016-2019
Partenaires :	CEA, LIP6
P.c.i. :	Emmanuelle Encrenaz (LIP6), Karine Heydemann (coordinatrice, LIP6), Quentin Meunier (LIP6), Pierre-Alain Moëllic, Bruno Robisson
Moyens financés :	36 HM doctorant et temps d'encadrement associé à la thèse
Faits marquants :	Application automatisée du polymorphisme de code à la compilation [Bel+19], application automatisée d'une contre-mesure de masquage [Bel+20a] et lien avec l'analyse formelle de robustesse du code automatiquement protégé
Implication :	Montage projet, responsable du WP3 (*Automatic insertion of protections*)

L4S – Learning for security in IoT

Programme :	Carnot Explorer Leti
Période :	2016-2017
Partenaires :	CEA DACLE, CEA Leti
P.c.i. :	Pierre-Alain Moëllic, Anca Molnos (coordinatrice), Florian Pebay-Peyroula
Moyens financés :	12 HM post-doc
Faits marquants :	Étude sur les techniques d'apprentissage pour la détection d'attaques sur systèmes embarqués contraints [Ker+18]
Implication :	Montage projet, contributeur

CLAPs – Méthodes et outils pour le déploiement massif de solutions de sécurisation pour l'IoT

Programme :	IRT Pulse
Période :	2017, 2018-2020
Partenaires :	CEA, VERIMAG, INRIA Corse
P.c.i. :	Yliès Falcone (INRIA), Laurent Maingault, Anca Molnos, Laurent Mounier (VERIMAG), Marie-Laure Potet (VERIMAG)
Moyens financés :	36 HM CDD, temps personnel de recherche
Faits marquants :	Maturation technologique de la chaîne de compilation Cogito ; évaluations de la robustesse aux fautes de protections logicielles appliquées à des composants sur étagère ; évaluation du potentiel des attaques par canal auxiliaire exploitant des réseaux de neurones profonds sur des traces de grande taille [Mas+20]
Implication :	Coordination du montage et du projet

Serene-IoT – Secured and Energy Efficient Healthcare Solutions for the IoT Market

Programme :	PENTA
Période :	2016-2019
Partenaires :	CEA, CHUGA (Hôpital de Grenoble), IDEMIA (ex-Morpho), LCIS, Orange Labs, STMicroelectronics, UAB, ...
P.c.i. :	Alberto Battistello (IDEMIA), Armand Castillejo (coordinateur, STM), David Hely (LCIS)
Moyens financés :	60 HM CDD (36 HM pourvus), temps personnel de recherche
Faits marquants :	Exécution polymorphe de code chiffré pour les applications embarquées [MCH21]
Implication :	Montage de projet, coordination du WP4 (*Software Factory*, 10 partenaires, 50 H.A.)

COFFI – Intégrité du flot d'exécution du logiciel à la micro-architecture

Programme :	ANR AAPG 2018
Période :	2019-2023
Partenaires :	CEA, EMSE, INVIA, LIP6
P.c.i. :	Jean-Max Dutertre (EMSE), Karine Heydemann (LIP6), Philippe Jaillon (EMSE), Pierre-Alain Moëllic, Olivier Potin (coordinateur, EMSE), Jean-Baptiste Rigaud (EMSE), André Sintzoff (INVIA)
Moyens financés :	36 HM doctorant et temps d'encadrement associé à la thèse
Faits marquants :	Solution mixte matérielle-logicielle pour la protection de la micro-architecture des processeurs embarqués contre les attaques par injection de fautes [CCH22a ; CCH23].
Implication :	Montage de projet, contributeur

Pulse-IA – Cybersécurité et technologies de liaison

Programme :	IRT NanoElec – programme Pulse
Période :	2020-2025
Partenaires :	CEA, INRIA, STMicroelectronics, VERIMAG, ...
P.c.i. :	Pierre-Alain Moëllic
Moyens financés :	36 HM CDD, temps personnel de recherche
Faits marquants :	
Implication :	Responsable de tâche

IDROMEL – Improving the Design of Secure Systems by a Reduction of Micro-architectural Effects on Side-Channel Attacks

Programme :	ANR AAPG 2020
Période :	2021-2025
Partenaires :	Arm, CEA, IRISA, LAAS, LIP6
P.c.i. :	Nicolas Belleville, Benoît Gérard (IRISA), Arnaud de Grandmaison (Arm), Annelie Heuser (IRISA), Karine Heydemann (LIP6, Thalès), Quentin Meunier (LIP6), Vincent Migliore (coordinateur, LAAS)
Moyens financés :	36 HM doctorant et temps d'encadrement associé à la thèse
Faits marquants :	
Implication :	Montage de projet, responsable de lot, chef de projet CEA

D-IIoT – Model-based Analysis for Dependable Evolving IIoT Applications

Programme :	UGA Persyval
Période :	2021-2024
Partenaires :	CEA, INRIA, LIG, VERIMAG
P.c.i. :	Yliès Falcone (INRIA/LIG), Stéphane Mocanu (INRIA/LIG), Laurent Mounier (VERIMAG), Gwen Salaün (coordinateur, INRIA/LIG)
Moyens financés :	aucun
Faits marquants :	
Implication :	Montage de projet, chef de projet CEA, participation à l'encadrement d'un stage et d'un CDD à Verimag

FlexSecurity – Flexsible System Hardening for Cybersecurity

Programme :	Carnot List
Période :	2021-2024
Partenaires :	CEA
P.c.i. :	Sébastien Bardin (coordinateur), Nicolas Belleville, Alexis Olivereau, Julien Signolès
Moyens financés :	36 HM doctorant, 24 HM CDD, temps personnel de recherche
Faits marquants :	
Implication :	Montage de projet, chef de projet pour le département DSCIN, contributeur

LOTR – a novel flow for designing highly customized RISC-V processor microarchitectures for embedded and IoT platforms

Programme :	ANR AAPG 2023
Période :	2023-2027
Partenaires :	CEA, IRISA
P.c.i. :	Steven Derrien (coordinateur, IRISA), Mathieu Jan, Isabelle Puaut (IRISA), Simon Rokicki (IRISA)
Moyens financés :	30 HM CDD et temps d'encadrement associé
Faits marquants :	
Implication :	Montage de projet, responsable de lot, chef de projet CEA

5.3 Liste des publications

Revues internationales avec comité de lecture

[Cas+23] Lorenzo Casalino, Nicolas Belleville, Damien Couroussé et Karine Heydeman. "A Tale of Resilience : On the Practical Security of Masked Software Implementations". In : *IEEE Access* (2023). doi : 10.1109/ACCESS.2023.3298436.

[CCH23b] Thomas Chamelot, Damien Couroussé et Karine Heydemann. "MAFIA : Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks". In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2023). doi : 10.1109/TCAD.2023.3276507.

[MCH21] Lionel Morel, Damien Couroussé et Thomas Hiscock. "Code Polymorphism Meets Code Encryption : Confidentiality and Side-Channel Protection of Software Components". In : *ACM Digital Threats : Research and Practice (DTRAP)* (2021). doi : 10.1145/3487058.

[Bel+20a] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Qentin Meunier et Ines Ben El Ouahma. "Maskara : Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation". In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2020). doi : 10.1109/TCAD.2020.3012237.

[Bel+19] Nicolas Belleville, Damien Couroussé, Karine Heydemann et Henri-Pierre Charles. "Automated Software Protection for the Masses against Side-Channel Attacks". In : *ACM Transactions on Architecture and Code Optimization (TACO)* (2019). doi : 10.1145/3281662.

[Luc+09] Annie Luciani, Jean-Loup Florens, Damien Couroussé et Julien Castet. "Ergotic Sounds : A New Way to Improve Playability, Believability and Presence of Virtual Musical Instruments". In : *Journal of New Music Research* 38.3 (2009), p. 309-323. doi : 10.1080/09298210903359187.

Ouvrages scientifiques, chapitres de livres avec comité de lecture

[Bel+18b] Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderramane Seriai et Henri-Pierre Charles. "Automatic Application of Software Countermeasures against Physical Attacks". In : *Cyber-Physical Systems Security*. Springer, 2018. doi : 10.1007/978-3-319-98935-8_7.

[CLC13] Damien Couroussé, Victor Lomüller et Henri-Pierre Charles. “Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform”. In : *Smart Multicore Embedded Systems*. Springer, 2013. doi : 10.1007/978-1-4614-8800-2_6.

[Cou07a] Damien Couroussé. “Haptic Board”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 126-127. isbn : 978-2-9530856-0-0.

[Cou07b] Damien Couroussé. “Mechanical Impedance”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 194-196. isbn : 978-2-9530856-0-0.

[Cou07c] Damien Couroussé. “Motion Capture”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 201-203. isbn : 978-2-9530856-0-0.

[CF07b] Damien Couroussé et Jean-Loup Florens. “Cobot”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 36-37. isbn : 978-2-9530856-0-0.

[CG07] Damien Couroussé et Jorge Juan Gil. “Contact Interaction”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 55-56. isbn : 978-2-9530856-0-0.

[Luc+07a] Annie Luciani, Damien Couroussé, Matthieu Evrard et Nicolas Castagné. “Gesture, Movement, Action”. In : *Enaction and Enactive Interfaces : A Handbook of Terms*. 2007, p. 4-5. isbn : 978-2-9530856-0-0.

Conférences internationales avec comité de sélection et actes

[Sel+24] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé et Sebastien Bardin. “Inference of Robust Reachability Constraints”. In : *POPL*. 2024. doi : 10.1145/3632933.

[Tol+23] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann et Mathieu Jan. “µARCHIFI : Formal Modeling and Verification Strategies for Microarchitectural Fault Injections”. In : *FMCAD*. 2023. doi : 10.34727/2023/isbn.978-3-85448-060-0_18.

[CCH22a] Thomas Chamelot, Damien Couroussé et Karine Heydeman. “SCI-FI – Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks”. In : *DATE*. 2022. doi : 10.23919/DATE54114.2022.9774685.

[Tol+22c] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann et Mathieu Jan. “Exploration of Fault Effects on Formal RISC-V Microarchitecture Models”. In : *FDTC*. 2022. doi : 10.1109/FDTC57191.2022.00017.

[Bel+21] Nicolas Belleville, Damien Couroussé, Emmanuelle Encrenaz, Karine Heydeman et Qentin Meunier. “PROSECCO : Formally-Proven Secure Compiled Code”. In : *C&ESAR*. 2021. url : https://www.cesar-conference.org.

[Bel+20b] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Quentin Meunier et Inès Ben El Ouama. “Maskara : Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation”. In : *CASES*. 2020. doi : 10.1109/TCAD.2020.3012237.

[Mas+20] Loïc Masure, Nicolas Belleville, Eleonora Cagli, Marie-Angela Cornélie, Damien Couroussé, Cécile Dumas et Laurent Maingault. “Deep Learning Side-Channel Analysis on Large-Scale Traces – a Case Study on a Polymorphic AES”. In : *ESORICS*. 2020. doi : 10.1007/978-3-030-58951-6_22.

[MC19b] Lionel Morel et Damien Couroussé. “Idols with Feet of Clay : On the Security of Bootloaders and Firmware Updaters for the IoT”. In : *NEWCAS*. 2019. doi : 10.1109/NEWCAS44328.2019.8961216.

[Ker+18] Sanaa Kerroumi, Damien Couroussé, Florian Pebay-Peyroula, Mohammed Ait Bendaoud et Anca Molnos. “On the Applicability of Binary Classification to Detect Memory Access Attacks in IoT”. In : *C&ESAR*. 2018. url : https://www.cesar-conference.org.

[Abd+17] Karim M. Abdellatif, Damien Couroussé, Olivier Potin et Philippe Jaillon. “Filtering-Based CPA : A Successful Side-Channel Attack against Desynchronization Countermeasures”. In : *Workshop on Cryptography and Security in Computing Systems (CS2)*. 2017. doi : 10.1145/3031836.3031842.

[BCR16] Thierno Barry, Damien Couroussé et Bruno Robisson. “Compilation of a Countermeasure against Instruction-Skip Fault Attacks”. In : *CS2*. 2016. doi : 10.1145/2858930.2858931.

[Cou+16a] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Olivier Potin et Jean-Louis Lanet. “Runtime Code Polymorphism as a Protection against Side Channel Attacks”. In : *WISTP*. 2016. doi : 10.1007/978-3-319-45931-8_9.

[CQC16] Damien Couroussé, Caroline Quéva et Henri-Pierre Charles. “Approximate Computing with Runtime Code Generation on Resource-Constrained Embedded Devices”. In : *Workshop on Approximate Computing (WAPCO)*. 2016. url : http://wapco.inf.uth.gr.

[ECC16] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. “Pushing the Limits of Online Auto-Tuning : Machine Code Optimization in Short-Running Kernels”. In : *MCSoC*. 2016. doi : 10.1109/MCSoC.2016.11.

[Le +16a] Hélène Le Bouder, Thierno Barry, Damien Couroussé, Jean-Louis Lanet et Ronan Lashermes. “A Template Attack Against VERIFY PIN Algorithms”. In : *SECRYPT*. 2016. doi : 10.5220/0005955102310238.

[Le +16b] Hélène Le Bouder, Ronan Lashermes, Jean-Louis Lanet, Thierno Barry et Damien Couroussé. “IoT and Physical Attacks”. In : *C&ESAR*. 2016. url : https://www.cesar-conference.org.

[QCC16] Caroline Quéva, Damien Couroussé et Henri-Pierre Charles. “Self-Optimisation Using Runtime Code Generation for Wireless Sensor Networks”. In : *ICDCN*. 2016. doi : 10.1145/2833312.2849557.

[ECC15a] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. “Micro-Architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT”. In : *RAPIDO*. 2015. doi : 10.1145/2693433.2693440.

[Cha+14b] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo et Rémy Gauguey. “deGoal a Tool to Embed Dynamic Code Generators into Applications”. In : *Compiler Construction*. 2014. doi : 10.1007/978-3-642-54807-9_6.

[Cou+14] Damien Couroussé, Bruno Robisson, Jean-Louis Lanet, Thierno Barry, Hassan Noura, Philippe Jaillon et Philippe Lalevée. “COGITO : Code Polymorphism to Secure Devices”. In : *SECRYPT*. 2014. doi : 10.5220/0005113704510456.

[ECC14] Fernando Endo, Damien Couroussé et Henri-Pierre Charles. “Micro-Architectural Simulation of in-Order and out-of-Order ARM Microprocessors with Gem5”. In : *SAMOS*. 2014. doi : 10.1109/SAMOS.2014.6893220.

[GCA14] Thierry Goubier, Damien Couroussé et Selma Azaiez. "τC : C with Process Network Extensions for Embedded Manycores". In : *International Conference on Computational Science*. 2014. doi : 10.1016/j.procs.2014.05.099.

[AC13] Charles Aracil et Damien Couroussé. "Software Acceleration of Floating-Point Multiplication Using Runtime Code Generation". In : *ICEAC*. 2013. doi : 10.1109/ICEAC.2013.6737630.

[Luc+08] Annie Luciani, Sile O'Modhrain, Charlotte Magnusson, Jean-Loup Florens et Damien Couroussé. "Perception of Virtual Multi-Sensory Objects : Some Musings on the Enactive Approach". In : *Cyberworlds*. 2008. doi : 10.1109/CW.2008.107.

[Luc+07b] Annie Luciani, Jean-Loup Florens, Damien Couroussé et Claude Cadoz. "Ergotic Sounds : A New Way to Improve Playability, Believability and Presence of Virtual Musical Instruments". In : *Proceedings of Enactive/07, 4th International Conference on Enactive Interfaces*. 2007.

[Luc+07d] Annie Luciani, Sile O'modhrain, Charlotte Magnusson, Jean-Loup Florens et Damien Couroussé. "Perception of Virtual Multisensory Mobile Objects – Wandering around the Enactive Assumption". In : *Proceedings of Enactive/07, 4th International Conference on Enactive Interfaces*. 2007.

[CFL06] Damien Couroussé, Jean-Loup Florens et Annie Luciani. "Effects of Stiffness on Tapping Performance". In : *HAPTICS*. IEEE, 2006. doi : 10.1109/HAPTIC.2006.1627101.

[Cou+06] Damien Couroussé, Gunnar Jansson, Jean-Loup Florens et Annie Luciani. "Visual and Haptic Perception of Object Elasticity in a Squeezing Virtual Event". In : *EuroHaptics*. 2006. doi : 10.1109/HAPTICS.2006.157.

[Luc+06a] Annie Luciani, Matthieu Evrard, Nicolas Castagné, Damien Couroussé, Jean-Loup Florens et Claude Cadoz. "A Basic Gesture and Motion Format for Virtual Reality Multisensory Applications". In : *GRAPP*. 2006. doi : 10.5220/0001355803490356.

[CFL05] Damien Couroussé, Jean-Loup Florens et Annie Luciani. "Effects of Stiffness on Tapping Performance – Do We Rely on Force to Keep Synchronized along with a Metronome ?" In : *Proceedings of the Enactive05 Conference*. 2005.

Conférences ou workshops internationaux avec comité de sélection, sans actes

[CCH23a] Thomas Chamelot, Damien Couroussé et Karine Heydeman. *MAFIA : Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks*. TASER. CHES Forum. Prague, 2023. url : https://ches.iacr.org/2023/forum.php.

[Tol+22a] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydeman et Mathieu Jan. *Exploration of Fault Effects on Formal RISC-V Microarchitecture Models*. JAIF. Workshop. Valence, 2022. url : https://jaif.io/2022/programme.html.

[CCH21] Thomas Chamelot, Damien Couroussé et Karine Heydeman. *SCI-FI – Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks*. JAIF. Workshop. Paris, 2021. url : https://jaif.io/2021.

[Bel+18a] Nicolas Belleville, Damien Couroussé, Karine Heydeman et Henri-Pierre Charles. *Automated Software Protection for the Masses against Side-Channel Attacks*. PHISIC - Workshop on Practical Hardware Innovation in Security and Characterization. 2018. URL : http://phisic2018.emse.fr/program.php.

[Mor+18a] Lionel Morel, Damien Couroussé, Alberto Battistello, Eric Poiret, Victor Servant, Armand Castillefo, Olivier Caffin, Gudrun Neumann, David Hely et Philippe Genestier. *The Emergence of New IoT Threats and HealthCare Mobile Applications*. ADTC - Nanoelectronics, Applications, Design & Technology Conference. Grenoble, France, 2018. URL : http://tima.imag.fr/sls/dtc/.

[Bar+17] Thierno Barry, Damien Couroussé, Karine Heydeman et Bruno Robisson. *Automated Combination of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault Attacks*. European LLVM Developers Meeting. 2017. URL : http://llvm.org/devmtg/20ee17-03//2017/02/20/accepted-sessions.html.

[Bei+17] Edith Beigné, Ivan Miro-Panades, Alexandre Valentian et al. "L-Iot : A Flexible Energy Efficient Platform Targeting Wide Range ioT Applications". In : *DAC IP-Track Session, Best Presentation Award*. 2017.

[Bel+17] Nicolas Belleville, Thierno Barry, Abderramane Seriai, Damien Couroussé, Karine Heydemann, Henri-Pierre Charles et Bruno Robisson. *The Multiple Ways to Automate the Application of Software Countermeasures against Physical Attacks : Pitfalls and Guidelines*. Cyber-Physical Security Education Workshop. 2017. URL : http://koclab.cs.ucsb.edu/cpsed/.

[CC16] Henri-Pierre Charles et Damien Couroussé. "Compilation and Runtime Code Generation for Performance and Security in Embedded Systems". In : *Workshop Dynamic Compilation Everywhere (DCE), in Conjunction with CGO*. Barcelona, Spain, 2016.

[Cou16d] Damien Couroussé. *Génération de Code Au Runtime Pour La Sécurité Des Systèmes Embarqués*. Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des systèmes d'information (RESSI). Toulouse, France, mai 2016. URL : https://ressi2016.sciencesconf.org/.

[ECC15b] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. "Towards a Dynamic Code Generator for Run-Time Self-Tuning Kernels in Embedded Application". In : *Workshop Dynamic Compilation Everywhere, in Conjunction with the 10th HiPEAC Conference*. DCE '15. 2015.

[QCC15] Caroline Quéva, Damien Couroussé et Henri-Pierre Charles. "Self-Optimisation Using Runtime Code Generation for Wireless Sensor Networks". In : *Internet-of-Things Symposium, ESWeek 2015*. Amsterdam, 2015. URL : http://conference.cs.cityu.edu.hk/IoT/FinalProgram.html.

[GCA13] Thierry Goubier, Damien Couroussé et Selma Azaiez. *Programming the P2012 Manycore : τC*. Workshop "Platform 2012 / STHORM embedded many-core acceleration" associated with the DATE conference. Grenoble, France, 2013.

[CC12] Damien Couroussé et Henri-Pierre Charles. "Dynamic Code Generation : An Experiment on Matrix Multiplication". In : *Proceedings of the Work-in-Progress Session, LCTES 2012*. 2012. URL : http://lctes12.cs.purdue.edu/content/work-progress-session.

[LC12] Yves Lhuillier et Damien Couroussé. “Embedded System Memory Allocator Optimization Using Dynamic Code Generation”. In : *Workshop "Dynamic Compilation Everywhere", in Conjunction with the 7th HiPEAC Conference*. Paris, France, 2012.

[CCL11] Damien Couroussé, Henri-Pierre Charles et Yves Lhuillier. *Flexible and Performing Kernels Dynamically Generated with deGoal for Embedded Systems*. P2012 developers' conference. Grenoble, France, déc. 2011.

[Luc+06b] Annie Luciani, Matthieu Evrard, Damien Couroussé, Nicolas Castagné, Claude Cadoz et Jean-Loup Florens. *A Basic Gesture and Motion Format for Virtual Reality Multisensory Applications*. 2nd Enactive Workshop. Montreal, Canada, mai 2006.

[Mag+06] Charlotte Magnusson, Annie Luciani, Damien Couroussé, Roy Davies et Jean-Loup Florens. *Preliminary Test in a Complex Virtual Dynamic Haptic Audio Environment*. 2nd Enactive Workshop. Montreal, Canada, mai 2006.

Présentations invitées

[Cou19a] Damien Couroussé. *Securing Embedded Software with Compilers*. Leti Days, Cybersecurity workshop. Grenoble, France, juin 2019. URL : http://www.leti-innovation-days.com/Pages/LID2019/WORKSHOPS/FRIDAY_JUNE_28TH/Cybersecurity.aspx.

[Cou19b] Damien Couroussé. *Side-Channel Attacks and Software Countermeasures*. SILM - Security of Software / Hardware Interfaces. Summer School of the French "GdR Sécurité Informatique". Rennes, France, juill. 2019. URL : https://silm-school.inria.fr.

[Cou18] Damien Couroussé. *Tutorial : Side-channel Attacks on Embedded Systems*. SSPREW - Software Security, Protection, and Reverse Engineering Workshop ; in conjunction with ACSAC 2018. Puerto Rico, USA, déc. 2018. URL : http://www.pprew.org.

[Cou+18] Damien Couroussé, Thierno Barry, Bruno Robisson, Nicolas Belleville, Philippe Jaillon, Olivier Potin, Hélène Le Bouder, Jean-Louis Lanet et Karine Heydemann. “All Paths Lead to Rome : Polymorphic Runtime Code Generation for Embedded Systems”. In : *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems*. 2018. DOI : 10.1145/3178291.3178296.

[Cou17a] Damien Couroussé. *COGITO – Runtime Code Generation to Secure Devices*. Workshop Interdisciplinaire sur la Sécurité Globale – WISG 17. Paris, France, sept. 2017. URL : http://www.wisg.fr/programme.html.

[Cou17b] Damien Couroussé. *Compilation Pour La Sécurité Des Systèmes Embarqués*. ISSISP 2017 – 8th International Summer School on Information Security and Protection. Gif-sur-Yvette, France, juill. 2017. URL : https://issisp2017.github.io.

[Cou16b] Damien Couroussé. *Compilation for Cybersecurity in Embedded Systems*. Workshop SERTIF : Simulation pour l'Evaluation de la RobusTesse des applications embarquées contre l'Injection de Fautes. Grenoble, France, oct. 2016. URL : http://sertif-projet.forge.imag.fr/fr/pages/workshop.html.

[Cou16c] Damien Couroussé. *Compilation Pour La Sécurité Des Systèmes Embarqués*. Summer School Cyber in Bretagne. Rennes, France, juill. 2016. URL : https://project.inria.fr/cyberinbretagne.

[Cou15a] Damien Couroussé. *Génération de Code Au Runtime : Applications à La Performance et à La Sécurité Des Systèmes Embarqués*. Séminaire CEA LSL. Saclay, France, mars 2015. URL : http://sebastien.bardin.free.fr/seminaire.html.

[Cou15b] Damien Couroussé. *Runtime Code Generation for Performance and Security in Embedded Systems*. Séminaire sécurité des systèmes électroniques embarqués, DGA-IRISA. Rennes, France, oct. 2015. URL : http://securite-elec.irisa.fr/2015/list_fr.html.

Autres workshops, séminaires

[MC19a] Lionel Morel et Damien Couroussé. *Idols with Feet of Clay : On the Security of Bootloaders and Firmware Updaters for the IoT*. Journée thématique "Sécurité des systèmes électroniques et communicants", GDR Onde - GT5 CEM. Paris, Jussieu, mai 2019. URL : http://gdr-ondes.cnrs.fr/2019/02/14/journee-thematique-securite-des-systemes-electroniques-et-communicants-21-mai-2019-paris-jussieu.

[Mor+18b] Lionel Morel, Marie-Laure Potet, Damien Couroussé, Laurent Mounier et Laurent Maingault. *Towards Fault Analysis of Firmware Updaters*. JAIF – Journée thématique sur les attaques par injection de fautes. Sorbonne Univ., Paris, 2018. URL : https://jaif.io/2018/jaif.html.

[Cou16a] Damien Couroussé. *Compilation and Cybersecurity in Embedded Systems*. 11e rencontre de la communauté française de compilation. Aussois, France, sept. 2016. URL : http://compilfr.ens-lyon.fr/onzieme-rencontre-compilation.

[Cou14] Damien Couroussé. *COGITO : Runtime Code Generation to Secure Devices*. 8e rencontres de la communauté française de compilation. Nice, France, juill. 2014. URL : http://compilfr.ens-lyon.fr/huitiemes-rencontres-de-la-communaute-francaise-de-compilation.

Posters et papiers courts associés

[CCH22b] Thomas Chamelot, Damien Couroussé et Karine Heydeman. *SCI-FI : Control Signal, Code, and Control-Flow Integrity against Fault Injection Attacks*. RISC-V week. Poster Session. Paris, France, mai 2022.

[Tol+22b] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydeman et Mathieu Jan. *Formal Analysis of Fault Injection Effects on RISC-V Microarchitecture Models*. RISC-V week, Paris. Poster Session. Paris, France, mai 2022. URL : https://open-src-soc.org/2022-05/posters.html.

[Tre+18] Tiago Trevisan Jost, Geneviève Ndour, Damien Couroussé, Christian Fabre et Anca Molnos. *ApproxRISC : An Approximate Computing Infrastructure for RISC-V*. RISC-V Workshop in Barcelona. Poster. Mai 2018. URL : https://hal-cea.archives-ouvertes.fr/cea-01893469.

[Cou+16b] Damien Couroussé, Olivier Potin, Bruno Robisson, Thierno Barry, Karim Abdelatif, Philippe Jaillon, Hélène Le Bouder et Jean-Louis Lanet. *Génération de Code Au Runtime Pour La Sécurisation de Composants*. Workshop Interdisciplinaire sur la Sécurité Globale (WISG). Troyes, France, fév. 2016.

[BCR15] Thierno Barry, Damien Couroussé et Bruno Robisson. *Compiler-Based Countermeasure against Fault Attacks*. Workshop on Cryptographic Hardware and Embedded Systems (CHES), Saint-Malo. Poster Session. 2015.

[Cou+15] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Jean-Louis Lanet et Olivier Potin. *Runtime Code Polymorphism as a Protection against Physical Attacks*. Poster Session. Workshop on Cryptographic Hardware et Embedded Systems (CHES), Saint-Malo, sept. 2015.

Rapports de recherche

[ECC17] Fernando Akira Endo, Damien Couroussé et Henri-Pierre Charles. "Pushing the Limits of Online Auto-Tuning : Machine Code Optimization in Short-Running Kernels". In : *CoRR* abs/1707.04566 (2017). URL : http://arxiv.org/abs/1707.04566.

[Cou11] Damien Couroussé. *An Introduction Study on Low Power Strategies for the Dynamic Generation of Algorithmic Kernels*. Rapport Interne LIST/DACLE/2011-0603/DC. CEA, 2011.

[CCJ11] Damien Couroussé, Henri-Pierre Charles et Mathieu Jan. *Intermediate Representation Format for DeGoal*. Contribution to the SMECY Report about the Intermediate Representation Format. CEA-LIST, LaSTRE laboratory, juin 2011.

[Cad+07] Claude Cadoz, Daniela Favaretto, Jean-Loup Florens et Damien Couroussé. *Emblematic Enactive Scenario "Real Virtual Physical Cooperation"*. Deliverable D.EES.3. NoE Enactive Interfaces IST-2004-002114-ENACTIVE, nov. 2007.

[Cas+07a] Nicolas Castagné, Matthieu Evrard, Damien Couroussé, Jean-Loup Florens, Annie Luciani et Claude Cadoz. *Guidelines on Data Formats for Structuring and Encoding Low-Level Gesture-Related Signals in Enactive Interfaces*. Deliverable D.RD1.4.4. NoE Enactive Interfaces IST-2004-002114-ENACTIVE, 2007.

[Flo+07] Jean-Loup Florens, Annie Luciani, Damien Couroussé et al. "State of the Art on Existing Sensors/Actuators Technologies for Haptic Interfaces". In : sous la dir. de Jean Loup Florens et Damien Couroussé. T. D3.1. ENACTIVE Interfaces NoE, 2007, p. 257.

[Cou04] Damien Couroussé. *Temporal Delays in Action-Vision Loop*. Deliverable D4b.1. ENACTIVE Interfaces NoE, sept. 2004.

[LC04a] Annie Luciani et Damien Couroussé. *Action Processing. Typology of Actions and Link with the Vision. Gesture Recognition*. Deliverable D4b.1. ENACTIVE Interfaces NoE, sept. 2004.

[LC04b] Annie Luciani et Damien Couroussé. *INPG State of the Art on Haptics, Audio and Vision. Cognition for Reactive Interfaces in Flexible HCI*. Deliverable D5.1. ENACTIVE Interfaces NoE, déc. 2004.

Logiciels

[BC21] Nicolas Belleville et Damien Couroussé. *Logiciel Maskara Version 1.0. Dépôt APP IDDN.FR.001.250029.001.S.A.2021.000.10000*. 2021. URL : https://secure2.iddn.org/app.server/certificate/?sn=2021250029001&key=6182f178500170987df1e6cc67b1lang=fr.

[BC16] Thierno Barry et Damien Couroussé. *Logiciel CACHAÇA Version 1.3. Dépôt APP IDDN.FR.001.470020.000.S.P.2016.000.10000*. 2016.

[Cha+14a] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller et Fernando Endo. *Logiciel deGoal Version 0.0. Dépôt APP IDDN.FR.001.110020.000.S.C.2014.000.10400*. 2014.

Brevets

[CCH22c] Thomas Chamelot, Damien Couroussé et Karine Heydeman. "SciPher : Protection de la micro-architecture des processeurs contre les attaques par injection de fautes". FR2208801. 2022.

[BC18] Nicolas Belleville et Damien Couroussé. "Method for executing a function, by a microprocessor, secured by time desynchronisation". FR20180056781, WO2019FR51640. 2018. URL : https://worldwide.espacenet.com/publicationDetails/biblio?II=3&ND=3&adjacent=true&locale=en_EP&FT=D&date=20210602&CC=EP&NR=3827549A1&KC=A1.

[BCH17] Thierno Barry, Damien Couroussé et Karine Heydemann. "Method for executing a machine code of a secure function". ICG020511, FR20170053175, WO2018FR50678. 2017. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20181018&DB=&CC=WO&NR=2018189443A1&KC=A1&ND=5.

[Cou17c] Damien Couroussé. "Method of executing a machine code of a secure function". WO2018FR52263, FR20170058799. 2017. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20190328&DB=&locale=&CC=WO&NR=2019058047A1&KC=A1&ND=1.

[CHS17] Damien Couroussé, Thomas Hiscock et Olivier Savry. "Method of executing, by a microprocessor, a polymorphic binary code of a predetermined function". FR20160062780, WO2017FR53592. 2017. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20180628&DB=&CC=WO&NR=2018115650A1&KC=A1&ND=4.

[CHS16] Damien Couroussé, Thomas Hiscock et Olivier Savry. "Procédé d'exécution par un microprocesseur d'un code machine polymorphique d'une fonction prédéterminée". IGC020502. 2016. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20200319&DB=EPODOC&CC=US&NR=2020089919A1.

[CQY15a] Damien Couroussé, Caroline Quéva et Yves Lhuillier. "Method for executing a computer program with a parameterised function". WO2016FR51583, FR3038086 (A1). 2015. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20170105&DB=EPODOC&CC=WO&NR=2017001753A1&KC=A1&ND=4.

[CQY15b] Damien Couroussé, Caroline Quéva et Yves Lhuillier. "Method for executing a computer program with a parameterised function". WO2016FR51584, FR3038087 (A1). 2015. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20170105&DB=EPODOC&CC=WO&NR=2017001754A1&KC=A1&ND=4.

[NC14] Hassan Noura et Damien Couroussé. "Method of encryption with dynamic diffusion and confusion layers". FR20140061917, WO2015EP78372. 2014. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20160609&DB=EPODOC&CC=WO&NR=2016087520A1&KC=A1&ND=4.

[Cou13] Damien Couroussé. “Method of executing, by a microprocessor, a polymorphic binary code of a predetermined function”. FR20130059473, US2015095659. 2013. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20150402&DB=&CC=US&NR=2015095659A1&KC=A1&ND=4.

Vulgarisation scientifique et communication au grand public

[Int18] Intelligence~Online. “Odo Software Prototype Provides Solution to Sophisticated Cyber-Spying”. In : *Intelligence Online* (nov. 2018). URL : https://www.intelligenceonline.com/surveillance--interception/2018/11/27/odo-software-prototype-provides-solution-to-sophisticated-cyber-spying,108334233-art (visité le 13/12/2018).

[Lar16] David Larousserie. “Lutter Contre La Contrefaçon de Composants Électroniques”. In : *Le Monde* (déc. 2016). URL : http://www.lemonde.fr/sciences/article/2016/12/05/lutter-contre-la-contrefacon-de-composants-electroniques_5043610_1650684.html (visité le 06/09/2017).

Thèses encadrées et soutenues

[Cas24] Lorenzo Casalino. “(On) The Impact of the Micro-architecture on Countermeasures against Side-Channel Attacks”. Thèse de Doctorat. Grenoble, France : Université Sorbonne, jan. 2024.

[Cha22] Thomas Chamelot. “Sécurisation de l’exécution Des Applications Contre Les Attaques Par Injection de Fautes Par Une Contre-Mesure Intégrée Au Processeur”. Thèse de Doctorat. Grenoble, France : École doctorale Informatique, Télécommunications et Électronique – Sorbonne Université, nov. 2022.

[Bel19] Nicolas Belleville. “Compilation Pour l’application de Contre-Mesures Contre Les Attaques Par Canal Auxiliaire”. Thèse de Doctorat. Grenoble, France : École Doctorale Mathématiques Sciences et Technologies de l’Information Informatique (MSTII) – Université de Grenoble, nov. 2019.

[Bar17] Thierno Barry. “Sécurisation à La Compilation de Logiciels Contre Les Attaques En Fautes”. Thèse de Doctorat. Grenoble, France : École Doctorale Sciences Ingénierie Santé (SIS spécialité Microélectronique) – École des Mines de Saint-Étienne, nov. 2017.

[End15] Fernando Endo. “Génération Dynamique de Code Pour l’optimisation Énergétique”. Thèse de Doctorat. Grenoble, France : École Doctorale Mathématiques Sciences et Technologies de l’Information Informatique (MSTII) – Université de Grenoble, sept. 2015.

Bibliographie

[Aba+09] Martin Abadi, Mihai Budiu, Ulfar Erlingsson et Jay Ligatti. “Control-Flow Integrity Principles, Implementations, and Applications”. In : *ACM Transactions on Information and System Security* (2009). doi : 10.1145/1609956.1609960 (cf. p. 48).

[Abd+17] Karim M. Abdellatif, Damien Couroussé, Olivier Potin et Philippe Jaillon. “Filtering-Based CPA : A Successful Side-Channel Attack against Desynchronization Countermeasures”. In : *Workshop on Cryptography and Security in Computing Systems (CS2)*. 2017. doi : 10.1145/3031836.3031842 (cf. p. 23).

[ABP12] Giovanni Agosta, Alessandro Barenghi et Gerardo Pelosi. “A Code Morphing Methodology to Automate Power Analysis Countermeasures”. In : *DAC*. 2012. doi : 10.1145/2228360.2228376 (cf. p. 5).

[Ago+15] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi et Michele Scandale. “The MEET Approach : Securing Cryptographic Embedded Software Against Side Channel Attacks”. In : *TCAD* (2015). doi : 10.1109/TCAD.2015.2430320 (cf. p. 27).

[Alm+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt et Pierre-Yves Strub. “Jasmin : High-Assurance and High-Speed Cryptography”. In : *CCS*. 2017. doi : 10.1145/3133956.3134078 (cf. p. 59, 63).

[Alm+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira et Pierre-Yves Strub. “The Last Mile : High-Assurance and High-Speed Cryptographic Implementations”. In : *S&P*. 2020. doi : 10.1109/SP40000.2020.00028 (cf. p. 59, 63).

[Ama+11] Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy et Michael Tunstall. “Can Code Polymorphism Limit Information Leakage ?” In : *WISTP*. 2011. doi : 10.1007/978-3-642-21040-2_1 (cf. p. 5).

[Anc+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard et Rémi Tucoulou. “Nanofocused X-Ray Beam to Reprogram Secure Circuits”. In : *CHES*. 2017. doi : 10.1007/978-3-319-66787-4_9 (cf. p. 37, 56).

[ABP21] Francesco Antognazza, Alessandro Barenghi et Gerardo Pelosi. “Metis : An Integrated Morphing Engine CPU to Protect Against Side Channel Attacks”. In : *IEEE Access* (2021). doi : 10.1109/ACCESS.2021.3077977 (cf. p. 26).

[AC13] Charles Aracil et Damien Couroussé. “Software Acceleration of Floating-Point Multiplication Using Runtime Code Generation”. In : *ICEAC*. 2013. doi : 10.1109/ICEAC.2013.6737630 (cf. p. 16, 57, 72).

[BFG15] Josep Balasch, Sebastian Faust et Benedikt Gierlichs. “Inner Product Masking Revisited”. In : *EUROCRYPT*. 2015. doi : 10.1007/978-3-662-46800-5_19 (cf. p. 34).

[Bal+15] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz et François-Xavier Standaert. “On the Cost of Lazy Engineering for Masked Software Implementations”. In : *CARDIS*. 2015. doi : 10.1007/978-3-319-16763-3_5 (cf. p. 5, 33, 59).

[BGV11] Josep Balasch, Benedikt Gierlichs et Ingrid Verbauwhede. “An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-Bit MCUs”. In : *FDTC*. 2011. doi : 10.1109/FDTC.2011.9 (cf. p. 46).

[Bar+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi et Francesco Regazzoni. “Countermeasures against Fault Attacks on Software Implemented AES : Effectiveness and Cost”. In : *WESS*. 2010. doi : 10.1145/1873548.1873555 (cf. p. 54).

[Bar17] Thierno Barry. “Sécurisation à La Compilation de Logiciels Contre Les Attaques En Fautes”. Thèse de Doctorat. Grenoble, France : École Doctorale Sciences Ingénierie Santé (SIS spécialité Microélectronique) – École des Mines de Saint-Étienne, nov. 2017 (cf. p. 4, 6-8, 11-15, 59, 69).

[BC16] Thierno Barry et Damien Couroussé. *Logiciel CACHAÇA Version 1.3. Dépôt APP IDDN.FR.001.470020.000.S.P.2016.000.10000*. 2016 (cf. p. 6, 69).

[Bar+17] Thierno Barry, Damien Couroussé, Karine Heydeman et Bruno Robisson. *Automated Combination of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault Attacks*. European LLVM Developers Meeting. 2017. url : http://llvm.org/devmtg/20ee17-03//2017/02/20/accepted-sessions.html (cf. p. 6, 69).

[BCH17] Thierno Barry, Damien Couroussé et Karine Heydemann. “Method for executing a machine code of a secure function”. ICG020511, FR20170053175, WO2018FR50678. 2017. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20181018&DB=&CC=WO&NR=2018189443A1&KC=A1&ND=5 (cf. p. 4, 6, 69).

[BCR15] Thierno Barry, Damien Couroussé et Bruno Robisson. *Compiler-Based Countermeasure against Fault Attacks*. Workshop on Cryptographic Hardware and Embedded Systems (CHES), Saint-Malo. Poster Session. 2015 (cf. p. 69).

[BCR16] Thierno Barry, Damien Couroussé et Bruno Robisson. “Compilation of a Countermeasure against Instruction-Skip Fault Attacks”. In : *CS2*. 2016. doi : 10.1145/2858930.2858931 (cf. p. 4, 6, 69).

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub et Rébecca Zucchini. “Strong Non-Interference and Type-Directed Higher-Order Masking”. In : *CCS*. 2016. doi : 10.1145/2976749.2978427 (cf. p. 28, 33, 59).

[Bat+23] Matthieu Baty, Pierre Wilke, Guillaume Hiet, Arnaud Fontaine et Alix Trieu. “A Generic Framework to Develop and Verify Security Mechanisms at the Microarchitectural Level : Application to Control-Flow Integrity”. In : *CSF*. 2023. doi : 10.1109/CSF57540.2023.00029 (cf. p. 62).

[Bea+15] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith et Louis Wingers. “The SIMON and SPECK Lightweight Block Ciphers”. In : *DAC*. 2015. doi : 10.1145/2744769.2747946 (cf. p. 32).

[Bel19] Nicolas Belleville. “Compilation Pour l’application de Contre-Mesures Contre Les Attaques Par Canal Auxiliaire”. Thèse de Doctorat. Grenoble, France : École Doctorale Mathématiques Sciences et Technologies de l’Information Informatique (MSTII) – Université de Grenoble, nov. 2019 (cf. p. 4, 15, 20, 29-32, 69).

[Bel+17] Nicolas Belleville, Thierno Barry, Abderramane Seriai, Damien Couroussé, Karine Heydemann, Henri-Pierre Charles et Bruno Robisson. *The Multiple Ways to Automate the Application of Software Countermeasures against Physical Attacks : Pitfalls and Guidelines*. Cyber-Physical Security Education Workshop. 2017. url : http://koclab.cs.ucsb.edu/cpsed/ (cf. p. 69, 71).

[BC18] Nicolas Belleville et Damien Couroussé. “Method for executing a function, by a microprocessor, secured by time desynchronisation”. FR20180056781, WO2019FR51640. 2018. url : https://worldwide.espacenet.com/publicationDetails/biblio?II=3&ND=3&adjacent=true&locale=en_EP&FT=D&date=20210602&CC=EP&NR=3827549A1&KC=A1 (cf. p. 69).

[BC21] Nicolas Belleville et Damien Couroussé. *Logiciel Maskara Version 1.0. Dépôt APP IDDN.FR.001.250029.001.S.A.2021.000.10000*. 2021. url : https://secure2.iddn.org/app.server/certificate/?sn=2021250029001&key=6182f178500170987df1e6cc67b lang=fr (cf. p. 15).

[Bel+21] Nicolas Belleville, Damien Couroussé, Emmanuelle Encrenaz, Karine Heydeman et Qentin Meunier. “PROSECCO : Formally-Proven Secure Compiled Code”. In : *C&ESAR*. 2021. url : https://www.cesar-conference.org (cf. p. 7, 61).

[Bel+18a] Nicolas Belleville, Damien Couroussé, Karine Heydeman et Henri-Pierre Charles. *Automated Software Protection for the Masses against Side-Channel Attacks*. PHISIC - Workshop on Practical Hardware Innovation in Security and Characterization. 2018. url : http://phisic2018.emse.fr/program.php (cf. p. 69).

[Bel+19] Nicolas Belleville, Damien Couroussé, Karine Heydemann et Henri-Pierre Charles. “Automated Software Protection for the Masses against Side-Channel Attacks”. In : *ACM Transactions on Architecture and Code Optimization (TACO)* (2019). doi : 10.1145/3281662 (cf. p. 4, 15, 20-26, 69, 75).

[Bel+20a] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Qentin Meunier et Ines Ben El Ouahma. “Maskara : Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2020). doi : 10.1109/TCAD.2020.3012237 (cf. p. 4, 15, 28, 29, 31, 32, 69, 75).

[Bel+20b] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Quentin Meunier et Inès Ben El Ouama. “Maskara : Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation”. In : *CASES*. 2020. doi : 10.1109/TCAD.2020.3012237 (cf. p. 69).

[Bel+18b] Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderramane Seriai et Henri-Pierre Charles. “Automatic Application of Software Countermeasures against Physical Attacks”. In : *Cyber-Physical Systems Security*. Springer, 2018. doi : 10.1007/978-3-319-98935-8_7 (cf. p. 15, 59, 69, 71).

[BM24] Nicolas Belleville et Loic Masure. “Combining Loop Shuffling and Code Polymorphism for Enhanced AES Side-Channel Security”. In : *COSADE*. 2024 (cf. p. 28).

[Ben21] Inès Ben El Ouahma. “Analyse de robustesse et sécurisation de codes assembleur contre des attaques physiques”. thèse de Doctorat. Paris : Sorbonne University, 2021 (cf. p. 32).

[Ben+19] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann et Emmanuelle Encrenaz. “Side-Channel Robustness Analysis of Masked Assembly Codes Using a Symbolic Approach”. In : *Journal of Cryptographic Engineering* (2019). doi : 10.1007/s13389-019-00205-7 (cf. p. 32).

[Bha+14] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley et Zakaria Najm. “NICV : Normalized Inter-Class Variance for Detection of Side-Channel Leakage”. In : *International Symposium on Electromagnetic Compatibility*. 2014. url : https://ieeexplore.ieee.org/document/6997167 (cf. p. 42).

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard et Robert Primas. “Power Contracts : Provably Complete Power Leakage Models for Processors”. In : *CCS*. 2022. doi : 10.1145/3548606.3560600 (cf. p. 34).

[Bré20] Jean-Baptiste Bréjon. “Quantification de la sécurité des applications en présence d’attaques physiques et détection de chemins d’attaques”. thèse de Doctorat. Paris : Sorbonne University, 2020 (cf. p. 7).

[BS20] Olivier Bronchain et François-Xavier Standaert. “Side-Channel Countermeasures’ Dissection and the Limits of Closed Source Security Evaluations”. In : *TCHES* (2020). doi : 10.13154/tches.v2020.i2.1-25 (cf. p. 25, 38).

[Bur+17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler et Mathias Payer. “Control-Flow Integrity : Precision, Security, and Performance”. In : *ACM Computing Surveys* (2017). doi : 10.1145/3054924 (cf. p. 48-50).

[Cab+22] Javier Cabrera Arteaga, Pierre Laperdrix, Martin Monperrus et Benoit Baudry. “Multi-Variant Execution at the Edge”. In : *Workshop on Moving Target Defense*. ACM, 2022. doi : 10.1145/3560828.3564007 (cf. p. 27).

[CDP17] Eleonora Cagli, Cécile Dumas et Emmanuel Prouff. “Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures”. In : *CHES*. Sous la dir. de Wieland Fischer et Naofumi Homma. 2017. doi : 10.1007/978-3-319-66787-4_3 (cf. p. 16).

[Car+12] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michael Quisquater et Matthieu Rivain. “Higher-Order Masking Schemes for S-Boxes”. In : *FSE*. Sous la dir. d’Anne Canteaut. 2012. doi : 10.1007/978-3-642-34047-5_21 (cf. p. 30).

[Cas24] Lorenzo Casalino. “(On) The Impact of the Micro-architecture on Countermeasures against Side-Channel Attacks”. Thèse de Doctorat. Grenoble, France : Université Sorbonne, jan. 2024 (cf. p. 4, 34).

[Cas+23] Lorenzo Casalino, Nicolas Belleville, Damien Couroussé et Karine Heydeman. “A Tale of Resilience : On the Practical Security of Masked Software Implementations”. In : *IEEE Access* (2023). doi : 10.1109/ACCESS.2023.3298436 (cf. p. 4, 70).

[CB23] Gaëtan Cassiers et Olivier Bronchain. “SCALib : A Side-Channel Analysis Library”. In : *Journal of Open Source Software* (2023). doi : 10.21105/joss.05196 (cf. p. 37).

[Cha22] Thomas Chamelot. “Sécurisation de l’exécution Des Applications Contre Les Attaques Par Injection de Fautes Par Une Contre-Mesure Intégrée Au Processeur”. Thèse de Doctorat. Grenoble, France : École doctorale Informatique, Télécommunications et Électronique – Sorbonne Université, nov. 2022 (cf. p. 4, 45, 52, 54, 69).

[CCH21] Thomas Chamelot, Damien Couroussé et Karine Heydeman. *SCI-FI – Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks*. JAIF. Workshop. Paris, 2021. URL : https://jaif.io/2021 (cf. p. 69).

[CCH22a] Thomas Chamelot, Damien Couroussé et Karine Heydeman. “SCI-FI – Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks”. In : *DATE*. 2022. DOI : 10.23919/DATE54114.2022.9774685 (cf. p. 45, 69, 76).

[CCH22b] Thomas Chamelot, Damien Couroussé et Karine Heydeman. *SCI-FI : Control Signal, Code, and Control-Flow Integrity against Fault Injection Attacks*. RISC-V week. Poster Session. Paris, France, mai 2022 (cf. p. 69).

[CCH22c] Thomas Chamelot, Damien Couroussé et Karine Heydeman. “SciPher : Protection de la micro-architecture des processeurs contre les attaques par injection de fautes”. FR2208801. 2022 (cf. p. 45, 55, 69).

[CCH23] Thomas Chamelot, Damien Couroussé et Karine Heydemann. “MAFIA : Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2023). DOI : 10.1109/TCAD.2023.3276507 (cf. p. 4, 45-50, 52-54, 69, 76).

[Cha+14a] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller et Fernando Endo. *Logiciel deGoal Version 0.0. Dépôt APP IDDN.FR.001.110020.000.S.C.2014.000.10400*. 2014 (cf. p. 16, 68).

[Cha+14b] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo et Rémy Gauguey. “deGoal a Tool to Embed Dynamic Code Generators into Applications”. In : *Compiler Construction*. 2014. DOI : 10.1007/978-3-642-54807-9_6 (cf. p. 1, 16, 18, 26, 57, 68, 74).

[CH01] Tzi-Cker Chiueh et Fu-Hau Hsu. “RAD : A Compile-Time Solution to Buffer Overflow Attacks”. In : *Proceedings 21st International Conference on Distributed Computing Systems*. 2001. DOI : 10.1109/ICDSC.2001.918971 (cf. p. 5).

[CCD00] Christophe Clavier, Jean-Sébastien Coron et Nora Dabbous. “Differential Power Analysis in the Presence of Hardware Countermeasures”. In : *CHES*. 2000. DOI : 10.1007/3-540-44499-8_20 (cf. p. 27).

[CPT18] Lucian Cojocar, Kostas Papagiannopoulos et Niek Timmers. “Instruction Duplication : Leaky and Not Too Fault-Tolerant !” In : *CARDIS*. Sous la dir. de Thomas Eisenbarth et Yannick Teglia. 2018. DOI : 10.1007/978-3-319-75208-2_10 (cf. p. 9).

[Col+22] Brice Colombier, Paul Grandamme, Julien Vernay, Émilie Chanavat, Lilian Bossuet, Lucie de Laulanié et Bruno Chassagne. “Multi-Spot Laser Fault Injection Setup : New Possibilities for Fault Injection Attacks”. In : *CARDIS*. 2022. DOI : 10.1007/978-3-030-97348-3_9 (cf. p. 56).

[Col+19] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud et Jean-Luc Danger. “Laser-Induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-Bit Microcontroller”. In : *HOST*. 2019. doi : 10.1109/HST.2019.8741030 (cf. p. 37, 56).

[CRV14] Jean-Sebastien Coron, Arnab Roy et Srinivas Vivek. “Fast Evaluation of Polynomials over Binary Finite Fields and Application to Side-channel Countermeasures”. In : *CHES*. 2014. doi : 10.1007/s13389-015-0099-9 (cf. p. 28, 30, 32).

[CK09] Jean-Sébastien Coron et Ilya Kizhvatov. “An Efficient Method for Random Delay Generation in Embedded Software”. In : *CHES*. 2009. doi : 10.1007/978-3-642-04138-9_12 (cf. p. 17, 23).

[CK10] Jean-Sébastien Coron et Ilya Kizhvatov. “Analysis and Improvement of the Random Delay Countermeasure of CHES 2009”. In : *CHES*. 2010. doi : 10.1007/978-3-642-15031-9_7 (cf. p. 17, 23).

[Cou13] Damien Couroussé. “Method of executing, by a microprocessor, a polymorphic binary code of a predetermined function”. FR20130059473, US2015095659. 2013. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20150402&DB=&CC=US&NR=2015095659A1&KC=A1&ND=4 (cf. p. 1, 4, 16, 75).

[Cou17] Damien Couroussé. “Method of executing a machine code of a secure function”. WO2018FR52263, FR20170058799. 2017. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20190328&DB=&locale=&CC=WO&NR=2019058047A1&KC=A1&ND=1 (cf. p. 45).

[Cou+18] Damien Couroussé, Thierno Barry, Bruno Robisson, Nicolas Belleville, Philippe Jaillon, Olivier Potin, Hélène Le Bouder, Jean-Louis Lanet et Karine Heydemann. “All Paths Lead to Rome : Polymorphic Runtime Code Generation for Embedded Systems”. In : *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems*. 2018. doi : 10.1145/3178291.3178296 (cf. p. 69).

[Cou+15] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Jean-Louis Lanet et Olivier Potin. *Runtime Code Polymorphism as a Protection against Physical Attacks*. Poster Session. Workshop on Cryptographic Hardware et Embedded Systems (CHES), Saint-Malo, sept. 2015 (cf. p. 69).

[Cou+16a] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Olivier Potin et Jean-Louis Lanet. “Runtime Code Polymorphism as a Protection against Side Channel Attacks”. In : *WISTP*. 2016. doi : 10.1007/978-3-319-45931-8_9 (cf. p. 4, 15, 18, 19, 21, 69, 72, 75).

[CC12] Damien Couroussé et Henri-Pierre Charles. “Dynamic Code Generation : An Experiment on Matrix Multiplication”. In : *Proceedings of the Work-in-Progress Session, LCTES 2012*. 2012. url : http://lctes12.cs.purdue.edu/content/work-progress-session (cf. p. 16).

[CHS17] Damien Couroussé, Thomas Hiscock et Olivier Savry. “Method of executing, by a microprocessor, a polymorphic binary code of a predetermined function”. FR20160062780, WO2017FR53592. 2017. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20180628&DB=&CC=WO&NR=2018115650A1&KC=A1&ND=4 (cf. p. 38).

[CLC13] Damien Couroussé, Victor Lomüller et Henri-Pierre Charles. “Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform”. In : *Smart Multicore Embedded Systems*. Springer, 2013. doi : 10.1007/978-1-4614-8800-2_6 (cf. p. 57).

[Cou+16b] Damien Couroussé, Olivier Potin, Bruno Robisson, Thierno Barry, Karim Abdelatif, Philippe Jaillon, Hélène Le Bouder et Jean-Louis Lanet. *Génération de Code Au Runtime Pour La Sécurisation de Composants*. Workshop Interdisciplinaire sur la Sécurité Globale (WISG). Troyes, France, fév. 2016 (cf. p. 23).

[CQC16] Damien Couroussé, Caroline Quéva et Henri-Pierre Charles. “Approximate Computing with Runtime Code Generation on Resource-Constrained Embedded Devices”. In : *Workshop on Approximate Computing (WAPCO)*. 2016. url : http://wapco.inf.uth.gr (cf. p. 57, 70, 75).

[CQY15a] Damien Couroussé, Caroline Quéva et Yves Lhuillier. “Method for executing a computer program with a parameterised function”. WO2016FR51583, FR3038086 (A1). 2015. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20170105&DB=EPODOC&CC=WO&NR=2017001753A1&KC=A1&ND=4 (cf. p. 16, 70).

[CQY15b] Damien Couroussé, Caroline Quéva et Yves Lhuillier. “Method for executing a computer program with a parameterised function”. WO2016FR51584, FR3038087 (A1). 2015. url : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20170105&DB=EPODOC&CC=WO&NR=2017001754A1&KC=A1&ND=4 (cf. p. 16, 70).

[Cou+14] Damien Couroussé, Bruno Robisson, Jean-Louis Lanet, Thierno Barry, Hassan Noura, Philippe Jaillon et Philippe Lalevée. “COGITO : Code Polymorphism to Secure Devices”. In : *SECRYPT*. 2014. doi : 10.5220/0005113704510456 (cf. p. 18, 19, 69, 70, 72, 75).

[CLH19] Valence Cristiani, Maxime Lecomte et Thomas Hiscock. “A Bit-Level Approach to Side Channel Based Disassembling”. In : *CARDIS*. 2019 (cf. p. 39).

[CH17] Ang Cui et Rick Housley. “BADFET : Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection”. In : *USENIX WOOT*. 2017. doi : 10.5555/3154768.3154771 (cf. p. 37).

[Dan+18] Jean-Luc Danger, Adrien Facon, Sylvain Guilley, Karine Heydemann, Ulrich Kuhne, Abdelmalek Si Merabet et Michael Timbert. “CCFI-Cache : A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity”. In : *DSD*. 2018. doi : 10.1109/DSD.2018.00093 (cf. p. 48).

[DBR20] Lesly-Ann Daniel, Sébastien Bardin et Tamara Rezk. “Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In : *S&P*. 2020. doi : 10.1109/SP40000.2020.00074 (cf. p. 63).

[DP08] Christophe De Cannière et Bart Preneel. “Trivium”. In : *New Stream Cipher Designs : The eSTREAM Finalists*. Springer, 2008. doi : 10.1007/978-3-540-68351-3_18 (cf. p. 40).

[De +17] Ruan De Clercq, Johannes Götzfried, David Übler, Pieter Maene et Ingrid Verbauwhede. “SOFIA : Software and Control Flow Integrity Architecture”. In : *Computers & Security* (2017). doi : 10.1016/j.cose.2017.03.013 (cf. p. 48).

[dHM22] Arnaud de Grandmaison, Karine Heydemann et Quentin L. Meunier. “ARMISTICE : Microarchitectural Leakage Modeling for Masked Software Formal Verification”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). doi : 10.1109/TCAD.2022.3197507 (cf. p. 34, 62).

[de +17] Wouter de Groot, Kostas Papagiannopoulos, Antonio de La Piedra, Erik Schneider et Lejla Batina. “Bitsliced Masking and ARM : Friends or Foes ?” In : *Lightweight Cryptography for Security and Privacy*. Springer, 2017. doi : 10.1007/978-3-319-55714-4_7 (cf. p. 33).

[Dur+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen et Philippe de Choudens. “FISSC : A Fault Injection and Simulation Secure Collection”. In : *SAFECOMP*. 2016. doi : 10.1007/978-3-319-45477-1_1 (cf. p. 9).

[Dur+13] François Durvaux, Mathieu Renauld, François-Xavier Standaert, Loic van Oldeneel tot Oldenzeel et Nicolas Veyrat-Charvillon. “Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models”. In : *CARDIS*. 2013. doi : 10.1007/978-3-642-37288-9_9 (cf. p. 17).

[End15] Fernando Endo. “Génération Dynamique de Code Pour l’optimisation Énergétique”. Thèse de Doctorat. Grenoble, France : École Doctorale Mathématiques Sciences et Technologies de l’Information Informatique (MSTII) – Université de Grenoble, sept. 2015 (cf. p. 1, 4, 16, 57, 68).

[ECC14] Fernando Endo, Damien Couroussé et Henri-Pierre Charles. “Micro-Architectural Simulation of in-Order and out-of-Order ARM Microprocessors with Gem5”. In : *SAMOS*. 2014. doi : 10.1109/SAMOS.2014.6893220 (cf. p. 57, 68).

[ECC15a] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. “Micro-Architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT”. In : *RAPIDO*. 2015. doi : 10.1145/2693433.2693440 (cf. p. 57, 68).

[ECC15b] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. “Towards a Dynamic Code Generator for Run-Time Self-Tuning Kernels in Embedded Application”. In : *Workshop Dynamic Compilation Everywhere, in Conjunction with the 10th HiPEAC Conference*. DCE ’15. 2015 (cf. p. 4, 26, 57, 68).

[ECC16] Fernando A. Endo, Damien Couroussé et Henri-Pierre Charles. “Pushing the Limits of Online Auto-Tuning : Machine Code Optimization in Short-Running Kernels”. In : *MCSoC*. 2016. doi : 10.1109/MCSoC.2016.11 (cf. p. 4, 57, 68).

[ECC17] Fernando Akira Endo, Damien Couroussé et Henri-Pierre Charles. “Pushing the Limits of Online Auto-Tuning : Machine Code Optimization in Short-Running Kernels”. In : *CoRR* abs/1707.04566 (2017). url : http://arxiv.org/abs/1707.04566 (cf. p. 1, 68).

[Fan+22] Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié et Olivier Potin. *Real-Time Frequency Detection to Synchronize Fault Injection on System-on-Chip*. 2022. url : https://eprint.iacr.org/2022/602 (visité le 05/01/2023) (cf. p. 37).

[Fum+11] Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff et Matthieu Rivain. “Affine Masking against Higher-Order Side Channel Analysis”. In : *SAC*. 2011. doi : 10.1007/978-3-642-19574-7_18 (cf. p. 17).

[Gen+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer et Yuval Yarom. “ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels”. In : *CCS*. 2016. doi : 10.1145/2976749.2978353 (cf. p. 27).

[GRD22] Jean-Michel Gorius, Simon Rokicki et Steven Derrien. “SpecHLS : Speculative Accelerator Design Using High-Level Synthesis”. In : *IEEE Micro* (2022). doi : 10.1109/MM.2022.3188136 (cf. p. 61).

[GCA14] Thierry Goubier, Damien Couroussé et Selma Azaiez. “τC : C with Process Network Extensions for Embedded Manycores”. In : *International Conference on Computational Science*. 2014. doi : 10.1016/j.procs.2014.05.099 (cf. p. 74).

[Gou+23] Théophile Gousselot, Olivier Thomas, Jean-Max Dutertre, Olivier Potin et Jean-Baptiste Rigaud. “Lightweight Countermeasures Against Original Linear Code Extraction Attacks on a RISC-V Core”. In : *HOST*. 2023. doi : 10.1109/HOST55118.2023.10133316 (cf. p. 39).

[Gro23] OpenHW Group. *GitHub - Openhwgroup/Core-v-Cores : CORE-V Family of RISC-V Cores*. 2023. url : https://github.com/openhwgroup/core-v-cores (visité le 18/12/2023) (cf. p. 54).

[HA22] Gregor Haas et Aydin Aysu. “Apple vs. EMA : Electromagnetic Side Channel Attacks on Apple CoreCrypto”. In : *DAC*. 2022. doi : 10.1145/3489517.3530437 (cf. p. 27).

[HSE22] Muhammad Hataba, Ahmed Sherif et Reem Elkhouly. “Enhanced Obfuscation for Software Protection in Autonomous Vehicular Cloud Computing Platforms”. In : *IEEE Access* (2022). doi : 10.1109/ACCESS.2022.3159249 (cf. p. 27).

[HOM06] Christoph Herbst, Elisabeth Oswald et Stefan Mangard. “An AES Smart Card Implementation Resistant to Power Analysis Attacks”. In : *ACNS*. 2006. doi : 10.1007/11767480_16 (cf. p. 17, 28, 29).

[HLB19] Karine Heydemann, Jean-François Lalande et Pascal Berthomé. “Formally Verified Software Countermeasures for Control-Flow Integrity of Smart Card C Code”. In : *Computers and Security* (2019). doi : 10.1016/j.cose.2019.05.004 (cf. p. 14).

[His17] Thomas Hiscock. “Microcontrôleur à Flux Chiffré d’instructions et de Données”. Thèse de Doctorat Spécialité Informatique. Grenoble, France : Université de Versailles Saint-Quentin en Yvelines, déc. 2017 (cf. p. 40).

[HSG19] Thomas Hiscock, Olivier Savry et Louis Goubin. “Lightweight Instruction-Level Encryption for Embedded Processors Using Stream Ciphers”. In : *Microprocessors and Microsystems* (2019). doi : 10.1016/j.micpro.2018.10.001 (cf. p. 38, 40).

[Joh+18] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune et Richard Ho. *Titan : Enabling a Transparent Silicon Root of Trust for Cloud*. 2018. url : https://old.hotchips.org/hc30/1conf/1.14_Google_Titan_GoogleFinalTitanHotChips2018.pdf (cf. p. 54).

[Ker+18] Sanaa Kerroumi, Damien Couroussé, Florian Pebay-Peyroula, Mohammed Ait Bendaoud et Anca Molnos. “On the Applicability of Binary Classification to Detect Memory Access Attacks in IoT”. In : *C&ESAR*. 2018. url : https://www.cesar-conference.org (cf. p. 71, 75).

[KS01] Seongwoo Kim et Arun K. Somani. “On-Line Integrity Monitoring of Microprocessor Control Logic”. In : *Microelectronics Journal* (2001). doi : 10.1016/S0026-2692(01)00090-8 (cf. p. 46).

[Ko+22] Yousun Ko, Alex Bradbury, Bernd Burgstaller et Robert Mullins. “Trace-and-Brace (TAB) : Bespoke Software Countermeasures against Soft Errors”. In : *LCTES*. 2022. doi : 10.1145/3519941.3535070 (cf. p. 11).

[Koo02] P. Koopman. “32-Bit Cyclic Redundancy Codes for Internet Applications”. In : *International Conference on Dependable Systems and Networks*. 2002. doi : 10.1109/DSN.2002.1028931 (cf. p. 53).

[LHB14] Jean-François Lalande, Karine Heydemann et Pascal Berthomé. “Software Countermeasures for Control Flow Integrity of Smart Card C Codes”. In : *ESORICS*. 2014. doi : 10.1007/978-3-319-11212-1_12 (cf. p. 10-12).

[Lau20] Johan Laurent. “Modélisation de fautes utilisant la description RTL de microarchitectures pour l’analyse de vulnérabilité conjointe matérielle-logicielle”. Thèse de doct. Université Grenoble Alpes, 2020 (cf. p. 46).

[Lau+19a] Johan Laurent, Vincent Beroulle, Christophe Deleuze et Florian Pebay-Peyroula. “Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures”. In : *DATE*. 2019. doi : 10.23919/DATE.2019.8715158 (cf. p. 47).

[Lau+19b] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula et Athanasios Papadimitriou. “Cross-Layer Analysis of Software Fault Models and Countermeasures Against Hardware Fault Attacks in a RISC-V Processor”. In : *Microprocessors and Microsystems* (2019). doi : 10.1016/j.micpro.2019.102862 (cf. p. 46).

[Lau+21] Johan Laurent, Christophe Deleuze, Florian Pebay-Peyroula et Vincent Beroulle. “Bridging the Gap between RTL and Software Fault Injection”. In : *Journal on Emerging Technologies in Computing Systems* (2021). doi : 10.1145/3446214 (cf. p. 46).

[Le +16a] Hélène Le Bouder, Thierno Barry, Damien Couroussé, Jean-Louis Lanet et Ronan Lashermes. “A Template Attack Against VERIFY PIN Algorithms”. In : *SECRYPT*. 2016. doi : 10.5220/0005955102310238 (cf. p. 15, 23, 69).

[Le +16b] Hélène Le Bouder, Ronan Lashermes, Jean-Louis Lanet, Thierno Barry et Damien Couroussé. “IoT and Physical Attacks”. In : *C&ESAR*. 2016. url : https://www.cesar-conference.org (cf. p. 69).

[LSB22] Gaetan Leplus, Olivier Savry et Lilian Bossuet. “Insertion of Random Delay with Context-Aware Dummy Instructions Generator in a RISC-V Processor”. In : *HOST*. 2022. doi : 10.1109/HOST54066.2022.9840060 (cf. p. 26).

[Ler09] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In : *Communications of the ACM* (2009). doi : 10.1145/1538788.1538814 (cf. p. 63).

[LC12] Yves Lhuillier et Damien Couroussé. “Embedded System Memory Allocator Optimization Using Dynamic Code Generation”. In : *Workshop "Dynamic Compilation Everywhere", in Conjunction with the 7th HiPEAC Conference*. Paris, France, 2012 (cf. p. 16, 57, 74).

[Lis+21] Oleksiy Lisovets, David Knichel, Thorben Moos et Amir Moradi. “Let’s Take It Offline : Boosting Brute-Force Attacks on iPhone’s User Authentication through SCA”. In : *Transactions on Cryptographic Hardware and Embedded Systems* (2021). doi : 10.46586/tches.v2021.i3.496-519 (cf. p. 27).

[LR21] Victor Lomné et Thomas Roche. *A Side Journey to Titan Side-Channel Attack on the Google Titan Security Key*. White Paper. NinjaLab, 2021. URL : https://ninjalab.io/a-side-journey-to-titan/ (cf. p. 38, 39).

[MOP07] Stefan Mangard, Elisabeth Oswald et Thomas Popp. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, 2007. DOI : 10.1007/978-0-387-38162-6 (cf. p. 17, 27).

[Man+22] Noura Ait Manssour, Vianney LapÔtre, Guy Gogniat et Arnaud Tisserand. "Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks". In : *DDECS*. 2022. DOI : 10.1109/DDECS54261.2022.9770170 (cf. p. 54).

[MPW22] Ben Marshall, Dan Page et James Webb. "MIRACLE : MIcRo-ArChitectural Leakage Evaluation : A Study of Micro-Architectural Power Leakage across Many Devices". In : *Transactions on Cryptographic Hardware and Embedded Systems* (2022). DOI : 10.46586/tches.v2022.i1.175-220 (cf. p. 34).

[Mas+20] Loïc Masure, Nicolas Belleville, Eleonora Cagli, Marie-Angela Cornélie, Damien Courousséé, Cécile Dumas et Laurent Maingault. "Deep Learning Side-Channel Analysis on Large-Scale Traces – a Case Study on a Polymorphic AES". In : *ESORICS*. 2020. DOI : 10.1007/978-3-030-58951-6_22 (cf. p. 4, 15, 24, 25, 28, 69, 76).

[Men+20] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud et Jean-Luc Danger. "Experimental Analysis of the Electromagnetic Instruction Skip Fault Model". In : *DTIS*. 2020. DOI : 10.1109/DTIS48698.2020.9081261 (cf. p. 9).

[MD19] Darius Mercadier et Pierre-Évariste Dagand. "Usuba : High-Throughput and Constant-Time Ciphers, by Construction". In : *PLDI*. 2019. DOI : 10.1145/3314221.3314636 (cf. p. 59).

[MOH20] Quentin L Meunier, Ines Ouahma Ben El et Karine Heydemann. "SELA : A Symbolic Expression Leakage Analyzer". In : *PROOFS*. 2020. URL : https://hal.science/hal-02983213 (cf. p. 31).

[MPH23] Quentin L. Meunier, Etienne Pons et Karine Heydemann. "LeakageVerif : Efficient and Scalable Formal Verification of Leakage in Symbolic Expressions". In : *IEEE Transactions on Software Engineering* (2023). DOI : 10.1109/TSE.2023.3252671 (cf. p. 31).

[Mon24] David Monniaux. "Memory Simulations, Security and Optimization in a Verified Compiler". In : *CPP*. Jan. 2024. DOI : 10.1145/3636501.3636952 (cf. p. 63).

[MSS09] Amir Moradi, Mohammad Taghi Manzuri Shalmani et Mahmoud Salmasizadeh. "Dual-Rail Transition Logic : A Logic Style for Counteracting Power Analysis Attacks". In : *Computers & Electrical Engineering* (2009). DOI : 10.1016/j.compeleceng.2008.06.004 (cf. p. 16).

[MC19a] Lionel Morel et Damien Courousséé. "Idols with Feet of Clay : On the Security of Bootloaders and Firmware Updaters for the IoT". In : *NEWCAS*. 2019. DOI : 10.1109/NEWCAS44328.2019.8961216 (cf. p. 37, 38, 71).

[MC19b] Lionel Morel et Damien Courousséé. *Idols with Feet of Clay : On the Security of Bootloaders and Firmware Updaters for the IoT*. Journée thématique "Sécurité des systèmes électroniques et communicants", GDR Onde - GT5 CEM. Paris, Jussieu, mai 2019. URL : http://gdr-ondes.cnrs.fr/2019/02/14/journee-thematique-securite-des-systemes-electroniques-et-communicants-21-mai-2019-paris-jussieu (cf. p. 71).

[Mor+18a] Lionel Morel, Damien Couroussé, Alberto Battistello, Eric Poiret, Victor Servant, Armand Castillefo, Olivier Caffin, Gudrun Neumann, David Hely et Philippe Genestier. *The Emergence of New IoT Threats and HealthCare Mobile Applications*. ADTC - Nanoelectronics, Applications, Design & Technology Conference. Grenoble, France, 2018. URL : http://tima.imag.fr/sls/dtc/ (cf. p. 71).

[MCH21] Lionel Morel, Damien Couroussé et Thomas Hiscock. "Code Polymorphism Meets Code Encryption : Confidentiality and Side-Channel Protection of Software Components". In : *ACM Digital Threats : Research and Practice (DTRAP)* (2021). DOI : 10.1145/3487058 (cf. p. 4, 20, 38-44, 71, 76).

[Mor+18b] Lionel Morel, Marie-Laure Potet, Damien Couroussé, Laurent Mounier et Laurent Maingault. *Towards Fault Analysis of Firmware Updaters*. JAIF – Journée thématique sur les attaques par injection de fautes. Sorbonne Univ., Paris, 2018. URL : https://jaif.io/2018/jaif.html (cf. p. 71).

[Mor+14a] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson et E. Encrenaz. "Experimental Evaluation of Two Software Countermeasures against Fault Attacks". In : *HOST*. 2014. DOI : 10.1109/HST.2014.6855580 (cf. p. 9, 13).

[Mor+14b] N. Moro, K. Heydemann, E. Encrenaz et B. Robisson. "Formal Verification of a Software Countermeasure against Instruction Skip Attacks". In : *JCEN* (2014). DOI : 10.1007/s13389-014-0077-7 (cf. p. 6-8).

[Mor+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson et Emmanuelle Encrenaz. "Electromagnetic Fault Injection : Towards a Fault Model on a 32-Bit Microcontroller". In : *FDTC*. 2013. DOI : 10.1109/FDTC.2013.9 (cf. p. 6, 46, 48).

[Mos+12] Andrew Moss, Elisabeth Oswald, Dan Page et Michael Tunstall. "Compiler Assisted Masking". In : *CHES*. 2012. DOI : 10.1007/978-3-642-33027-8_4 (cf. p. 28, 29, 59).

[Nas+22] Pascal Nasahl, Miguel Osorio, Pirmin Vogel, Michael Schaffner, Timothy Trippel, Dominic Rizzo et Stefan Mangard. "SYNFI : Pre-Silicon Fault Analysis of an Open-Source Secure Element". In : *Transactions on Cryptographic Hardware and Embedded Systems* (2022). DOI : 10.46586/tches.v2022.i4.56-87 (cf. p. 62).

[Nas+21] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed et Stefan Mangard. "CrypTag : Thwarting Physical and Logical Memory Vulnerabilities Using Cryptographically Colored Memory". In : *CCS*. 2021. DOI : 10.1145/3433210.3453684. arXiv : 2012.06761 [cs] (cf. p. 54).

[Nas+23] Pascal Nasahl, Martin Unterguggenberger, Rishub Nagpal, Robert Schilling, David Schrammel et Stefan Mangard. "SCFI : State Machine Control-Flow Hardening Against Fault Attacks". In : *DATE*. 2023. DOI : 10.23919/DATE56975.2023.10137038 (cf. p. 54).

[NC14] Hassan Noura et Damien Couroussé. "Method of encryption with dynamic diffusion and confusion layers". FR20140061917, WO2015EP78372. 2014. URL : https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20160609&DB=EPODOC&CC=WO&NR=2016087520A1&KC=A1&ND=4 (cf. p. 70).

[NC15a] Hassan Noura et Damien Couroussé. *HLDCA-WSN : Homomorphic Lightweight Data Confidentiality Algorithm for Wireless Sensor Network*. Cryptology ePrint Archive, Report 2015/928. 2015. URL : https://eprint.iacr.org/2015/928 (visité le 12/03/2024) (cf. p. 70).

[NC15b] Hassan Noura et Damien Couroussé. “Lightweight, Dynamic, and Flexible Cipher Scheme for Wireless and Mobile Networks”. In : *Ad Hoc Networks*. 2015. doi : 10.1007/978-3-319-25067-0_18 (cf. p. 70).

[OC14] Colin O’Flynn et Zhizhang (David) Chen. “ChipWhisperer : An Open-Source Platform for Hardware Embedded Security Research”. In : *COSADE*. 2014. doi : 10.1007/978-3-319-10175-0_17 (cf. p. 37).

[OSM02] N. Oh, P.P. Shirvani et E.J. McCluskey. “Control-Flow Checking by Software Signatures”. In : *IEEE Transactions on Reliability* (2002). doi : 10.1109/24.994926 (cf. p. 10, 46).

[Osw+13] David Oswald, Daehyun Strobel, Falk Schellenberg, Timo Kasper et Christof Paar. “When Reverse-Engineering Meets Side-Channel Analysis – Digital Lockpicking in Practice”. In : *SAC*. 2013. doi : 10.1007/978-3-662-43414-7_29 (cf. p. 38).

[Ozd+06] H. Ozdoganoglu, T.N. Vijaykumar, C.E. Brodley, B.A. Kuperman et A. Jalote. “SmashGuard : A Hardware Solution to Prevent Security Attacks on the Function Return Address”. In : *IEEE Transactions on Computers* (2006). doi : 10.1109/TC.2006.166 (cf. p. 50).

[Pén+20] Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy et Erven Rohou. “NOP-Oriented Programming : Should We Care ?” In : *SILM*. 2020. doi : 10.1109/EuroSPW51379.2020.00100 (cf. p. 9, 10).

[Pha+21] Thinh Hung Pham, Ben Marshall, Alexander Fell, Siew-Kei Lam et Daniel Page. “XDIVINSA : eXtended DIVersifying INStruction Agent to Mitigate Power Side-Channel Leakage”. In : *ASAP*. 2021. doi : 10.1109/ASAP52443.2021.00034 (cf. p. 26).

[Pro+17] Julien Proy, Karine Heydemann, Alexandre Berzati et Albert Cohen. “Compiler-Assisted Loop Hardening Against Fault Attacks”. In : *ACM Transactions on Architecture and Code Optimization (TACO)* (2017). doi : 10.1145/3141234 (cf. p. 11, 14, 15).

[QCC15] Caroline Quéva, Damien Couroussé et Henri-Pierre Charles. “Self-Optimisation Using Runtime Code Generation for Wireless Sensor Networks”. In : *Internet-of-Things Symposium, ESWeek 2015*. Amsterdam, 2015. url : http://conference.cs.cityu.edu.hk/IoT/FinalProgram.html (cf. p. 16, 57, 70, 75).

[Rei+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan et David I. August. “SWIFT : Software Implemented Fault Tolerance”. In : *CGO*. 2005. doi : 10.1109/CGO.2005.34 (cf. p. 5, 54).

[Ric+22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich et Tim Güneysu. “VERICA - Verification of Combined Attacks : Automated Formal Verification of Security against Simultaneous Information Leakage and Tampering”. In : *Transactions on Cryptographic Hardware and Embedded Systems* (2022). doi : 10.46586/tches.v2022.i4.255-284 (cf. p. 62).

[RP10] Matthieu Rivain et Emmanuel Prouff. “Provably Secure Higher-Order Masking of AES”. In : *CHES*. 2010. doi : 10.1007/978-3-642-15031-9_28 (cf. p. 17, 28, 30, 31).

[RPD09] Matthieu Rivain, Emmanuel Prouff et Julien Doget. “Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers”. In : *CHES*. 2009. doi : 10.1007/978-3-642-04138-9_13 (cf. p. 17).

[Riv+15] L. Rivière, Z. Najm, P. Rauzy, J. L. Danger, J. Bringer et L. Sauvage. “High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures”. In : *HOST*. 2015. doi : 10.1109/HST.2015.7140238 (cf. p. 46, 48).

[Ron+17] Eyal Ronen, Adi Shamir, Achi-Or Weingarten et Colin OFlynn. “IoT Goes Nuclear : Creating a ZigBee Chain Reaction”. In : *S&P*. 2017. doi : 10.1109/SP.2017.14 (cf. p. 37).

[SSG23] Manuel San Pedro, Victor Servant et Charles Guillemet. *LASCAR – Ledger's Advanced Side Channel Analysis Repository*. Ledger-Donjon. 2023. url : https://github.com/Ledger-Donjon/lascar (visité le 10/08/2023) (cf. p. 37).

[SEH20] Olivier Savry, Mustapha El-Majihi et Thomas Hiscock. “CONFIDAENT : CONtrol FLow Protection with Instruction and Data Authenticated ENcrypTion”. In : *DSD*. 2020. doi : 10.1109/DSD51259.2020.00048 (cf. p. 49, 54).

[Sch+21] Robert Schilling, Pascal Nasahl, Stefan Weiglhofer et Stefan Mangard. “SecWalk : Protecting Page Table Walks Against Fault Attacks”. In : *HOST*. 2021. doi : 10.1109/HOST49136.2021.9702269 (cf. p. 54).

[SWM18] Robert Schilling, Mario Werner et Stefan Mangard. “Securing Conditional Branches in the Presence of Fault Attacks”. In : *DATE*. 2018. doi : 10.23919/DATE.2018.8342268 (cf. p. 54).

[Sch+18] Robert Schilling, Mario Werner, Pascal Nasahl et Stefan Mangard. “Pointing in the Right Direction - Securing Memory Accesses in a Faulty World”. In : *ACSAC*. 2018. doi : 10.1145/3274694.3274728 (cf. p. 54).

[Sel+24] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé et Sebastien Bardin. “Inference of Robust Reachability Constraints”. In : *POPL*. 2024. doi : 10.1145/3632933 (cf. p. 62, 71).

[She+21] Carlton Shepherd, Konstantinos Markantonakis, Nico van Heijningen, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann et David Naccache. “Physical Fault Injection and Side-Channel Attacks on Mobile Devices : A Comprehensive Analysis”. In : *Computers & Security* (2021). doi : 10.1016/j.cose.2021.102471 (cf. p. 27).

[SP12] Daehyun Strobel et Christof Paar. “An Efficient Method for Eliminating Random Delays in Power Traces of Embedded Software”. In : *Information Security and Cryptology*. 2012. doi : 10.1007/978-3-642-31912-9_4 (cf. p. 17, 19).

[TAV02] K. Tiri, M. Akmal et I. Verbauwhede. “A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards”. In : *ESSCIRC*. 2002 (cf. p. 16).

[Tol+22a] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydeman et Mathieu Jan. *Exploration of Fault Effects on Formal RISC-V Microarchitecture Models*. JAIF. Workshop. Valence, 2022. url : https://jaif.io/2022/programme.html (cf. p. 70).

[Tol+22b] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydeman et Mathieu Jan. *Formal Analysis of Fault Injection Effects on RISC-V Microarchitecture Models*. RISC-V week, Paris. Poster Session. Paris, France, mai 2022. url : https://open-src-soc.org/2022-05/posters.html (cf. p. 70).

[Tol+22c] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann et Mathieu Jan. “Exploration of Fault Effects on Formal RISC-V Microarchitecture Models”. In : *FDTC*. 2022. doi : 10.1109/FDTC57191.2022.00017 (cf. p. 47, 53, 62, 70).

[Tol+23] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann et Mathieu Jan. “µARCHIFI : Formal Modeling and Verification Strategies for Microarchitectural Fault Injections”. In : *FMCAD*. 2023. doi : 10.34727/2023/isbn.978-3-85448-060-0_18 (cf. p. 62, 70).

[Tol+24] Simon Tollec, Vedad Hadžić, Pascal Nasahl, Mihail Asavoae, Roderick Bloem, Damien Couroussé, Karine Heydemann, Mathieu Jan et Stefan Mangard. *Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults*. 2024. url : https://eprint.iacr.org/2024/247 (visité le 16/02/2024) (cf. p. 60, 62, 70).

[Tre+18] Tiago Trevisan Jost, Geneviève Ndour, Damien Couroussé, Christian Fabre et Anca Molnos. *ApproxRISC : An Approximate Computing Infrastructure for RISC-V*. RISC-V Workshop in Barcelona. Poster. Mai 2018. url : https://hal-cea.archives-ouvertes.fr/cea-01893469 (cf. p. 72).

[Tro+21] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes et Guillaume Bouffard. “Electromagnetic Fault Injection against a Complex CPU, toward New Micro-Architectural Fault Models”. In : *Journal of Cryptographic Engineering* (2021). doi : 10.1007/s13389-021-00259-6 (cf. p. 37).

[Tua21] Son Tuan Vu. “Optimizing Property-Preserving Compilation”. Doctoral Thesis. Sorbonne University, 2021 (cf. p. 59).

[TWO14] Michael Tunstall, Carolyn Whitnall et Elisabeth Oswald. “Masking Tables—An Underestimated Security Risk”. In : *FSE*. Sous la dir. de Shiho Moriai. 2014. doi : 10.1007/978-3-662-43933-3_22 (cf. p. 28-30).

[UWM19] Thomas Unterluggauer, Mario Werner et Stefan Mangard. “MEAS : Memory Encryption and Authentication Secure against Side-Channel Attacks”. In : *Journal of Cryptographic Engineering* (2019). doi : 10.1007/s13389-018-0180-2 (cf. p. 54).

[vWB11] Jasper G. J. van Woudenberg, Marc F. Witteman et Bram Bakker. “Improving Differential Power Analysis by Elastic Alignment”. In : *CT-RSA*. 2011. doi : 10.1007/978-3-642-19074-2_8 (cf. p. 17, 19).

[Vey+12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof et François-Xavier Standaert. “Shuffling against Side-Channel Attacks : A Comprehensive Study with Cautionary Note”. In : *ASIACRYPT*. 2012. doi : 10.1007/978-3-642-34961-4_44 (cf. p. 17).

[Vu+21] Son Tuan Vu, Albert Cohen, Arnaud de Grandmaison, Christophe Guillon et Karine Heydemann. “Reconciling Optimization with Secure Compilation”. In : *OOPSLA*. 2021. doi : 10.1145/3485519 (cf. p. 59).

[Vu+20] Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison et Albert Cohen. “Secure Delivery of Program Properties through Optimizing Compilation”. In : *Compiler Construction*. 2020. doi : 10.1145/3377555.3377897 (cf. p. 59).

[Wer+18] Mario Werner, Thomas Unterluggauer, David Schaffenrath et Stefan Mangard. “Sponge-Based Control-Flow Protection for IoT Devices”. In : *EuroS&P*. 2018. doi : 10.1109/EuroSP.2018.00023 (cf. p. 48, 49, 55).

[WWM15] Mario Werner, Erich Wenger et Stefan Mangard. “Protecting the Control Flow of Embedded Processors against Fault Attacks”. In : *CARDIS*. 2015. doi : 10.1007/978-3-319-31271-2_10 (cf. p. 10, 46, 48, 49, 55).

[WS90] K. Wilken et J. P. Shen. “Continuous Signature Monitoring : Low-Cost Concurrent Detection of Processor Control Errors”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (1990). doi : 10.1109/43.55193 (cf. p. 48, 49, 55).

[Yuc+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick et Patrick Schaumont. “Software Fault Resistance Is Futile : Effective Single-Glitch Attacks”. In : *FDTC*. 2016. doi : 10.1109/FDTC.2016.21 (cf. p. 9, 46).

Application outillée de contre-mesures contre les attaques matérielles

Résumé

Ce mémoire d'habilitation porte sur la sécurisation des systèmes embarqués contre les attaques par canal auxiliaire et par injection de fautes. Les attaques par canal auxiliaire exploitent des observations physiques du circuit en fonctionnement pour inférer des informations additionnelles, non prévues dans un modèle d'attaquant en cryptanalyse pure, permettant d'altérer une propriété de sécurité, typiquement la confidentialité d'informations manipulées pendant un calcul. Les attaques par injection de fautes exploitent des perturbations physiques du circuit pour induire des effets inattendus dans son fonctionnement. Les conséquences logiques de ces perturbations matérielles permettent d'outrepasser la sécurité du système attaqué, par exemple faire fuir un secret ou obtenir des droits d'administration.

La première partie du mémoire s'intéresse au potentiel du compilateur pour l'application de contre-mesures logicielles contre les attaques matérielles. Cette question est traitée sous deux aspects. (i) Montrer qu'il est possible d'exploiter le compilateur pour l'application de contre-mesures. Cette question n'est pas triviale puisqu'un compilateur traditionnel s'intéresse seulement aux propriétés fonctionnelles du programme cible, alors que, pour l'application de contre-mesures, on s'intéresse à des propriétés de sécurité, non fonctionnelles. Cette question concerne également l'optimisation de la contre-mesure elle-même et du code ciblé par la protection. (ii) L'utilisation du compilateur comme levier pour la sécurisation de programmes : on exploite le compilateur pour augmenter le potentiel d'une contre-mesure ou pour la rendre paramétrable, donc rendre son application plus flexible. Le compilateur peut ainsi permettre l'implémentation de contre-mesures qui seraient autrement impossibles à mettre en œuvre.

Dans l'objectif d'adresser des modèles d'attaquants plus puissants, la deuxième partie du mémoire aborde la conception conjointe, logicielle et matérielle, de schémas de protection. Cette problématique est traitée sous deux aspects. (i) Comment exploiter des contre-mesures matérielles et les articuler avec des contre-mesures logicielles pour obtenir une protection à la couverture large permettant de couvrir un scenario d'attaque complet, depuis l'analyse préliminaire de la cible jusqu'à l'exploitation de techniques d'attaque spécifiques. (ii) Comment articuler un schéma de protection logiciel avec des contre-mesures matérielles pour obtenir un niveau de protection global élevé contre les injections de fautes, de manière à assurer l'intégrité complète du chemin d'instructions d'un processeur, en particulier les signaux de contrôle de sa micro-architecture.

Mots clés : sécurité matérielle, canal auxiliaire, injection de faute, compilation, micro-architecture des processeurs

---

Abstract

This dissertation deals with securing embedded systems against side-channel and fault injection attacks. Side-channel attacks exploit physical observations of the circuit in operation to infer additional information that is not exploitable in a pure cryptanalysis attacker model, making it possible to alter a security property, typically the confidentiality of information manipulated during a computation. Fault injection attacks exploit physical perturbations of the circuit to induce unexpected effects in its operation. The logical consequences of these perturbations can be used to override the security of the system under attack, e.g. to leak a secret or escalate privileges.

The first part of the dissertation focuses on the role of the compiler in a toolchain for securing software against hardware attacks, in particular, the compiler's potential for applying software countermeasures. This question is addressed from two angles. (i) Show that it is possible to rely on a (modified) compiler for the application of countermeasures. This question is not trivial since a traditional compiler is only interested in the functional properties of the target program, whereas the application of countermeasures deals with non-functional, security properties. This axis also concerns the optimization of the countermeasure itself and the code targeted by the protection. (ii) Using the compiler as a lever for program security: The aim is to increase the potential of a countermeasure, or to make the countermeasure parameterizable and therefore flexible in application. The compiler can also be used to support countermeasures that would otherwise be difficult to implement in practice.

With the aim of addressing more powerful attacker models, the second part of the dissertation addresses the joint design, in software and hardware, of protection schemes. This problem is addressed from two angles. (i) How to exploit hardware countermeasures and articulate them with software countermeasures to obtain full-coverage protection, enabling a complete attack scenario to be covered, from the target analysis phase to the exploitation of analysis techniques. (ii) How to articulate a software protection scheme with hardware protection elements to obtain a high level of protection against fault injections, so as to provide complete integrity of a processor's instruction path, in particular the control signals of its micro-architecture.

Keywords: hardware security, side-channel, fault injection, compilation, processor microarchitecture

---

CEA List, Dpt. Systèmes et Circuits Intégrés Numériques (DSCIN)
17 avenue des Martyrs – 38054 Grenoble Cedex – France