Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Analyse et renforcement de sécurité face aux attaques par canaux auxiliaires de composants matériels et logiciels protégés

---

Noé Amiot

Présentée et soutenue publiquement le *23 septembre 2026* devant un jury composé de :

Louis Goubin, Professeur, Université de Versailles Saint-Quentin-en-Yvelines	*Rapporteur*
Patrick Schaumont, Professeur, Worcester Polytechnic Institute	*Rapporteur*
Lesly-Ann Daniel, Enseignant-Chercheur, EURECOM	*Examinatrice*
Giorgio Di Natale, Directeur de recherche, Université Grenoble Alpes	*Examinateur*
Guénaël Renault, Chercheur HDR, ANSSI	*Examinateur*
Emmanuelle Encrenaz, Professeur, Sorbonne Université	*Directrice*
Karine Heydemann, Expert sécurité, Thales	*Directrice*
Quentin Meunier, Maître de conférences, Sorbonne Université	*Encadrant*

Sorbonne Université
Laboratoire d'Informatique de Paris 6 - LIP6
Place Jussieu, 75005 Paris, France

Ces travaux sont distribués sous licence **CC BY-NC-ND**
https://creativecommons.org/licenses/by-nc-nd/4.0/

À Hubert, Martine, Chloé, Victor et Eliott

Remerciements

Il est difficile de partir d'une page blanche pour remercier toutes les personnes qui ont contribué de manière directe ou indirecte à ces travaux. J'espère que ce manuscrit, dans lequel j'ai concentré avec exigence trois années de travail, rendra fiers ceux qui ont cru en moi et m'ont permis d'arriver jusqu'ici.

En premier lieu, je tiens à remercier Emmanuelle, Karine et Quentin, qui sont à l'origine du sujet de cette thèse, qui m'ont fait confiance pour le prendre en main et qui m'ont accompagné au quotidien pour le mener à bien. Vos profils se complètent tant sur le plan scientifique qu'humain, m'offrant un cadre idéal de recherche. Je n'aurais pas pu atteindre ce niveau de qualité sans vos très nombreuses relectures de ce manuscrit.

Je remercie ensuite les nombreux membres du LIP6 avec lesquels j'ai pu échanger. Nos discussions, liées ou non à la recherche, ont participé à rendre possible cette thèse. Bien qu'il y ait trop de personnes, je tiens particulièrement à remercier Adrien, Roselyne, Franck, Marie-Minerve, Alix et Julien. Je remercie aussi les doctorants (désormais docteurs pour certains) et stagiaires du laboratoire, particulièrement Thomas, Nathan, Maxime, Ali, Alexandre, Xunyue, Yacine, Rémi, Enrique, Nadhir, Corentin, Ivan et Lucie. Nos discussions quotidiennes permettent de maintenir l'effort sur la durée et d'explorer de nouveaux sujets. J'ai aussi une pensée particulière pour Pascal, Michaël et Guillaume, qui traversent ou ont récemment traversé des étapes similaires de leur vie.

I am thankful to Naofumi Homma and all the members of the Homma laboratory for welcoming me this summer in Japan. The friendly atmosphere made my stay truly enjoyable.

Je suis extrêmement reconnaissant envers ma famille pour son soutien indéfectible et pour s'être intéressée à ces travaux (j'imagine pour moi plutôt que par intérêt pour le sujet). Je remercie également Stéphane, qui m'a guidé sur cette voie et qui est à l'origine de ma passion pour le matériel. Pour leur soutien tout au long de ce parcours et aux différentes étapes qui y ont mené, je remercie aussi Albert ainsi qu'Alban, Adrien, Audrey, Cédric, Chloé, Florine, Léa-Lou, Lucas, Lucie, Lucien, Marie, Mathilde, Matthieu, Romain, Siméon, Victor et tant d'autres.

Je remercie finalement Lesly-Ann Daniel, Giorgio Di Natale et Guénaël Renault pour avoir accepté d'examiner cette thèse, ainsi que Louis Goubin et Patrick Schaumont pour m'avoir fait l'honneur de rapporter ce manuscrit.

Abstract

Side-channel attacks are a major threat to the security of systems, and more particularly of embedded systems performing cryptographic operations. These attacks exploit physical measurements such as power or electromagnetic emissions to recover secret data handled by the systems while they are operating. By targeting potential vulnerabilities of physical implementations at runtime instead of the cryptographic algorithms themselves, side-channel attacks make the recovery of cryptographic keys possible, even for algorithms that were proven resistant against cryptanalysis. Targeted systems range from cryptographic accelerators that perform a hard-coded cryptographic function to microcontrollers that execute any cryptographic function.

The main countermeasure against side-channel attacks is masking, which aims to remove the correlation between the handled data and the physical quantities that can be measured by transforming the computation. Sensitive data is split into shares using uniformly distributed randomly generated values and are always used in the computation in such a way that not all shares are used at the same time, thus removing the correlation between sensitive values and physical quantities. Masking is hard to implement but can be verified to be correctly implemented, in a given "leakage model". This model represents the information that can be retrieved by an attacker doing measurements on the running circuit. State-of-the-art leakage models account for hardware side-effects such as transitions and glitches, to obtain verification verdicts closer to the leakage observed in practice.

This PhD thesis focuses on the verification of masking of hardware accelerators and software executed by general-purpose processors. A new formal method of verification implemented in a tool named `aLEAKator` is presented, making use of "mixed-domain simulation" of hardware descriptions of circuits. `aLEAKator` is open-source, allows for the verification in various leakage models, is able to verify programs making use of precomputed tables and introduces the notion of verification granularity, previously missing from the state-of-the-art. The method is validated by reproducing existing verification verdicts from the state-of-the-art and by providing new results on software implementations and cycle-by-cycle comparison of verdict with real measurements of widely deployed processors. Finally, cryptographic functions that were not verified in complex leakage models in the past such as

AES masked with Herbst scheme are verified in this PhD thesis on five different processors with both RISC-V and ARMv7-M architectures.

Moreover, this PhD thesis provides an inventory of common hardware leakage sources on general-purpose processors. Additionally, countermeasures against these sources are provided and validated with real measurements. A method to automatically eradicate leakages from software executed on processors by making use of the detailed leakage nature and origin report of `aLEAKator` is then presented. This method is applied to the ARM Cortex-M4 processor and allows for the hardening of several software implementations in the presence of glitches and transitions. Notably, this method was used on the AES masked with the Herbst scheme, allowing for the first implementation, to the best of our knowledge, of this cipher on these leakage models.

Keywords : Side-channel attacks, Masking, Formal Verification, Micro-architectural leakage detection, Software rewriting

Résumé

Les attaques par canaux auxiliaires représentent une menace majeure pour la sécurité des systèmes informatiques et plus particulièrement des systèmes embarqués qui réalisent des opérations cryptographiques. Ces attaques exploitent des mesures de diverses grandeurs physiques telles que la consommation électrique ou les émissions électromagnétiques pour récupérer des informations sur les données manipulées par ces systèmes pendant leur fonctionnement. En ne ciblant pas les algorithmes cryptographiques mais d'éventuelles vulnérabilités dans leurs implémentations sur des cibles physiques, ces attaques permettent la récupération de clés cryptographiques même pour des algorithmes dont la résistance face aux cryptanalyses est prouvée. Les systèmes ciblés peuvent être des accélérateurs cryptographiques, dédiés à une implémentation cryptographique, ou des microcontrôleurs, qui peuvent exécuter des fonctions cryptographiques variées.

La contre-mesure principale face à ces attaques est le masquage, qui vise à supprimer la corrélation entre les données manipulées et les grandeurs physiques mesurées en transformant le calcul réalisé. Les valeurs sensibles sont transformées en « shares » à l'aide de valeurs aléatoires uniformément distribuées et seuls ces shares sont manipulés au cours du calcul, empêchant la corrélation entre les valeurs sensibles et les grandeurs physiques mesurées. L'application correcte du masquage peut être prouvée dans un « modèle de fuite », qui représente les éléments observables par un attaquant pendant les mesures sur le système ciblé. Les modèles de fuite de l'état de l'art prennent désormais en compte les effets matériels tels que les « glitches » et les transitions dans le but d'obtenir des verdicts plus fidèles vis-à-vis des fuites qui peuvent être observées en pratique.

Ces travaux se concentrent sur la vérification du masquage d'implémentations matérielles et d'implémentations logicielles exécutées sur des processeurs généralistes. Une nouvelle méthode formelle, implantée dans un outil distribué sous licence libre nommé `aLEAKator`, est présentée. Cette méthode repose sur la « simulation mixte de domaines » de la représentation matérielle des circuits. Les domaines permettent de représenter la propagation des valeurs sensibles du calcul à travers la simulation pour vérifier qu'un circuit valide, à tout instant, une propriété de sécurité dans un modèle de fuite donné. Cette méthode propose de réaliser des vérifications dans des modèles de fuite variés, permet la vérification en présence de tables précalculées et introduit la notion de granularité de vérification, absente

de l'état de l'art. L'approche a été validée en reproduisant les résultats de vérification de l'état de l'art ainsi qu'en proposant de nouveaux résultats pour des implémentations logicielles et des comparaisons, cycle par cycle, avec des mesures réelles sur des processeurs mondialement utilisés. Enfin, des fonctions cryptographiques qui n'avaient pas jusqu'à présent pu être vérifiées dans des modèles de fuite avancés, comme l'AES masqué avec le schéma de Herbst, ont été vérifiées sur cinq processeurs utilisant les jeux d'instructions RISC-V et ARMv7-M.

Dans un second temps, ces travaux proposent un inventaire des sources communes de fuite sur les processeurs généralistes. De plus, des contre-mesures pour chacune sont proposées et validées à l'aide de mesures réelles. Une méthode d'élimination des fuites au sein de programmes exécutés sur des processeurs, qui exploite les informations précises d'`aLEAKator` concernant la nature et l'origine des fuites, est ensuite proposée. Cette méthode est appliquée au processeur ARM Cortex-M4 et permet la sécurisation de plusieurs programmes dans un modèle de fuite prenant en compte les glitches et les transitions. Notamment, cette méthode a été appliquée à l'AES masqué avec le schéma de Herbst, permettant la conception de la première implémentation, à notre connaissance, de cet algorithme dans ce modèle de fuite.

Mots clés Attaques par canaux auxiliaires, Masquage, Vérification formelle, Détection de fuites micro-achitecturales, Réécriture de programmes

Table des matières

1 Introduction **1**

I Contexte, état de l'art et problématique **5**

2 Attaques par canaux auxiliaires et contre-mesures associées **7**
2.1 Contexte des attaques par canaux auxiliaires 7
2.1.1 Cibles des attaques 8
2.1.2 Mécanismes sous-jacents aux variations des grandeurs physiques . 9
2.1.3 Réalisation des attaques 11
2.2 Attaques par canaux auxiliaires exploitant la consommation 12
2.2.1 Attaques non profilées 13
2.2.2 Attaques profilées 15
2.3 Contre-mesures 16
2.3.1 Dissimulation 16
2.3.2 Masquage 19
2.3.3 Limitations du masquage 22

3 Vérification d'implémentations cryptographiques masquées **25**
3.1 Modèles de fuite 26
3.2 Méthodes de vérification 30
3.2.1 Analyses statistiques 30
3.2.2 Méthodes formelles 33
3.3 De l'importance de la largeur des supports 35
3.4 Comparaison des méthodes formelles de co-vérification logiciels-matériels de l'état de l'art 37
3.5 État de l'art des méthodes de renforcement automatique de la sécurité de programmes par transformation logicielle 39
3.5.1 Sécurisation en amont de l'exécution 39
3.5.2 Sécurisation en aval de l'exécution 40

4 Problématique et plan **43**

II Contributions **45**

5 Modélisation et vérification de micro-architectures et d'accélérateurs matériels 47
5.1 Modélisation et simulation de composants matériels 49
5.1.1 Modélisation des composants matériels 49
5.1.2 Simulation en domaines mixtes 52
5.1.3 Évaluations concrètes et symboliques 54
5.1.4 Évaluation de stabilité 55
5.1.5 Évaluation des ensembles de fuite 56
5.2 Orchestration de la vérification 58
5.2.1 Ensembles d'expressions à vérifier en cohérence avec un modèle de fuite 58
5.2.2 Ensembles réduits d'expressions à vérifier 60
5.3 Implémentation d'`aLEAKator` 63
5.3.1 Flux d'implémentation 63
5.3.2 Modifications apportées à `cxxrtl` 71
5.4 Résultats expérimentaux 72
5.4.1 Accélérateurs cryptographiques masqués 72
5.4.2 Implémentations logicielles masquées exécutées sur des processeurs 74
5.4.3 Ordre supérieur et autres propriétés de sécurité 87
5.4.4 Discussion 89
5.4.5 Conclusion 91

6 Sources de fuites, contre-mesures et résultats d'élimination des fuites 93
6.1 Sources communes de fuite et leurs contre-mesures 94
6.1.1 Fuites en valeur 95
6.1.2 Fuites liées à la mémoire 99
6.1.3 Fuites liées au banc de registres 107
6.1.4 Fuites liées à l'unité arithmétique et logique (ALU) 113
6.1.5 Instructions réalisant de multiples opérations 115
6.2 Système expert pour l'élimination automatique de fuites 115
6.2.1 Hypothèses de sécurisation 116
6.2.2 Stratégie de sécurisation 117
6.2.3 Conception de correctifs adaptés à une application automatique . . 122
6.2.4 Organisation des applications de correctif 126
6.3 Résultats de sécurisation pour le Cortex-M4 132
6.4 Limitations et horizon 133

III Conclusion **139**

7 Conclusion **141**
7.1 Perspectives 142
7.2 Publications 144
7.2.1 Articles 144
7.2.2 Communications 144
7.2.3 Productions 144

Bibliographie **145**

A Brève explication de l'AES masqué avec le schéma de Herbst **169**

B Fonctions de propagation dans les différents domaines **173**
B.1 Domaine concret et domaine symbolique 174
B.2 Domaine de la stabilité 175
B.3 Domaine des ensembles de fuite 176

C Complexité de l'exploitation de la stabilité pour le banc de registres, cas du jeu d'instructions RISC-V appliqué au processeur ibex **177**

D Correctifs pour le processeur Ibex **181**

1 Introduction

Dans un monde de plus en plus connecté, l'activité humaine repose plus que jamais sur les systèmes embarqués, qu'il s'agisse de microcontrôleurs, de cartes à puces ou de systèmes complexes comme les téléphones. Que ce soit à des fins de communication avec les cartes SIM, de paiement avec les cartes bancaires, d'identification avec les passeports, de régulation d'énergie avec les systèmes de distribution intelligents, de sécurité avec les *HSM* (*H*ardware *S*ecurity *M*odules), ou de qualité de vie avec la robotique, la santé, l'IoT (Internet des Objets), les téléphones, ces systèmes sont omniprésents.

La quantité croissante de données personnelles, et sensibles pour certaines, qui transitent quotidiennement entre ces systèmes et vers des serveurs rend vitale leur protection. Cette protection passe par l'utilisation d'algorithmes cryptographiques tels que *AES* (*A*dvanced *E*ncryption *S*tandard) ou *Dilithium* pour chiffrer ou authentifier ces données dans le but d'empêcher un acteur, qui observe des messages entre ces systèmes, d'en récupérer ou d'en falsifier le contenu. Afin d'obtenir des garanties de sécurité, la robustesse de ces algorithmes est mathématiquement prouvée pour résister aux cryptanalyses, qui visent à retrouver le contenu de messages sans connaissance préalable des clés cryptographiques.

Les systèmes embarqués réalisant des opérations cryptographiques sont devenus, depuis la fin des années 1990 [Koc96 ; KJJ99], les cibles d'une nouvelle catégorie d'attaques : les attaques physiques. Parmi celles-ci, les *attaques par canaux auxiliaires* reposent sur l'exploitation de grandeurs physiques observables pendant le fonctionnement d'un microcontrôleur dans le but d'obtenir des informations sur les données manipulées ou les instructions exécutées. Les attaques par *injection de fautes* reposent sur la perturbation des systèmes embarqués pendant leur fonctionnement pour outrepasser certains mécanismes de sécurité. Ces attaques observent des résultats intermédiaires de calculs, qui étaient supposés inaccessibles lors de la preuve des algorithmes cryptographiques, ou empêchent certaines étapes du calcul d'être réalisées correctement. Autrement dit, ces attaques ciblent non pas les algorithmes cryptographiques, mais les implémentations de ceux-ci pendant leur exécution.

Les applications de ces attaques sont multiples. De l'usurpation de titres d'identité, de moyens de paiement ou de communication, à la fraude rendue possible par la compromission de capteurs dans des objets du quotidien, la protection des systèmes embarqués face aux attaques physiques est devenue un enjeu de société concret. De nombreuses

contre-mesures et nouvelles attaques ont été proposées au cours des dernières décennies, faisant de la thématique des attaques physiques un domaine actif de recherche.

Les contre-mesures issues de ces recherches visant à améliorer la sécurité des systèmes embarqués sont implémentées par les industriels et requises par de multiples pays en raison de la criticité de leurs activités. Afin d'offrir un cadre international, des normes telles que les Critères Communs [CCR26] ont vu le jour pour évaluer et certifier les systèmes de sécurité face à divers types de menaces, y compris les attaques physiques. Ces normes progressent en se basant en retour sur les travaux de recherche pour rester pertinentes.

Les contre-mesures proposées face aux attaques physiques nécessitent d'être vérifiées pour apporter de la confiance en leur efficacité. Ces contre-mesures peuvent être physiques, matérielles ou logicielles et concernent aussi bien les accélérateurs matériels - des implémentations matérielles dédiées à un algorithme cryptographique et donc non programmables - que les microcontrôleurs.

La méthode la plus évidente pour évaluer la résistance face aux attaques par canaux auxiliaires consiste en la production des implémentations cryptographiques puis en de multiples tentatives d'attaques via des mesures réelles, afin de montrer la complexité de leur mise en place. Cette méthode, devant être réalisée avec le circuit final et n'offrant aucune garantie, est de plus en plus complétée par des méthodes statistiques et des méthodes formelles, raisonnant sur des simulations ou des modèles mathématiques. Les travaux de l'état de l'art s'intéressent désormais à la prise en compte de la microarchitecture dans le cadre de l'analyse des contre-mesures [GHM22 ; Gig+21 ; MM24 ; Bec+22]. Ces travaux montrent que les vérifications au niveau logiciel (source, assembleur ou binaire) n'exposent pas suffisamment d'informations pour établir des modèles de fuite (formalisant les informations observables sur un système pendant son exécution) assez précis et pouvant expliquer les failles observées en pratique.

De plus, les travaux récents [GHM22] insistent sur l'importance de pouvoir vérifier des implémentations matérielles sans devoir en réaliser une modélisation manuelle, qui est une tâche complexe, source d'erreurs et dont le résultat peut ne pas représenter parfaitement l'architecture matérielle ciblée.

La vérification formelle de ces contre-mesures, logicielles comme matérielles, offrant des garanties dans un modèle de fuite donné, reste une thématique ouverte. Le support de modèles de fuite réalistes et le passage à l'échelle de l'analyse sur des implémentations matérielles et logicielles de complexité moyenne (dizaines de milliers d'instructions sur des microcontrôleurs) participent à cette thématique.

Dans ce contexte, les travaux présentés dans cette thèse portent en un premier temps sur la vérification formelle des implémentations de contre-mesures face aux attaques par

canaux auxiliaires, sur des microcontrôleurs et sur des accélérateurs cryptographiques. Ces travaux portent dans un second temps sur la correction automatique des implémentations logicielles pour microcontrôleurs en exploitant le cadre formel de la vérification.

Les contributions clés de cette thèse sont :

— Une étude des différents modèles de fuites et des propriétés de sécurité associées, ainsi que l'identification d'une notion manquante dans l'état de l'art : la granularité de vérification.

— Une nouvelle méthode de modélisation des microarchitectures et des accélérateurs matériels, permettant de réaliser des vérifications au niveau matériel avec des modèles de fuite de l'état de l'art tout en considérant la granularité de vérification. Cette méthode est étoffée avec des optimisations visant à ne vérifier qu'un sous-ensemble conservatif des circuits modélisés en exploitant la nature des modèles de fuite, permettant des vérifications de circuits et logiciels plus complexes que ceux étudiés formellement jusqu'à présent.

— Une implémentation de référence de cette nouvelle méthode, publiée sous licence libre, nommée `aLEAKator`. Celle-ci est appliquée sur cinq processeurs différents, ARM et RISC-V, exécutant des applications cryptographiques. Des résultats nouveaux comparant les différents verdicts de vérification avec des mesures réelles sur des microcontrôleurs embarquant des processeurs ARM Cortex-M3 et ARM Cortex-M4 [1] sont également présentés.

— Une étude approfondie des fuites usuelles face aux attaques par canaux auxiliaires au sein des processeurs non sécurisés, complétée par des contre-mesures prouvées par `aLEAKator` et illustrées par des mesures réelles.

— Une méthode nouvelle d'élimination automatique des fuites pour ces processeurs, exploitant les informations rendues disponibles par `aLEAKator` et permettant de renforcer la sécurité des codes sources sur les processeurs supportés. L'efficacité de cette méthode est démontrée avec une implémentation d'AES masquée selon le schéma de Herbst [HOM06] pour les processeurs précédemment mentionnés. À notre connaissance, aucune implémentation sécurisée de cet algorithme n'existe pour un modèle de fuite aussi précis.

La partie I de ce manuscrit introduit les attaques par canaux auxiliaires, les contre-mesures, ainsi que les méthodes de vérification et de renforcement de sécurité associées. Des points manquants de l'état de l'art sont identifiés pour définir ensuite la problématique étudiée par cette thèse. La partie II détaille les contributions de ces travaux et les positionne dans l'état de l'art pour répondre à la problématique énoncée. Enfin, la partie III conclut ce manuscrit en résumant les contributions et proposant des perspectives pour des travaux futurs.

1. Ces processeurs sont présents sur plusieurs milliards d'appareils [Lim26b] vendus dans le monde.

Première partie

Contexte, état de l'art et problématique

2 Attaques par canaux auxiliaires et contre-mesures associées

2.1 Contexte des attaques par canaux auxiliaires

Ce manuscrit se concentre sur les attaques par canaux auxiliaires. Initialement mises en œuvre sur des implémentations cryptographiques par *Kocher et al.* vers la fin des années 1990 [Koc96 ; KJJ99], ces attaques sont depuis largement étudiées en raison de leur criticité. Les grandeurs physiques utilisées pour réaliser ces attaques sont variées. Des attaques ont notamment été exécutées en exploitant la puissance instantanée [KJJ99 ; HME25], les radiations électromagnétiques [Agr+03 ; Roc25], la temporalité [Lip+18 ; Koc+19], des informations thermiques [KS22], des informations optiques [Vie+24 ; FH08] ou encore l'acoustique [GST17] de circuits en fonctionnement.

Les attaques combinées à des fautes sont aussi étudiées, notamment dans des travaux exploitant les canaux auxiliaires pour synchroniser des actionneurs de faute [Fan+23].

Les attaques peuvent être passives, c'est-à-dire que l'attaquant n'a aucune interaction avec le circuit, mais aussi actives dans lesquelles l'attaquant, sans modifier le fonctionnement du système, va interférer avec les émissions des grandeurs physiques. Un exemple d'attaque active est le contournement des boucliers physiques, obtenu en irradiant d'ondes électromagnétiques les ouvertures nécessaires pour l'alimentation de circuits puis en mesurant la réflexion causée par les éléments internes au bouclier [Kit+26].

Ces attaques visent généralement les implémentations cryptographiques dans l'objectif de retrouver les données secrètes manipulées, le plus souvent une clé cryptographique.

Pour réaliser ces attaques, un attaquant doit pouvoir effectuer des mesures de la grandeur physique de son choix, à une fréquence suffisamment élevée par rapport à la cible qu'il souhaite attaquer. Cette section aborde la conception des cibles (matérielles et logicielles), détaille les effets physiques à l'origine des variations de grandeurs physiques en prenant l'exemple de la consommation puis décrit finalement plusieurs types d'attaques de la littérature.

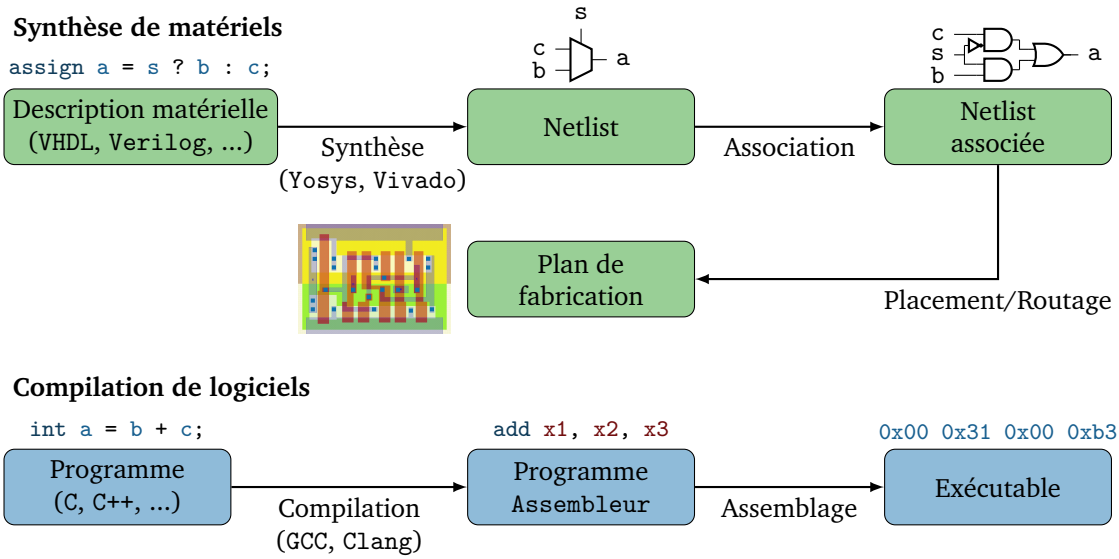


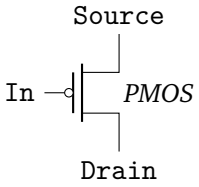
Fig. 2.1 : Vue simplifiée des traitements appliqués aux sources pour produire des circuits et des logiciels.

2.1.1 Cibles des attaques

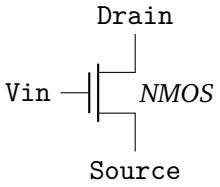
Les cibles des attaques par canaux auxiliaires sont des plateformes matérielles qui embarquent une implémentation cryptographique qui peut être sous la forme d'un accélérateur cryptographique dédié (et donc non programmable) ou d'un microcontrôleur exécutant différentes implémentations logicielles.

Ces plateformes matérielles sont décrites dans des *langages de description de matériel* (`HDL`, pour *H*ardware *D*escription *L*anguage) comme le `Verilog`, le `SystemVerilog` ou le `VHDL`. Le procédé de transformation d'une description `HDL` en un plan de fabrication de circuit est illustré dans la figure 2.1. Ces descriptions sont ensuite transformées par un outil de synthèse qui produit une liste de portes logiques et des connexions entre elles nommée *netlist*. Les netlists sont ensuite associées avec une *technologie*, c'est-à-dire que les portes logiques de haut niveau et les signaux de plusieurs bits sont convertis en portes élémentaires n'utilisant que des signaux de 1 bit de la technologie (qui définit l'agencement et la nature des transistors pour chaque porte logique supportée). Une fois la netlist associée, les étapes de placement et de routage doivent être réalisées pour obtenir un plan de fabrication du circuit, précisant l'emplacement physique de chaque transistor du circuit et les connexions réaliser entre ces transistors.

Les implémentations logicielles peuvent être définies à plusieurs niveaux. Cependant des étapes de transformation sont nécessaires pour produire un programme pouvant être exécuté par un microcontrôleur, à partir de sources dans un langage compilé tel que le `C`. Les étapes de transformation d'une description logicielle de haut niveau vers un programme

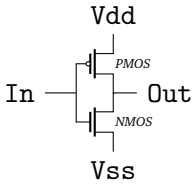


(a) *PMOS* : `Drain` et `Source` sont connectés si la tension `Vin` ≈ `Vdrain`.

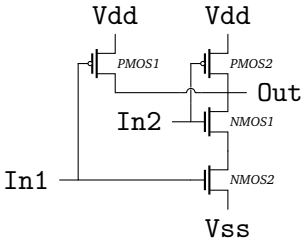


(b) *NMOS* : `Drain` et `Source` sont connectés si la tension `Vin` ≈ `Vsource`.

Fig. 2.2 : Transistors *NMOS* et *PMOS*, utilisés pour implémenter des portes logiques dans la technologie *CMOS*.



(a) Inverseur : `Out` = $\overline{\texttt{In}}$.



(b) Porte NON ET : `Out` = $\overline{\texttt{In1}.\texttt{In2}}$.

Fig. 2.3 : Exemples de portes réalisées avec la technologie *CMOS*.

binaire à exécuter sont aussi illustrées dans la figure 2.1. Les sources en `C` sont converties en un *langage d'assemblage* (couramment appelé `Assembleur`) par un compilateur. Ce programme assembleur est ensuite transformé en une représentation binaire exécutable, aussi appelée binaire, qui représente la suite de directives à suivre par la microarchitecture. Toutes les microarchitectures ne fonctionnent pas avec le même langage d'assemblage. Les étapes individuelles des programmes assembleurs sont des instructions et doivent être compatibles avec le jeu d'instructions (ou *ISA*, pour *I*nstruction *S*et *A*rchitecture) de la microarchitecture qui exécutera le programme. Les deux langages d'assemblage utilisés dans ce manuscrit sont le `RV32IMC` et l'`ARMv7-M`, respectivement pour les microcontrôleurs embarquant des processeurs RISC-V et ARM.

2.1.2 Mécanismes sous-jacents aux variations des grandeurs physiques

Les exemples d'attaques de ce manuscrit sont réalisés en observant la consommation, mais des variantes similaires exploitant d'autres canaux existent.

La consommation d'un circuit est liée aux opérations qui sont réalisées ainsi qu'aux données manipulées. Cela s'explique physiquement par l'évolution de la tension aux bornes des transistors qui composent les portes logiques au cours des calculs.

La technologie la plus commune de réalisation des fonctions logiques est la technologie *CMOS* (pour *C*omplementary *M*etal-*O*xide-*S*emiconductor). Elle associe des paires de

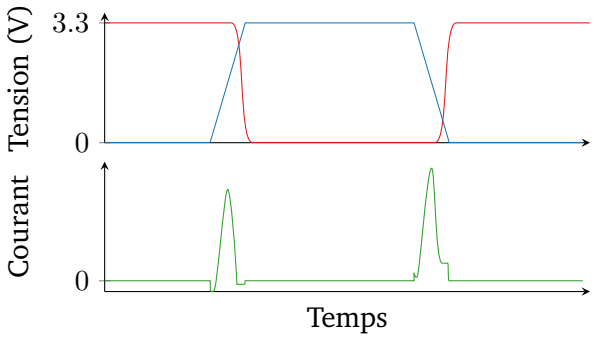


Fig. 2.4 : Évolution de la tension de sortie (en rouge) et du courant (en vert) pendant le changement d'état d'un inverseur CMOS dont l'entrée (en bleu) évolue de `Vss` à `Vdd` puis de nouveau à `Vss`.

transistors *PMOS* et *NMOS* tels que décrits dans la figure 2.2. Chaque transistor est assimilé à un interrupteur contrôlé par une tension. Deux niveaux de tension sont considérés : une tension haute (`Vdd`), assimilée à un état logique `1`, et une tension basse (`Vss`), assimilée à l'état logique `0`. Les portes logiques, réalisées en agençant des transistors, produisent sur leur sortie une tension correspondant à l'application de la fonction booléenne considérée sur les niveaux de tension des entrées. La figure 2.3 illustre un inverseur, composé d'un transistor *PMOS* et d'un transistor *NMOS* (figure 2.3a) ainsi qu'une porte logique réalisant l'opération NON ET composée de deux transistors de chaque type (figure 2.3b).

Ces transistors ont une consommation *statique*, subtilement différente selon que la sortie du transistor est à l'état `1` ou à l'état `0`. À cette consommation statique s'ajoute une consommation *dynamique*, prépondérante, lorsque la sortie change d'état (`1`→`0` ou `0`→`1`). Cette consommation est due à deux facteurs : les capacités internes aux portes logiques sont chargées ou déchargées pendant les changements d'état, et un court-circuit temporaire, inhérent à l'agencement interne des transistors CMOS, se produit à chaque changement d'état. La variation de consommation liée aux *transitions* d'un inverseur CMOS tel que celui de la figure 2.3a est illustrée dans la figure 2.4.

Les transistors qui changent d'état ne font pas instantanément basculer leurs sorties. De même, les connexions entre les transistors participent au décalage temporel (ou temps de propagation) accumulé pendant le transfert de l'information au sein d'un réseau de transistors. Les *glitches*, illustrés dans la figure 2.5, sont des effets se produisant lorsque ces temps de propagation se cumulent et provoquent au moins deux transitions d'un transistor dans un même cycle d'horloge, déclenchant des variations de consommation liées aux valeurs transitoires.

Les interconnexions entre les entrées et les sorties des portes logiques du circuit peuvent influencer l'inductance des autres interconnexions à proximité et causer des variations de

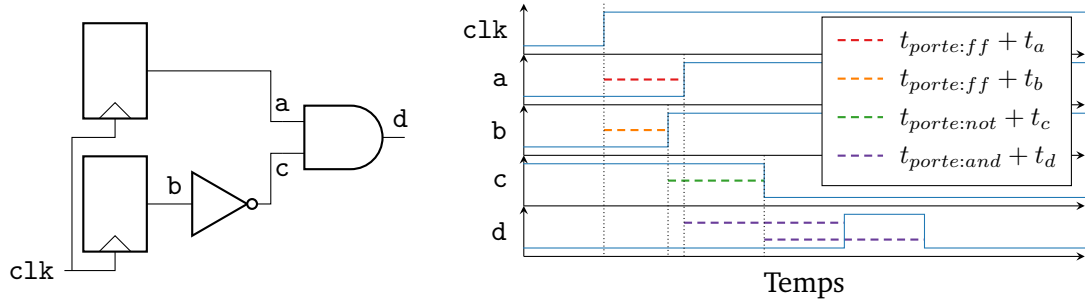


Fig. 2.5 : Visualisation des glitches sur un circuit d'exemple lorsque le front montant de l'horloge `clk` déclenche des fronts montants sur les interconnexions `a` et `b`. Pendant le court instant au cours duquel la porte NOT qui pilote `c` n'a pas encore propagé son résultat sur sa sortie, `d` propage temporairement un résultat qui n'est pas le résultat stabilisé sur sa sortie. Lorsque la porte NOT propage finalement la négation de `b` sur `c`, un délai de propagation est de nouveau nécessaire pour que cette transition se répercute sur `d`.

consommation. Cet effet, appelé effet de *couplage*, se produit naturellement pendant le fonctionnement des circuits.

2.1.3 Réalisation des attaques

Les attaques par canaux auxiliaires sont généralement réalisées en trois phases : acquisition, post-traitement, puis exploitation.

La phase d'acquisition consiste en la réalisation de mesures physiques sur une plateforme pendant son exécution. Le résultat de cette acquisition est une ou plusieurs *traces* qui associent une mesure de la grandeur physique à chaque instant d'échantillonnage.

Certaines propriétés des traces de grandeur physique peuvent compliquer la mise en œuvre des attaques, notamment la présence de bruit ou encore des décalages temporels entre les traces. L'étape de post-traitement permet, optionnellement, de débruiter [MDS99 ; WP20] ou de resynchroniser [TH12 ; MWB11] les traces.

Des plateformes « sur l'étagère » existent pour faciliter la mise en œuvre des attaques, fournissant des cibles à attaquer ainsi que des sondes, oscilloscopes et autres moyens d'instrumentation au sein d'un même circuit. La qualité du matériel utilisé pour réaliser les mesures influe sur la simplicité de réalisation des attaques puisque des matériels précis et rapides permettent d'obtenir des traces moins bruitées et avec des échantillons moins espacés.

Parmi les fournisseurs de telles solutions, *NewAE* propose la carte de mesure ChipWhisperer CW1200 [OC14]. Cette carte, utilisée pour réaliser les expériences décrites dans ce manuscrit, permet de changer à la demande le système embarqué instrumenté et, en option,

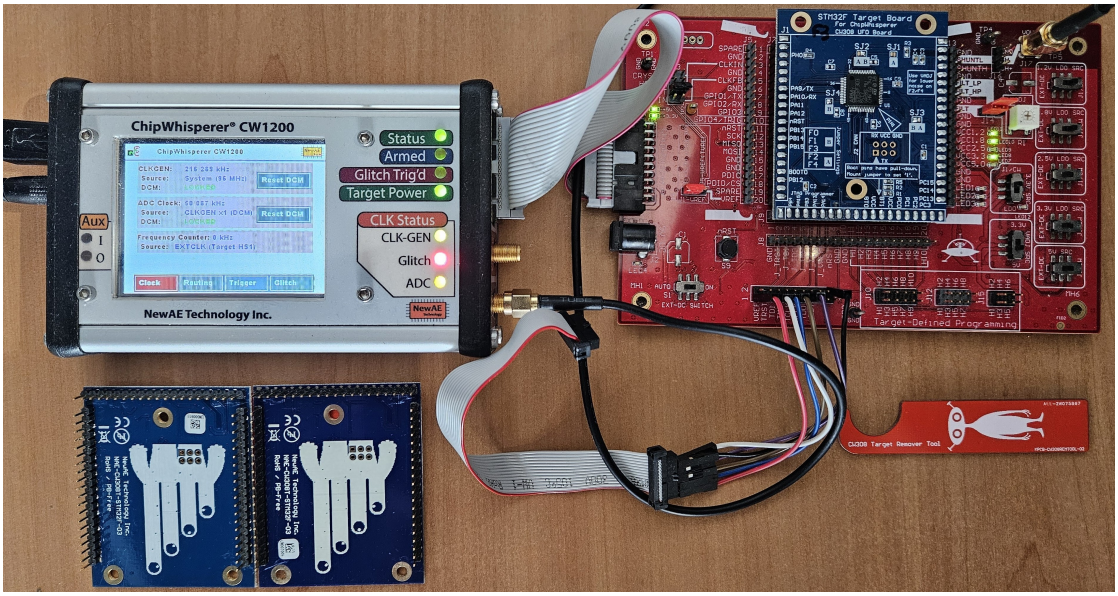


Fig. 2.6 : Photographie du banc de mesure incluant la carte ChipWhisperer Pro et trois cibles interchangeables.

d'ajouter des sondes électromagnétiques. La figure 2.6 illustre le banc de test expérimental ainsi que les trois microcontrôleurs utilisés pour réaliser les expériences de ce manuscrit. Certaines solutions de mesure sont abordables pour les particuliers (autour de 50€) et d'autres, disponibles pour les professionnels, sont plus onéreuses mais proposent plus de fonctionnalités.

La phase d'exploitation consiste à analyser des traces dans afin de récupérer les secrets. Cette étape de récupération est généralement réalisée avec une approche « diviser pour mieux régner » qui procède par fragments de secrets. La section suivante est consacrée aux différentes attaques de l'état de l'art qui exploitent des traces de consommation.

2.2 Attaques par canaux auxiliaires exploitant la consommation

Les méthodes d'exploitation sont séparées en deux catégories : les attaques non profilées (ou directes) qui sont directement réalisées sur la cible à attaquer (sous-section 2.2.1) et les attaques profilées qui reposent sur une phase préliminaire de profilage puis une phase d'exploitation (sous-section 2.2.2).

2.2.1 Attaques non profilées

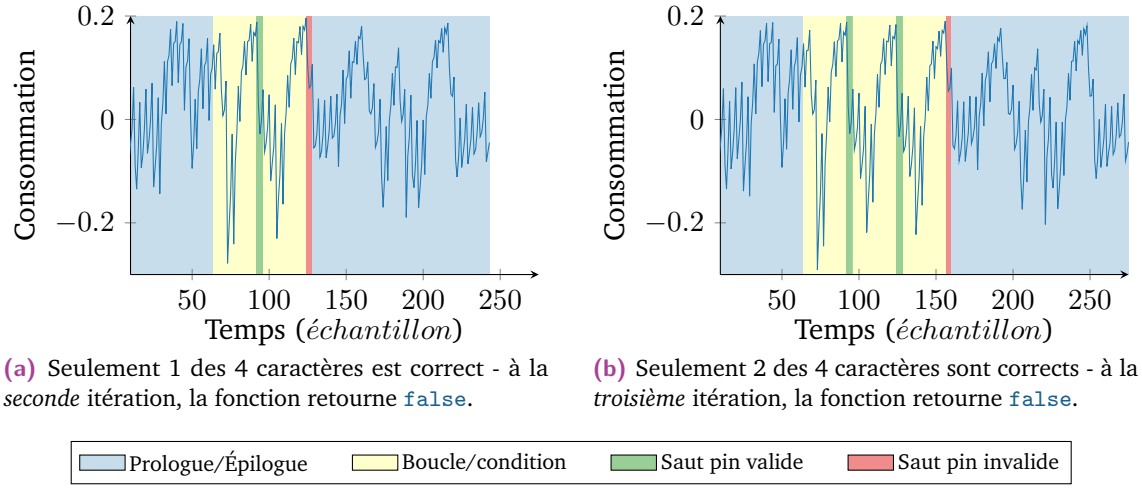
Les *SPA* [KJJ99 ; May00] (pour *S*imple *P*ower *A*nalysis) consistent en l'identification, sur une ou plusieurs traces, de variations de consommation liées aux instructions dont l'exécution change avec les données secrètes. Les SPA exploitent généralement les instructions dont la durée d'exécution varie selon les données manipulées comme, par exemple, les sauts conditionnels qui nécessitent un nombre variable de cycles pour s'exécuter selon que le saut est pris ou non. De même, certaines opérations comme les multiplications sont facilement distinguables des autres, avec très peu de traces. Bien que ciblant généralement les instructions dont les durées d'exécution sont dépendantes des variables secrètes, les SPA peuvent tout de même être utilisées pour distinguer différents résultats de calcul selon les données utilisées à l'exécution [HME25].

Exemple 1. L'implémentation en `C` d'une *vérification de code pin*, donnée dans le listing 2.a, peut être attaquée via une SPA. Les instructions ont des signatures de consommation différentes, permettant de distinguer, par exemple, le nombre d'itérations de boucle réalisées pour un pin donné. Un attaquant peut donc reconstruire le code pin correct en essayant les caractères un par un et en interprétant comme correct tout caractère qui cause une itération supplémentaire. Cette différence, illustrée dans la figure 2.7, peut être exploitée par un attaquant pour retrouver le mot de passe correct avec une complexité réduite (de 10^4 tentatives dans le pire des cas à $4 * 10$ tentatives).

```
1  int verify_pin(uint8_t* input) {
2      uint8_t correct_pin [4] = {0, 1, 2, 3};
3      for (int i = 0; i < 4; i++) {
4          if (input[i] != correct_pin[i]) {
5              return false;
6          }
7      }
8      return true;
9  }
```

Listing 2.a : Fonction `verify_pin`: vérification d'un mot de passe de 4 chiffres dans l'intervalle $[0, 9]$ par rapport à une référence.

Afin de cibler les données traitées par la cible, il est nécessaire de pouvoir distinguer celles-ci au sein de mesures de consommation. Le *poids de Hamming*, calculé comme le nombre de bits à `1` d'un mot binaire, est un estimateur de la consommation [HME25 ; May00 ; CKN01 ; Koc+11] communément utilisé lorsqu'un attaquant possède peu d'informations sur la cible qu'il attaque. Ce modèle suppose que de nombreuses portes logiques vont évoluer de `0` (ou d'une valeur constante) vers la valeur secrète si celle-ci est manipulée dans un calcul et donc que la consommation globale reflètera ces transitions. L'utilisation du poids de



(a) Seulement 1 des 4 caractères est correct - à la *seconde* itération, la fonction retourne `false`.

(b) Seulement 2 des 4 caractères sont corrects - à la *troisième* itération, la fonction retourne `false`.

Fig. 2.7 : Différence de consommation lors de la vérification d'un code PIN selon le nombre de caractères corrects. Traces réalisées sur le `STM32F303` sur une unique exécution par PIN.

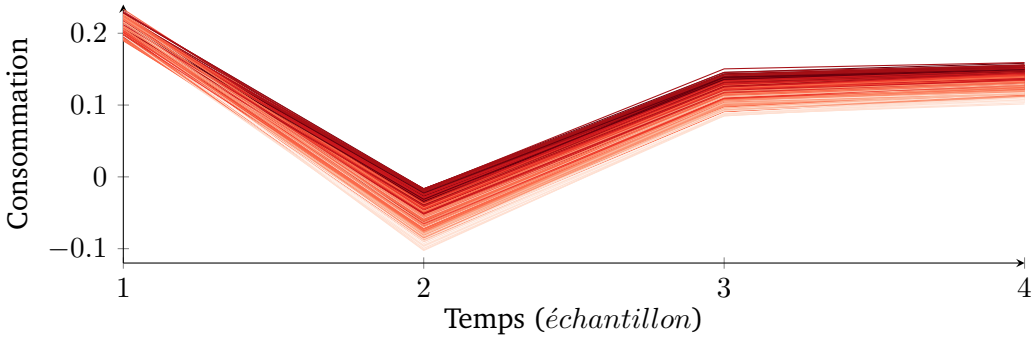


Fig. 2.8 : Superposition des consommations pour 3000 traces en sortie de la première lecture de la SBox d'AES (octet 0 de la clé) sur le `STM32F303`. La couleur est déterminée à partir du poids de Hamming du plus clair ($HW = 0$) au plus foncé ($HW = 8$).

Hamming n'est pas une modélisation idéale, mais ne nécessite pas d'informations sur l'état et la structure du circuit, expliquant son utilisation en pratique.

Exemple 2. La figure 2.8 illustre des traces de consommation superposées puis colorées selon le poids de Hamming de la donnée lue à l'instant de la mesure. L'évolution continue de la teinte selon le poids de Hamming met en évidence que celui-ci est distinguable dans une mesure de consommation globale.

Les *DPA* [KJJ99 ; Koc+11] - pour *D*ifferential *P*ower *A*nalysis - sont plus généralement utilisées pour cibler les données et non les instructions. Elles consistent en la mesure de nombreuses traces pour lesquelles les entrées du calcul sont connues, puis en leur regroupement en fonction d'une valeur (dépendante du secret) en commun (par exemple les traces pour lesquelles un bit ciblé vaut 1 et celles pour lesquelles il vaut 0). Une analyse de différence moyenne de consommation est ensuite réalisée entre ces groupes

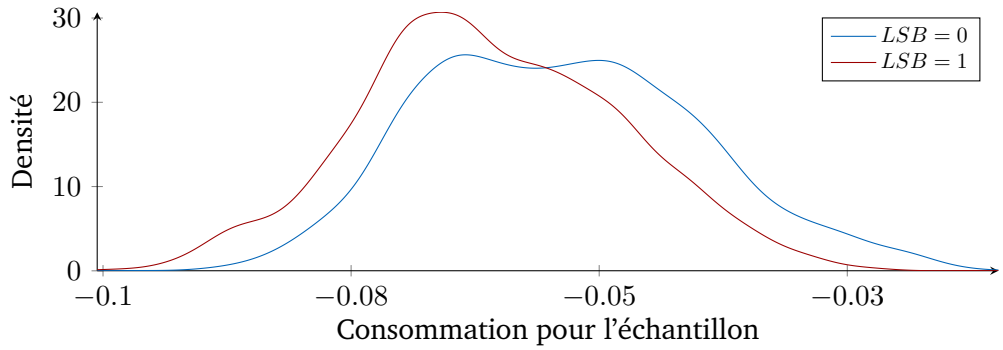


Fig. 2.9 : Différence de distribution de la consommation à l'échantillon de la première de la SBox d'AES (octet 0 de la clé) sur le `STM32F303`, lorsque le bit de poids faible de l'indice vaut `1` ou `0`.

afin de déterminer, pour une observation (avec ou sans conversion vers un modèle de consommation), quelle valeur est la plus probable. Cette méthode doit être utilisée pour reconstruire, itérativement, la valeur de la clé. La figure 2.9 illustre la différence de distribution d'un calcul de *SBox* (*S*ubstitution *Box*, table de substitution) lorsque le bit de poids faible de la sortie vaut `1` ou `0`.

Parmi les attaques différentielles, les *CPA* [BCO04 ; Meu19] - pour *C*orrelation *P*ower *A*nalysis - sont aussi utilisées principalement pour cibler les données et non les instructions. Elles ne consistent pas en la séparation des traces en groupes, mais en le calcul de la corrélation entre de nombreuses traces avec une hypothèse de clé sous un modèle de consommation tel que le poids de Hamming. L'hypothèse de clé qui maximise la corrélation est considérée comme la bonne.

2.2.2 Attaques profilées

Les attaques profilées (ou template) [CRR03] sont une catégorie d'attaques basées sur une phase préalable à l'attaque dite « hors ligne » ou phase de profilage. Cette phase consiste en l'établissement d'un profil statistique de consommation réalisé sur un matériel le plus proche possible de celui qui sera attaqué mais sur lequel l'attaquant a le contrôle total.

Dans la seconde phase de ces attaques, l'attaquant peut ensuite n'utiliser qu'un nombre réduit de traces sur le matériel ciblé pour réaliser l'analyse statistique afin de déterminer les hypothèses de secret les plus vraisemblables, à partir du profilage préalable.

De nombreuses attaques profilées existent, utilisant des templates gaussiens [CRR03] ou des solutions de machine learning [MPP16].

Ces différentes attaques exploitent la relation entre les grandeurs physiques mesurées et les données ou instructions manipulées. En réponse à ces menaces, des contre-mesures perturbant cette relation ont été proposées pour complexifier la réalisation des attaques.

2.3 Contre-mesures

Les contre-mesures sont séparées en deux grandes catégories avec d'une part les contre-mesures de *dissimulation* qui visent à complexifier les attaques en cachant le lien entre les éléments secrets et les grandeurs physiques observables. D'autre part, les contre-mesures de *masquage* visent à transformer le calcul réalisé de manière à décorréler les valeurs intermédiaires des valeurs secrètes.

Ces contre-mesures sont des concepts abstraits qui ne sont concrétisés qu'à leur déploiement dans une implémentation cryptographique, à un niveau pouvant être physique, matériel ou, pour les processeurs, logiciel. Ces niveaux d'implémentation ciblent respectivement la technologie utilisée pour fabriquer un circuit, la logique d'un circuit et le programme à exécuter.

Quel que soit le niveau d'implémentation choisi pour une contre-mesure, l'objectif est de limiter les fuites sur le circuit final et, pour les processeurs, sur le logiciel final exécuté sur ceux-ci.

2.3.1 Dissimulation

La dissimulation vise à rendre difficile l'identification d'une valeur secrète lors d'observations de grandeurs physiques, sans altérer le traitement fonctionnel de ces valeurs secrètes au sein d'un calcul. La dissimulation procède par ajout de bruit, d'irrégularités temporelles ou d'irrégularités spatiales pour noyer l'information pertinente dans un ensemble plus vaste (soit très variable, soit constant d'une exécution à l'autre), rendant plus difficile son identification au sein d'une observation. Néanmoins, la fonctionnalité à la granularité du calcul reste identique. Les protections mises en œuvre ne font donc qu'obfusquer la dépendance statistique de la grandeur observée vis-à-vis des valeurs secrètes, sans la supprimer.

La sécurité apportée par ces contre-mesures n'est généralement pas prouvable, mais celles-ci sont néanmoins déployées en pratique car elles complexifient la réalisation des attaques, créant un contexte hostile à la répétabilité ou à la mesure.

Contre-mesures de dissimulation physiques

La littérature propose divers mécanismes, généralement spécifiques à une grandeur physique donnée, afin de complexifier la mise en œuvre des attaques au niveau de la technologie utilisée pour produire des circuits.

Il existe une distinction entre les approches passives, dont l'objectif est de perturber les mesures et les approches actives, qui visent à détecter des activités suspectes (telles que la proximité d'une sonde) par un circuit pendant son fonctionnement. Ces premières incluent notamment l'utilisation de boucliers pour limiter les émanations électromagnétiques [Dob+25], l'ajout de cellules pour égaliser les émissions électromagnétiques [Wan+18], le développement de portes logiques alternatives présentant une consommation électrique indépendante des données manipulées [TAV02], l'intégration de régulateurs de tension internes au circuit [RWB04], ou encore des portes logiques additionnelles opérant les transitions inverses à celles réalisées par le circuit avec la logique à double rail [TV04].

Un exemple de méthode active est l'intégration d'oscillateurs LC pour détecter la présence de sondes électromagnétiques [Hom+14] à proximité d'un circuit.

Contre-mesures de dissimulation matérielles

Les contre-mesures de dissimulation matérielles peuvent être temporelles, comme par exemple l'intégration, à l'exécution par la microarchitecture, d'instructions factices à des instants irréguliers et aléatoires, telle qu'implémentée dans des co-processeurs sécurisés [Pha+21], l'Ibex [Con18] et le CV32E40P [LSB22]. Des travaux proposent aussi d'ajouter des modules matériels entre les processeurs et les mémoires pour rendre aléatoire (mais en maintenant la cohérence des dépendances) l'ordre d'exécution des instructions [Bay+12] ou d'utiliser des processeurs non déterministes qui choisissent aléatoirement la prochaine instruction à exécuter [MMS01a]. De même, l'exécution d'instructions en temps constant est aussi utilisée pour dissimuler les opérations sensibles. Cette contre-mesure est implémentée en rendant la durée d'exécution (en nombre de cycles) des instructions indépendante des données qu'elles manipulent. Cela peut s'appliquer aux sauts conditionnels, rendant leur temps d'exécution indépendant du fait que le saut soit pris ou non. Les instructions comme les multiplications qui proposent, selon les données, des raccourcis d'exécution, sont aussi concernées. Par exemple, les multiplications triviales (par 0 par exemple) sont tout de même exécutées comme si elles ne l'étaient pas [Con18].

Des contre-mesures de dissimulation matérielles spatiales ont également été proposées. Par exemple, la permutation aléatoire des affectations de registres ou des adresses mémoire [Con20 ; Sas+15 ; MMS01b] impacte la localité des variations de grandeurs physiques, complexifiant les attaques qui nécessite des sondes placées précisément sur la cible (sondes électromagnétiques, par exemple).

Des fonctionnalités de nettoyage de caches, mémoire et registres sont également des mesures de dissimulation existantes [Con20]. Elles utilisent des valeurs aléatoires pour limiter les variations de grandeurs physiques liées aux données effacées. Le processeur sécurisé Aubrac [EM26] nettoie automatiquement les registres avant leur réécriture pour empêcher les transitions. D'autres contre-mesures sont possibles : l'utilisation de l'espace inutilisé sur le circuit pour générer du bruit [GM11] ou encore la conception de bancs de registres et de mémoires exempts de glitches, tel qu'implémenté dans le processeur Coco-Ibex [Gig+21] mais pouvant aussi être implémentées sur des accélérateurs cryptographiques.

Contre-mesures de dissimulation logicielles

Des contre-mesures logicielles de dissimulation similaires aux contre-mesures matérielles de dissimulation présentées ont été développées. L'ajout d'instructions factices [RPD09 ; CK10] ainsi que la programmation en temps constant en sont deux exemples communs. Le polymorphisme de code a aussi été étudié [Ama+11]. Cette technique consiste en la génération de plusieurs versions différentes mais fonctionnellement équivalentes d'une même fonction puis en la sélection au hasard d'une de ces versions à l'exécution pour complexifier l'alignement des traces. Certains travaux proposent de réaliser la génération des versions à exécuter à la volée pendant le fonctionnement [Cou+16]. Certaines de ces approches sont implantées au travers de passes de compilation, notamment dans le projet LLVM [BF].

Plusieurs librairies cryptographiques [Ope ; Bea] annoncent implémenter des contre-mesures de sécurisation face aux attaques par canaux auxiliaires liées au temps, notamment la programmation en temps constant.

D'autres techniques se basent sur la dissimulation des adresses mémoires effectivement utilisées par un calcul ou leur changement au fil des exécutions [HOM06 ; RLT15] ou encore la parallélisation d'opération sensibles avec d'autres manipulant des valeurs aléatoires [US24 ; PBP23].

Leur implémentation complexifie les attaques en pratique sans offrir de garantie de protection ni supprimer le lien entre les données manipulées et les grandeurs physiques observables. Elles doivent donc être complétées par des contre-mesures qui rompent ce lien

et pour lesquelles cette rupture est prouvable. Le masquage est proposé à cette fin et est la contre-mesure principalement étudiée dans ce manuscrit.

2.3.2 Masquage

Contrairement à la dissimulation, le masquage transforme fonctionnellement le calcul réalisé sur les valeurs secrètes, de telle sorte que celles-ci ne soient plus corrélées avec les valeurs intermédiaires du calcul. Le masquage est appliqué dans le but que les observations des grandeurs physiques, corrélées aux valeurs manipulées, ne révèlent plus d'information directe sur les valeurs secrètes.

Le masquage, introduit en 1999 [GP99 ; Cha+99], consiste à transformer un programme de manière à ne pas manipuler directement les valeurs secrètes, mais à faire en sorte que les expressions manipulées aient une distribution de valeurs indépendantes des secrets. Le masquage consiste en l'ajout d'opérations entre la valeur secrète et une variable dite *masque*, qui change de valeur à chaque exécution, en suivant une distribution aléatoire uniforme. Lorsque cette opération est le « ou exclusif » ($\oplus$), le masquage est dit *booléen* tandis que lorsqu'une autre opération telle qu'une addition ou une multiplication est utilisée, le masquage est *arithmétique* ou *multiplicatif*. Cela est illustré sur le tableau 2.1, qui montre la table de vérité de l'expression `e = k` $\oplus$ `m` : pour les 2 valeurs possibles du secret `k`, la distribution de l'expression est identique : `0` avec une probabilité $\frac{1}{2}$, `1` avec une probabilité $\frac{1}{2}$, car pour un `k` fixé, chaque ligne, correspondant à une valeur de masque a la même probabilité (car les masques ont une distribution uniforme). Ainsi, en observant la valeur de l'expression, il n'est pas possible de déduire d'information sur la valeur secrète. Il s'ensuit que sous l'hypothèse que la fuite observable est la valeur de l'expression, il n'existe pas de fuite secrète, et cela peut être *prouvé*, ce qui est l'attrait principal pour cette contre-mesure. La propriété de sécurité définie informellement ici et introduite dans [ISW03] est appelée Threshold Probing Security à l'ordre 1. Elle a été mise en œuvre dans diverses implémentations, comme le schéma de masquage logiciel de l'AES [RD01] appelé schéma de Herbst [HOM06] et dont une brève explication est donnée dans l'annexe A.

k	m	e
0	0	0
	1	1
1	0	1
	1	0

Tab. 2.1 : Table de vérité de `e = k` $\oplus$ `m`, avec un secret `k` et un masque `m`.

Pour un programme purement linéaire vis-à-vis de l'opération de masquage ($\oplus$ pour le masquage booléen, $+$ pour le masquage arithmétique), le programme initial peut être conservé et simplement opérer sur des entrées masquées et produire un résultat devant être démasqué. Cependant, la majorité des programmes utilisent des opérations non linéaires vis-à-vis de l'opération de masquage et imposent donc la transformation du programme. Ces changements ne sont pas toujours triviaux, et nécessitent également des opérations sur les masques eux-mêmes. Il devient donc en général plus aisé d'adopter une autre représentation des expressions du programme masqué, dans lesquelles les parties du secret sont représentées de manière symétrique. Ainsi, le masquage peut être défini de manière plus restrictive de la façon suivante :

Définition 1. À l'ordre d, le masquage consiste en la séparation d'une variable secrète s en $d+1$ parties $s_0 \dots s_d$ appelées *shares*, en utilisant d variables aléatoires uniformément réparties de manière à ce que $s = s_0 \star \cdots \star s_d$, avec $\star$ l'opérateur utilisé pour la séparation en shares.

Les deux représentations sont proches, mais pas tout à fait équivalentes, ce qui se traduit notamment par le fait que les propriétés de sécurité sur l'une et l'autre représentation vont différer, le Threshold Probing Security ne pouvant pas être validé sur cette représentation en shares qui ne fait plus apparaître de secret. De plus, le masquage utilisant une représentation des variables en secrets et masques permet plus de libertés quant à la réutilisation des masques pour masquer différents secrets (par exemple dans l'AES Herbst, le même masque est utilisé pour masquer 16 secrets différents), ce qui n'est pas possible dans une représentation avec des shares.

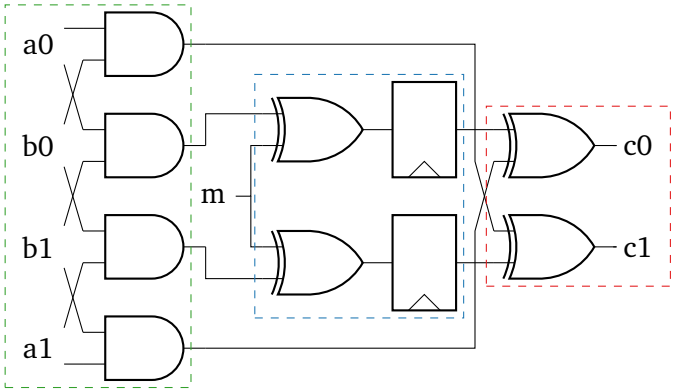
Plus concrètement, avec une représentation du masquage booléen à l'ordre 1 en shares, deux secrets a et b sont exprimés (sa_1, sa_2) et (sb_1, sb_2). Tandis qu'avec une représentation avec un masquage booléen à l'ordre 1 en masques et secrets, a et b sont exprimés $(a \oplus m_1, m_1)$ et $(b \oplus m_2, m_2)$. Dans cette représentation il devient possible de choisir $m_1 = m_2$, ce qui impliquerait que le $\oplus$ d'une partie de chaque secret $(a \oplus m_1) \oplus (b \oplus m_2)$ soit simplifiable en $a \oplus b$. Cette sélection du masque n'est pas possible avec la représentation en shares, donc $sa_1 \oplus sb_1$ ne peut pas révéler les secrets a ou b dans cette représentation.

Par la suite, on parlera de représentation en *secrets et masques* ou de représentation en *shares* selon le besoin.

Depuis les travaux initiaux sur le masquage booléen [ISW03] de nombreux schémas de masquage, notamment Threshold Implementation (TI) [NRR06] ou Domain-Oriented Masking (DOM) [GMK16] ont été proposés pour limiter le nombre de variables aléatoires nécessaires, limiter les glitches ou pour favoriser la composabilité des implémentations

```
1  module et(input a0, input a1,
2   input b0, input b1, input m,
3   input clk, output c0,
4   output c1);
5   assign a0b0 = a0 & b0;
6   assign a0b1 = a0 & b1;
7   assign a1b0 = a1 & b0;
8   assign a1b1 = a1 & b1;
9
10  assign a0b1z = a0b1 ^ m;
11  assign a1b0z = a1b0 ^ m;
12  always @(posedge clk) begin
13   reg0 = a0b1z;
14   reg1 = a1b0z;
15  end
16
17  assign c0 = a1b1 ^ reg0;
18  assign c1 = a0b0 ^ reg1;
19 endmodule
```

Listing 2.b : Implémentation `Verilog`.



(b) Représentation en portes logiques.

Fig. 2.10 : ET logique matériel, masqué avec le schéma `DOM` [GMK16]. La propriété suivante est respectée sur les sorties : `c0` $\oplus$ `c1 = (a0` $\oplus$ `a1) & (b0` $\oplus$ `b1)`.

masquées. Les petits circuits réalisant des opérations simples qui sont créés dans le but d'être agencés ensemble pour créer des circuits complexes sont appelés *gadgets* et leur agencement, *composition*.

Le masquage, au même titre que les contre-mesures de dissimulation, est un concept abstrait devant être implémenté de manière concrète, ce qui implique de choisir un niveau d'implémentation.

Le masquage repose sur la transformation même du calcul réalisé et n'est donc pas implémenté physiquement au sein d'une technologie. Les implémentations sont cependant possibles au niveau matériel et au niveau logiciel.

Contre-mesures de masquage matérielles

De nombreux travaux proposent des implémentations d'accélérateurs cryptographiques masqués [GMK16 ; BKN22 ; SM21b ; Tri03]. La figure 2.10 illustre un circuit réalisant la fonction ET logique, avec un masquage booléen.

Certains concepteurs implémentent aussi des architectures de bus ou de caches protégés, conçus pour acheminer uniquement des données masquées et faisant transiter les masques associés par une voie distincte. Les données masquées sont ensuite démasquées en sortie du bus ou du cache [Tal+22].

```c
#include <stdint.h>
uint32_t a0, a1, b0, b1, m;
uint32_t c0, c1;
int main () {
 // Produits croisés
 uint32_t a0b0 = a0 & b0;
 uint32_t a0b1 = a0 & b1;
 uint32_t a1b0 = a1 & b0;
 uint32_t a1b1 = a1 & b1;

 // Combinaison avec le masque
 volatile uint32_t a0b1m = a0b1^m;
 volatile uint32_t a1b0m = a1b0^m;

 // Résultat
 c0 = a1b1 ^ a0b1m;
 c1 = a0b0 ^ a1b0m;
 return 0;
}
```

Listing 2.c : Implémentation `C`.

```
// r2=a0, r5=a1, r3=b1, r6=b0
// r1=m, r11=@c0, r10=@c1
// Produits croisés
and r4, r2, r3 // r4=a0b1
and r0, r5, r3 // r0=a1b1
and r7, r5, r6 // r7=a1b0
and r8, r2, r6 // r8=a0b0

// Combinaison avec le masque
eor r4, r4, r1 // r4=a0b1^m
eor r7, r7, r1 // r7=a1b0^m

// Résultat
eor r0, r0, r4 //r0=a1b1^(a0b1^m)
eor r9, r7, r8 //r9=a0b0^(a1b0^m)

str r0, [r11] // store c0
str r9, [r10] // store c1
mov r0, #0
```

Listing 2.d : Implémentation `Assembleur`.

Fig. 2.11 : Masquage logiciel du ET logique avec le schéma `DOM` [GMK16], en `C` et en `Assembleur` ARM.

Contre-mesures de masquage logicielles

Le masquage peut aussi être implémenté au niveau logiciel, à tous les niveaux d'abstraction évoqués précédemment.

La figure 2.11 illustre, aux niveaux `C` et `Assembleur`, le ET logique masqué avec le schéma DOM [GMK16]. Ces implémentations logicielles sont similaires à l'implémentation matérielle de cette même opération décrite dans la figure 2.10, à la différence près qu'elles sont réalisées sur des registres de 32 bits.

2.3.3 Limitations du masquage

Au cours de la synthèse et de la compilation (illustrées dans la figure 2.1), des optimisations préservant l'équivalence fonctionnelle des descriptions sont appliquées. Cependant, différentes opérations du calcul peuvent être supprimées ou réordonnées par ces optimisations. Cela peut rendre vulnérable une implémentation en exposant des résultats intermédiaires secrets, temporairement démasqués.

Une seconde source de fragilisation de cette contre-mesure survient lors de l'exécution de logiciels sur des microarchitectures. Les logiciels sont décrits au niveau du jeu d'instructions, qui est l'interface avec les différents microcontrôleurs qui implémentent ce même jeu d'instructions. Ces microcontrôleurs sont composés de registres et de portes logiques

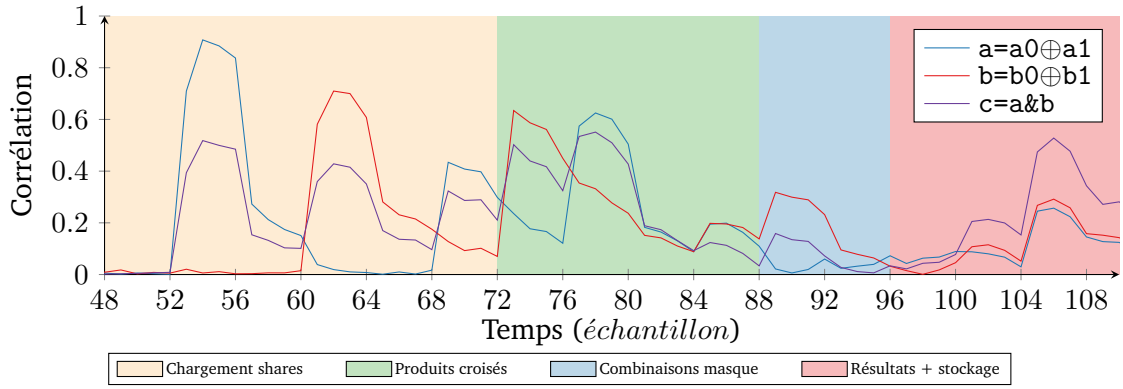


Fig. 2.12 : Corrélation sur 10 000 traces entre les valeurs secrètes et la consommation pour un ET masqué avec le schéma DOM [GMK16], tel que présenté dans le listing 2.d. L'implémentation ne présente aucune transition pouvant fuire avec les informations disponibles au niveau ISA.

internes qui ne sont pas visibles par le logiciel et qui ne sont pas les mêmes d'un microcontrôleur à un autre. Ces éléments peuvent causer des variations de consommation (liées aux changements de valeurs des transistors) qui ne sont pas prévisibles en ne prenant en compte que le logiciel. De plus, les effets matériels tels que les glitches renforcent ces différences en causant des variations de la consommation (liées aux valeurs transitoires propagées à travers les portes du circuit) qui dépendent du microcontrôleur et qui sont imprévisibles au niveau d'abstraction du logiciel.

Exemple 3. L'implémentation du `ET` masqué avec le schéma `DOM` proposée dans le listing 2.d est correctement masquée au niveau ISA et, à ce niveau de description, garantit l'absence de transition faisant intervenir des variables secrètes entre les registres généraux du processeur. Cependant, ces garanties ne sont plus valables lorsque le logiciel est exécuté sur une microarchitecture. La figure 2.12 montre la corrélation entre la consommation du `STM32F303` et les valeurs secrètes, en fonction du temps, lors de l'exécution de ce `ET`.

Vérifier qu'après toutes les étapes d'élaboration, les circuits et logiciels masqués sont sécurisés et qu'un logiciel reste sécurisé lorsqu'il est exécuté sur un processeur est donc crucial pour l'implémentation du masquage [Bec+22 ; Ami+26 ; GHM22 ; Gig+21 ; Bar+19 ; Bal+15 ; BP18]. Le chapitre 3 traite de la vérification du masquage d'implémentations logicielles et matérielles.

3

Vérification d'implémentations cryptographiques masquées

La contre-mesure de masquage est une méthode efficace de sécurisation des implémentations. Sa mise en place est cependant complexe car elle nécessite la prise en compte de l'implémentation complète, en présence d'effets causant des variations de consommation pouvant éventuellement être exploitées tels que les transitions et les glitches. Pour cette raison, le masquage doit être vérifié, afin d'assurer sa bonne mise en œuvre. La vérification peut avoir lieu à différentes étapes de la conception. Elle peut être pré-silicium (avant la production du circuit) à l'aide de simulations ou de modèles formels des descriptions matérielles et logicielles. La vérification peut aussi, complémentairement, être post-silicium en réalisant des mesures réelles sur le circuit produit. Une vérification au plus tôt permet de limiter le nombre d'itérations pendant la phase de conception, et donc, le coût de conception d'un produit, sans éliminer pour autant la nécessité de vérifier le circuit final.

Les méthodes pré-silicium sont l'objet de ces travaux. Qu'elles soient basées sur des simulations ou sur des modèles formels, ces méthodes utilisent généralement des *modèles de fuite* qui représentent, à un niveau d'abstraction choisi, les informations observables au cours de l'exécution. Le choix d'un niveau d'abstraction permet de réaliser la vérification à différentes étapes de la conception du produit. Sans cette abstraction, les méthodes par simulation devraient réaliser des simulations physiques complexes et coûteuses sur le plan de fabrication final du circuit (au niveau transistors), tandis qu'avec des hypothèses sur les sources de fuite, une simulation logique (au niveau des portes logiques) peut devenir suffisante. Ces modèles de fuite peuvent abstraire plus ou moins la microarchitecture, c'est le cas des vérifications réalisées au niveau ISA. Il est cependant capital de s'assurer de la pertinence physique du modèle utilisé, comme illustré dans la sous-section 2.3.3 sur les limitations du masquage, pour que la vérification donne une indication sur l'absence de fuites en pratique. Réaliser la vérification, pour les accélérateurs matériels comme pour les microarchitectures, à un niveau n'abstrayant pas le matériel est donc une nécessité. La vérification conjointe d'un logiciel fonctionnant sur un matériel est appelée *co-vérification logiciels-matériels*. Cela correspond à la vérification d'un logiciel exécuté par une microarchitecture.

3.1 Modèles de fuite

La sécurité d'une implémentation est généralement étudiée dans un modèle de fuite qui définit les informations observables lorsque le système est en fonctionnement. Différents niveaux d'abstraction peuvent exister pour définir ces modèles, exposant plus ou moins l'implémentation matérielle des circuits.

Dans le modèle `d-probing` [ISW03], un attaquant peut observer n'importe quel `d`-uplet de valeurs intermédiaires calculées. Intuitivement, pour une représentation en secrets et masques, une implémentation est *sécurisée à l'ordre* d si la distribution de chaque `d`-uplet est statistiquement indépendante des variables secrètes. L'analogie pour les représentations en shares est qu'une implémentation est *sécurisée à l'ordre* d si la distribution de chaque `d`-uplet dépend au plus du nombre de shares par variable moins un. Il est important de noter que l'ordre de masquage et l'ordre du modèle de fuite ne sont pas nécessairement les mêmes.

Le modèle `d-probing` est longtemps resté le plus utilisé en raison de sa simplicité mais il est difficile de le faire correspondre à des fuites observées en pratique, puisque d'autres effets physiques participent à la variation des grandeurs physiques observables pour réaliser une attaque par canaux auxiliaires. Parmi ces effets, les *glitches* et les *transitions* sont les deux plus pertinents à modéliser. Les raisons physiques derrière ces effets sont expliquées dans la sous-section 2.1.2 mais l'impact de ces effets sur les fuites dans un modèle de fuite est donné dans la définition 2.

Définition 2 (Glitches)**.** Les glitches sont des phénomènes d'instabilité temporaire de propagation des signaux se produisant naturellement dans des circuits avant leur stabilisation. Les glitches peuvent causer des alternances de valeurs en entrée des portes logiques, éventuellement propagées sur le signal de sortie de ces portes, et peuvent en conséquence révéler des informations sur les entrées de ces portes, quand bien même leurs valeurs stabilisées ne seraient pas observables sur leurs sorties.

Exemple 4. Un exemple de circuit et de propagation dont un glitch cause la fuite d'un secret est donné dans la figure 3.1.

Définition 3 (Transitions)**.** Les transitions sont de fortes variations instantanées des grandeurs physiques observables [MOP07] lorsque la sortie d'une porte logique utilisant la technologie CMOS bascule d'un `0` logique vers un `1` logique et inversement. En conséquence, un attaquant est capable d'observer la transition entre deux valeurs successives en réalisant une observation.

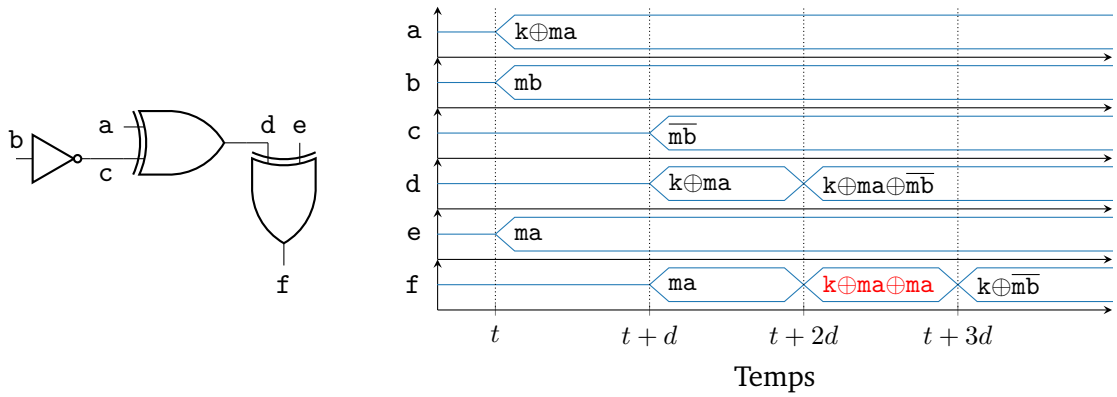


Fig. 3.1 : Glitches faisant apparaître k⊕ma⊕ma=k. Sans considérer les glitches, ce circuit ne présente pas de fuite. Pour ce chronogramme, le temps de propagation des interconnexions entre les portes est ignoré et le délai de stabilisation de toutes les portes est considéré identique et égal à d. Les entrées a, b et e sont respectivement pilotées aux valeurs k⊕ma, mb et ma, au même instant t.

Les fuites en transition peuvent être modélisées au niveau logiciel, pour les éléments microarchitecturaux visibles à ce niveau de représentation. Par exemple, en considérant les registres manipulés au niveau du jeu d'instruction il devient possible d'en approximer les transitions, sans pour autant représenter toutes les transitions de la microarchitecture. Les fuites en transition peuvent aussi être modélisées plus finement, au niveau des portes logiques ou directement au niveau des transistors. Il faut cependant évaluer en pratique la pertinence du niveau de représentation du modèle choisi.

Des modèles de fuites plus complets que le modèle `d-probing` ont été proposés pour modéliser plus finement ces particularités liées à la physique des transistors et la combinaison de leurs effets. Le modèle `robust d-probing` [Fau+18] étend le modèle `d-probing` avec les transitions, les glitches et les couplages (et toutes leurs combinaisons) et est défini comme le modèle `(g,t,c) d-probing`, dans lequel `g`, `t` et `c` sont définis à 0 ou 1 en conséquence. Le paramètre `c` correspondant aux effets de couplage n'est pas considéré dans ces travaux, le modèle est donc simplifié en `(g,t) d-probing` par la suite. À notre connaissance, aucun outil de vérification d'absence de fuite ne permet la prise en compte des effets de couplage, ceux-ci nécessitent des informations disponibles uniquement sur le plan de fabrication final des circuits et restreignent donc les preuves à un plan de fabrication donné.

L'approche généralement choisie pour prendre en compte les glitches pouvant se produire dans un circuit est de considérer que chaque porte logique combinatoire peut faire apparaître sur sa sortie toutes les valeurs pouvant apparaître sur ses entrées et que les portes séquentielles peuvent faire apparaître sur leurs sorties leur valeur courante et leur valeur précédente. Cette approche est une sur-approximation des glitches qui peuvent se produire en pratique. Elle permet de s'affranchir d'informations temporelles fines (délais de propagation des portes et des interconnexions) et des informations de placement (longueur

des interconnexions entre les portes) qui sont nécessaires pour connaître exactement la survenue de ces glitches. Pour limiter cette sur-approximation tout en bénéficiant d'une estimation conservative des glitches, la *stabilité* a récemment été introduite [HB21 ; MM24] et vise à ne pas considérer une partie des glitches qui ne peuvent pas se produire en pratique.

Définition 4 (Stabilité)**.** Un signal est stable s'il ne change pas de valeur entre deux cycles d'horloge consécutifs et qu'aucun glitch ne peut se produire. La sortie d'une porte séquentielle est stable uniquement si la valeur écrite dans celle-ci est la même sur deux cycles consécutifs. Il en découle qu'une porte combinatoire est stable si tout l'arbre de portes combinatoires participant aux entrées de celle-ci est stable.

Comme une porte logique dont les entrées sont stables ne peut pas faire apparaître de glitch en sortie, un attaquant qui observe une telle sortie ne peut observer que sa valeur courante (et stable). Le modèle `Robust but Relaxed d-probing` [MM24] ou `RR d-probing` est défini comme étant le modèle `(1,1) d-probing` avec la considération de la stabilité des signaux.

Il est nécessaire, lors de l'utilisation d'un modèle de fuite à l'ordre supérieur ($d > 1$), de préciser quelles sont les combinaisons de taille d qui sont considérées. En effet, elles peuvent être spatiales (d signaux ou variables à un instant donné), temporelles (d évolutions d'un même signal ou d'une même variable) ou spatio-temporelles (d évolutions de n'importe quel signal ou variable).

Ces modèles de fuite permettent de définir les informations qu'un attaquant peut obtenir en observant un circuit. Pour vérifier si un circuit est sécurisé dans un modèle de fuite donné, il est nécessaire de vérifier une *propriété de sécurité*.

Propriétés de sécurité

La propriété la plus intuitive associée à une implémentation masquée à l'ordre t avec des secrets et des masques est le fait que la distribution jointe de tous les t-uplets de valeurs intermédiaires possibles soit indépendante de toutes les valeurs secrètes. Cette propriété est appelée *T*hreshold *P*robing *S*ecurity à l'ordre t, ou *t-TPS*. On peut déjà noter que le nombre de t-uplets à vérifier croît avec l'ordre de vérification et avec le nombre de points spatio-temporels p à vérifier. Le nombre de t-uplets est $\binom{p}{t} = \frac{p!}{t!(p-t)!}$, ce qui rend difficile la vérification d'implémentations aux ordres élevés.

Néanmoins, dans une représentation d'un schéma de masquage utilisant des shares, les propriétés s'expriment en termes de shares. Ainsi, une des propriétés de sécurité les plus

répandues pour ce type de schéma est la propriété de *N*on-*I*nterference à l'ordre t, ou *t-NI* [Bar+15]. Pour vérifier la propriété t-NI, une implémentation doit garantir que chaque t-uplet de valeurs ait une distribution qui dépende au plus de t shares par entrée (toutes les entrées sont supposées avoir le même nombre de shares, ainsi que les sorties). Si le sharing est fait de manière uniforme, alors une implémentation qui vérifie la propriété t-NI vérifie également la propriété t-TPS ; celle-ci s'exprime par le fait qu'un t-uplet ne dépende pas de tous les shares, pour chaque entrée, ou bien on peut faire la vérification t-TPS classiquement en passant à une représentation avec masques et secrets via une association arbitraire des shares vers une représentation en masques et secrets.

Un des inconvénients de la propriété NI est qu'elle n'est pas composable : le chaînage de deux gadgets qui vérifient NI n'est pas forcément NI. Ainsi deux propriétés supplémentaires ont été définies afin de permettre la composabilité, et donc la vérification de systèmes plus complexes. Contrairement à TPS et NI, ces propriétés ne sont définies que pour les gadgets matériels stateless (dont les entrées ne changent pas au cours du temps) qui utilisent une représentation du masquage en shares.

La propriété de t-*S*trong *N*on-*I*nterference [Bar+16], ou *t-SNI*, est dérivée de la propriété NI, et impose que pour chaque t-uplet correspondant à une observation, où o est le nombre de signaux parmi les t qui sont des signaux de sortie du gadget, la distribution de ce t-uplet dépende au plus $t - o$ shares de chaque entrée du gadget. Une implémentation qui vérifie t-SNI vérifie trivialement t-NI.

Si t-SNI permet la composition de gadgets, un des problèmes est que cette propriété n'est pas vérifiée sur l'implémentation la plus simple du XOR de deux entrées séparées en shares, comme illustré dans la figure 3.2. Pour cette raison, la propriété de t-*P*robe *I*solating *N*on-*I*nterference, ou *t-PINI*, a été introduite [CS20]. Cette propriété exploite l'indice des shares des entrées et des sorties, et est définie informellement de la manière suivante : toute observation faite sur $t1$ signaux internes et un ensemble O de $t2$ indices de shares tels que $t1 + t2 = t$, doit vérifier que la distribution du n-uplet comprenant les $t1$ signaux internes et tous les signaux de sortie dont les shares sont dans O doit dépendre au plus des shares d'entrées dont l'indice est dans O et de $t1$ shares additionnels. Ainsi, le gadget simpliste représenté sur la figure 3.2 est trivialement 1-PINI, et PINI est composable. Enfin, une implémentation t-PINI est également t-NI.

Une implémentation est dite sécurisée par rapport à un modèle de fuite et pour une propriété de sécurité si cette propriété est satisfaite en tout instant et en tout point en considérant le modèle de fuite. Si une propriété n'est pas vérifiée, on dit que l'implémentation contient une fuite secrète (ou « fuit », par abus de langage) mais la notion de fuite est intrinsèquement liée au modèle de fuite et à la propriété.

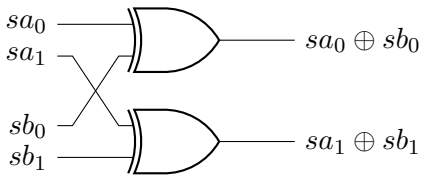


Fig. 3.2 : Gadget simpliste réalisant le XOR des shares de mêmes indices. Ce gadget est 1-PINI mais pas 1-SNI.

Des approches variées ont été proposées pour évaluer la sécurité d'implémentations masquées. Deux grandes classes de méthodologies coexistent : les approches statistiques, qui se basent sur une analyse statistique réalisée sur des traces simulées ou réelles de grandeurs physiques ; et les approches formelles, qui visent à prouver l'absence de fuite par rapport à un modèle de fuite pour toutes les exécutions possibles d'une implémentation.

3.2 Méthodes de vérification

Les méthodes de vérification permettent d'identifier, au cours de l'exécution d'une plateforme matérielle, les instants pour lesquels les grandeurs physiques sont liées aux valeurs secrètes. L'identification de tels instants n'indique cependant pas nécessairement qu'une attaque pourrait être mise en œuvre pour compromettre la sécurité de la plateforme.

3.2.1 Analyses statistiques

Tests statistiques

Une solution pour estimer, à partir de mesures réelles ou simulées, si une exécution présente des fuites est l'utilisation de tests statistiques pour vérifier l'indépendance des traces vis-à-vis des variables secrètes. Le test statistique le plus communément utilisé est le *test de Student* [Stu08], ou t-test. Plus concrètement, ce test estime la probabilité qu'un écart de moyenne entre deux échantillons de mesures soit au moins aussi important que celui observé, sous une hypothèse dite hypothèse nulle. Cette hypothèse, nommée H_0, correspond au fait que les distributions des échantillons de mesures sont identiques quelles que soient les valeurs secrètes. Une valeur de la statistique de test supérieure à 4,5 en valeur absolue indique, à condition d'avoir un nombre de traces suffisamment élevé, que la probabilité d'observer cet écart de moyenne sous H_0 est inférieure à 10^{-5}. Si ce seuil d'erreur est jugé suffisant par le réalisateur du test, une valeur de t-test supérieure à 4,5 conduit donc au rejet de H_0 et par extension à la conclusion que les valeurs de consommation dépendent des valeurs secrètes. En pratique, un attaquant sélectionne un

point temporel fixe dans les traces où il peut connaître la valeur manipulée. Il sépare ensuite les mesures à ce point en deux échantillons, selon la valeur ciblée dans le calcul ou une fonction de cette valeur (bit ou poids de Hamming, par exemple). La moyenne empirique $\overline{x_n}$ et la variance empirique non biaisée s_n^2 d'un échantillon ainsi que le calcul de la statistique t sont présentées dans l'équation 3.1, pour des échantillons de tailles n_0 et n_1.

$$\overline{x_n} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x_n})^2 \qquad t = \frac{|\overline{x_{n_0}} - \overline{x_{n_1}}|}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{3.1}$$

Le rapport signal sur bruit (*SNR*), le f-test, le G-test et bien d'autres tests statistiques peuvent aussi être utilisés pour caractériser les fuites, à partir de traces [SMY09 ; MM24 ; Mor+18 ; Gao+16].

Méthodes reposant sur l'analyse statistique pour la vérification de microarchitectures

Différentes méthodes reposant sur l'analyse statistique de traces réelles ou simulées ont été proposées au cours de la dernière décennie [Buh+22]. Les méthodes de simulation qui ne sont pas réalisées avec des modèles physiques de transistors utilisent des modèles de fuites, parfois implicites. La propriété de sécurité est elle aussi implicite ; à notre connaissance, les méthodes statistiques basées sur des modèles de fuite utilisent généralement la propriété TPS.

`ELMO` [MWO16] est un générateur de traces de consommation à partir de simulations de programmes binaires s'exécutant sur les cœurs ARM Cortex-M0 et ARM Cortex-M4. Le modèle de consommation a été estimé à l'aide de régressions linéaires sur des traces réelles par instruction et ne nécessite donc pas de modèle de fuite. `ELMO*` [She+21b] est une version améliorée qui intégre un modèle de consommation plus précis pour les microarchitectures pipelinées qui considère de petites séquences d'instructions ainsi que les accès mémoires. Ce générateur de traces peut ensuite être utilisé pour réaliser des tests statistiques tels que le t-test ou le SNR.

`EMSim` [Seh+20] est un générateur de traces d'émissions électromagnétiques à partir de simulations de programmes assembleurs. L'outil permet, à partir de nombreuses simulations de séquences d'instructions, d'obtenir un modèle précis d'émission dont les auteurs ont montré l'efficacité sur un processeur RISC-V avec un pipeline de 5 étages. Les traces générées sont comparées avec des mesures réelles du cœur sur FPGA. `EMSim` prend en compte les séquences d'instructions et de nombreux effets architecturaux comme les échecs

de prédiction de branchement, les bulles de pipeline, les échecs de récupération de données en cache.

`MAPS` [CGD18] est un autre générateur de traces de consommation opérant à partir des programmes assembleurs. En modélisant uniquement les registres du chemin de données du processeur ARM Cortex-M3 définis dans la description `HDL`, `MAPS` simule des programmes binaires ARM pour identifier les fuites, et plus particulièrement les fuites liées aux registres de pipeline.

`RTL-PSC` [He+19] réalise des simulations `HDL` à l'aide de l'outil propriétaire *Synopsys VCS* pour déterminer les transitions qui ont lieu au sein d'un circuit. Ces traces de transitions sont ensuite analysées à l'aide de métriques telles que la divergence de *Kullback-Leibler* ou le *success rate* pour identifier les fuites. L'outil permet de déterminer les modules (au sens `HDL` du terme) qui participent le plus aux fuites. `RTL-PSC` a été utilisé sur deux accélérateurs cryptographiques de l'AES distincts.

`PLAN` [Ars+20] est un outil de vérification prenant en entrée une description `HDL`, un programme à exécuter et un ensemble d'opérations sensibles sur le programme. L'outil réalise de nombreuses simulations en faisant varier les entrées sensibles et utilise le *Side-Channel Vulnerability Factor* pour identifier les modules `HDL` présentant des fuites. `PLAN` a été utilisé par les auteurs pour produire `PARAM`, une version renforcée contre les attaques par canaux auxiliaires du processeur RISC-V Shakti-C.

`Prolead` [MM22] est un outil de vérification qui réalise une simulation HDL (avec support des accélérateurs cryptographiques) puis des tests statistiques sur les valeurs simulées. En réalisant de multiples simulations avec des vecteurs d'entrée différents et en plaçant des points d'observation aux points clés d'un circuit, `Prolead` calcule l'indépendance statistique à l'aide du G-test et peut vérifier des circuits dans le modèle `RR d-probing`. La confiance en le verdict de l'outil croît avec le nombre de simulations. Il permet la vérification de circuits *stateful* (dont les entrées changent à travers le temps) complexes et permet l'utilisation de modèles de fuites allant jusqu'au modèle `RR d-probing`.

`Prolead_sw` [ZMM23] est une extension de `Prolead` permettant de simuler l'exécution de binaires ARM sur un modèle générique de processeur, créé par les auteurs. Ce modèle abstrait les sources communes de fuite identifiées dans les travaux de l'état de l'art.

Les approches statistiques permettent d'analyser des circuits et logiciels complexes. Cependant, leur utilisation ne donne pas de garanties d'absence de fuites et, lorsqu'une fuite est trouvée, ne permet pas d'en comprendre aisément la cause. Les tests statistiques qui rejettent l'hypothèse d'indépendance des traces avec les secrets ne donnent pas d'information sur la nature des données manipulées. Les signaux à l'origine des fuites ne sont d'ailleurs pas distinguables pour les méthodes reposant sur la génération de traces, auquel cas seul

l'instant temporel du rejet est retourné. De plus, les méthodes statistiques qui utilisent des modèles de fuite doivent prendre en compte l'ordre de vérification du modèle et par conséquent, réaliser les tests statistiques correspondant aux d combinaisons sur tous les points d'importance pour le modèle de fuite. À mesure que l'ordre croît, ces vérifications deviennent coûteuses, sans possibilité d'optimisation basée sur la nature des données manipulées.

3.2.2 Méthodes formelles

Les méthodes formelles de vérification d'implémentations logicielles et matérielles, dans un modèle de fuite donné, fonctionnent en deux étapes.

La première consiste en la construction d'un modèle du circuit représentant toutes les sources possibles de fuite dans le modèle de fuite. Les entrées du circuit sont étiquetées (ou typées) : *publiques* pour les données non sensibles, *masques* pour les variables aléatoires uniformément distribuées et *secrets* ou *shares* pour les données sensibles. Ensuite, une représentation symbolique de toutes les données, sensibles ou non, propagées à travers le circuit est extraite de celui-ci.

La seconde étape est l'application d'une méthode de vérification pour prouver que les représentations symboliques intermédiaires représentant les données ne fuient pas d'information par rapport aux variables sensibles dans un modèle de fuite donné et pour une propriété de sécurité, pour toutes les valeurs d'entrée possibles. Cette étape de vérification est elle-même réalisée de différente manière dont les trois suivantes sont les plus communes :

1. Les méthodes par *substitution* se basent sur des expressions symboliques pour représenter les valeurs intermédiaires propagées dans le circuit. Elles réalisent ensuite des remplacements successifs des sous-expressions masquées avec un masque bijectif dans l'objectif de conclure sur une expression qui ne fait plus apparaître de secrets, auquel cas l'expression est statistiquement indépendante des secrets. Introduite par [Bar+15] cette méthode est implémentée dans `maskVerif` [Bar+19], `VerifMSI` [MT23] et sa version précédente [MPH23].
2. Les méthodes par *inférence* utilisent aussi des expressions symboliques, en combinaison avec un type décrivant la distribution de la valeur du résultat de l'expression et des règles pour inférer la distribution de ses sous-expressions. Initialement proposée par [Ben+19] pour vérifier des binaires masqués au premier ordre, cette approche a été étendue par [Gao+19] pour permettre la vérification de codes plus complexes, à l'aide de règles d'inférence plus fines.

k	m_1	m_2	e	
0	0	0	0	$P(e=0\|k=0)=\frac{3}{4}$
		1	0	
	1	0	0	$P(e=1\|k=0)=\frac{1}{4}$
		1	1	
1	0	0	0	$P(e=0\|k=1)=\frac{3}{4}$
		1	1	
	1	0	0	$P(e=1\|k=1)=\frac{1}{4}$
		1	0	

(a) `e = (k ⊕ m1) & m2`.

k	m_1	e'	
0	0	0	$P(e'=0\|k=0)=\frac{1}{2}$
	1	1	$P(e'=1\|k=0)=\frac{1}{2}$
1	0	0	$P(e'=0\|k=1)=1$
	1	0	$P(e'=1\|k=1)=0$

(b) `e' = (k ⊕ m1) & m1`.

Fig. 3.3 : Énumération d'expressions symboliques, avec un secret `k` et deux masques m_1 et m_2. L'expression 3.3a est bien indépendante de la valeur du secret `k` puisque connaître la valeur de ce secret ne change pas la probabilité d'obtenir un résultat ou un autre sur `e`. L'expression 3.3b n'est pas indépendante de `k`, si `k` vaut `0` alors les deux valeurs possibles pour `e'` sont équiprobables, ce qui n'est plus vrai quand `k` vaut `1`.

3. Les méthodes basées sur la *satisfaction de contraintes* permettent l'expression de l'absence de fuite au niveau des valeurs intermédiaires sous la forme d'un problème de satisfiabilité résolu à l'aide d'un solveur SAT ou SMT comme dans `Coco` [HB21]. `SILVER` [KSM20] et plus récemment une version améliorée nommée `Prover` [ZCF24] utilisent des ROBDD afin de déterminer efficacement la distribution des valeurs intermédiaires. INDIANA [Bei+25] exploite des MTBDD pour vérifier des circuits avec une représentation probabiliste des fuites.

Ces méthodes de vérification, sauf celle utilisant les ROBDD, souffrent de faux positifs. Un faux positif se produit lorsqu'une méthode conclut incorrectement qu'une implémentation n'est pas sécurisée ou qu'elle ne peut pas conclure sur la sécurité de celle-ci. Pour distinguer les vrais positifs des faux positifs, les méthodes qui conservent les dépendances exactes des données sensibles propagées sur les interconnexions entre les portes logiques (les expressions symboliques, par exemple) peuvent avoir recours à l'énumération pour déterminer l'indépendance statistique d'un élément par rapport aux variables secrètes. Un exemple donnant l'intuition du fonctionnement de l'énumération est donné dans la figure 3.3. L'indépendance statistique d'une expression par rapport aux secrets revient dans ce cas à vérifier que la probabilité d'observer un résultat d'expression est la même quelles que soient les valeurs des secrets.

Cependant, l'énumération n'est une option viable que pour un nombre limité de bits symboliques au sein d'une expression, en raison de l'explosion combinatoire du nombre de cas à évaluer.

Les spécificités physiques liées aux transistors ont jusqu'ici été les justifications de la nécessité de vérifier les microarchitectures exécutant des logiciels au niveau matériel. À

ces spécificités s'ajoute un élément supplémentaire, défini par la suite sous le terme de *granularité de vérification*, qui est généralement implicite dans les modèles de fuite de l'état de l'art. La *granularité*, présentée dans la section 3.3, vise à harmoniser la taille physique des supports d'information avec celle des objets observables dans les modèles de fuites.

3.3 De l'importance de la largeur des supports

Le modèle `d-probing` est utilisé depuis les premiers travaux sur la vérification de masquage [ISW03] pour prouver la bonne application de schémas de masquage [ISW03 ; RP10]. Dans ce modèle, la largeur des « probes » est implicitement celle de la description analysée, soit 1 bit pour les circuits exprimés au niveau portes logiques [ISW03], soit plusieurs bits - la largeur des calculs intermédiaires dans une représentation choisie - pour des algorithmes. Par exemple, pour ce dernier cas, n bits pour la multiplication masquée dans $\mathbb{F}_{2^n}$ ou encore 8 bits pour l'AES masqué [RP10].

Les méthodes de vérification présentées dans la section précédente ciblent principalement un seul type d'implémentations, logicielles ou matérielles, et à un niveau de représentation spécifique, RTL ou portes logiques pour le matériel, et assembleur ou pseudo-code pour le logiciel. Ces méthodes utilisent implicitement des probes de 1 bit pour le matériel [Bar+19 ; KSM20 ; ZCF24] et 8 bits pour le logiciel [Ben+19 ; Gao+19].

Lorsque l'on souhaite vérifier des implémentations logicielles s'exécutant sur du matériel, une question persiste donc : quelle est la taille des probes à considérer ? Les vérifications ne concernant que le matériel sont aussi impactées par ce choix implicite.

La taille des probes définit quels signaux matériels ou variables logicielles peuvent être observés en une unique observation par un attaquant. En prenant l'exemple du modèle `1-probing`, un attaquant peut-il observer un signal matériel de 32 bits à la fois ou seulement un seul bit de ce signal à la fois ?

Lorsque des schémas de masquage logiciel sont vérifiés, la taille du support est supposée être celle de la donnée manipulée. Cependant, une donnée de type `int8_t` en `C` sera manipulée par une microarchitecture dans des registres matériels et des signaux de 32 bits pour un jeu d'instructions de 32 bits, par exemple. De plus, selon les optimisations ayant lieu pendant les étapes de compilation, plusieurs variables pourraient éventuellement être manipulées sur un seul de ces supports. Des exemples concrets de telles transformations sont données en sous-section 6.2.2.

La largeur des probes peut paraître arbitraire d'un point de vue matériel puisque dans le circuit final les interconnexions de portes d'un même signal ne sont pas nécessairement à

a_1a_0	m_0m_0	e_1e_0	Granularité 2 bits
00	00	00	$P(e = 00 \text{ resp. } 11 \mid a = 00) = \frac{1}{2}$
	11	11	$P(e = 10 \text{ resp. } 01 \mid a = 00) = 0$
01	00	01	$P(e = 00 \text{ resp. } 11 \mid a = 01) = 0$
	11	10	$P(e = 10 \text{ resp. } 01 \mid a = 01) = \frac{1}{2}$
10	00	10	$P(e = 00 \text{ resp. } 11 \mid a = 10) = 0$
	11	01	$P(e = 10 \text{ resp. } 01 \mid a = 10) = \frac{1}{2}$
11	00	11	$P(e = 00 \text{ resp. } 11 \mid a = 11) = \frac{1}{2}$
	11	00	$P(e = 10 \text{ resp. } 01 \mid a = 11) = 0$

a_0	m_0	e_0	Granularité 1 bit
0	0	0	$P(e_0 = 0 \mid a_0 = 0) = \frac{1}{2}$
	1	1	$P(e_0 = 1 \mid a_0 = 0) = \frac{1}{2}$
1	0	1	$P(e_0 = 0 \mid a_0 = 1) = \frac{1}{2}$
	1	0	$P(e_0 = 1 \mid a_0 = 1) = \frac{1}{2}$

a_1	m_0	e_1	
0	0	0	$P(e_1 = 0 \mid a_1 = 0) = \frac{1}{2}$
	1	1	$P(e_1 = 1 \mid a_1 = 0) = \frac{1}{2}$
1	0	1	$P(e_1 = 0 \mid a_1 = 1) = \frac{1}{2}$
	1	0	$P(e_1 = 1 \mid a_1 = 1) = \frac{1}{2}$

Tab. 3.1 : Soit l'opération de masquage $e = a \oplus m$ avec des variables larges de deux bits : a, m et e. Ces variables de deux bits sont composées de variables 1 bit concaténées ; Soient $a=(a_1, a_0)$, $m=(m_0, m_0)$ et $e=(e_1, e_0)$. m ne peut prendre que deux valeurs car un seul bit de masque est utilisé pour le composer. La distribution de e en considérant une granularité de deux bits n'est pas uniforme. Cependant, les distributions de e_1 et de e_0 sont bien uniformes lorsque considérées indépendamment.

proximité les unes des autres. Cependant, cette largeur a un réel impact sur les garanties de sécurité issues d'une vérification. Par exemple, un schéma de masquage utilisant un masque qui ne peut prendre que les valeurs `0b11` et `0b00`, en raison de l'utilisation d'un seul bit d'aléatoire pour sa création, pourrait être prouvé comme sécurisé en vérification en granularité 1 bit mais non sécurisé avec une granularité plus large. Un tel exemple est donné dans le tableau 3.1, dans lequel une même expression peut être considérée comme fuyant ou non un secret, selon la prise en compte de la granularité. La propriété de sécurité vérifiée lors d'une vérification en granularité bit ou multi-bits n'est donc pas la même, et une vérification de granularité plus large apportera toujours une garantie de sécurité plus forte.

Pour les vérifications matérielles, si l'objectif est de vérifier un schéma de masquage ayant été créé sous l'hypothèse que chaque bit d'un signal fuit indépendamment des autres, alors la vérification avec une granularité différente de 1 n'est pas pertinente. Cependant, pour détecter des non-respect de l'indépendance des masques ou pour vérifier des schémas de masquage reposant sur l'hypothèse que les bits d'un signal fuient conjointement, la vérification devrait être réalisée en considérant la largeur effective des supports matériels.

Pour expliciter les choix réalisés dans les vérifications de ce manuscrit, le modèle de fuite `d-probing` est par la suite raffiné en modèle `d-bit-probing` pour les vérifications en granularité bit et en modèle `d-sw-probing` (pour « support-wise » en anglais) pour les vérifications en granularité support.

Toutefois, une limite du modèle `d-sw-probing` est que la granularité de chaque vérification dépend de la largeur utilisée par le concepteur du matériel pour définir les signaux. Pour une conception en HDL par exemple, un concepteur pourrait théoriquement n'utiliser que

Support/Outil	Armistice	Coco	Closing the gap
Supports physiques de largeur variable	✓	✗	✓
Modèle de fuite	`(g=0,t=1) 1-sw`	`RR d-bit`	`RR d-bit`
Accélérateurs cryptographiques	✗	✓	✗
Automatique à partir du HDL	✗	✓	✗[1]
CPU(s) supporté(s)	Modèle de CPU générique	Coco-Ibex, Ibex, SweRV	Ibex
Modèle mémoire	✓	✓	✗
Propriétés de sécurité	TPS	NI[2]	TPS

Tab. 3.2 : Comparaison des méthodes formelles de co-vérification logiciels-matériels de l'état de l'art.

des signaux d'un bit. Cependant, il est raisonnable de considérer que la définition des signaux est cohérente avec l'utilisation qui en est faite et donc, que la granularité utilisée est pertinente.

> Élément de problématique 1 : Pour des raisons de clarté et de reproductibilité, la largeur des probes considérée dans une vérification devrait systématiquement être explicitée. De plus, pour les vérifications de matériel - et plus particulièrement de logiciel s'exécutant sur du matériel - les vérifications devraient être réalisées pour chaque signal en utilisant comme granularité la taille de ce signal.

3.4 Comparaison des méthodes formelles de co-vérification logiciels-matériels de l'état de l'art

Cette section est consacrée à la comparaison des approches existantes et des outils qui leur sont associés pour vérifier formellement des logiciels s'exécutant sur des microarchitectures de processeurs. Ces approches sont agrégées dans le tableau 3.2.

`Armistice` [GHM22] est un simulateur du chemin de données du Cortex-M3 avec une modélisation en boite noire d'une mémoire, permettant de vérifier des implémentations

1. La rédaction du contrat est une étape manuelle.
2. La propriété de sécurité n'est pas formellement mentionnée mais se rapproche, d'après notre compréhension, de celle de NI.

logicielles masquées et identifier les fuites qui apparaissent dans la microarchitecture. La modélisation du chemin de données est réalisée manuellement à partir de la description matérielle en `Verilog` du cœur. Cette étape est complexe, non portable à d'autres cœurs et source d'erreurs, indiquant donc qu'une méthode automatisée de modélisation serait préférable. `Armistice` supporte la vérification à granularité variable (la largeur des signaux faisant partie de la modélisation) en valeur et en transitions, sans glitch. Cela correspond au modèle `(g=0,t=1) 1-sw-probing`. Le modèle du Cortex-M3 produit par les auteurs n'est pas publié sous licence libre, ceux-ci ayant eu accès aux sources du processeur dans le cadre du programme Arm Academic Access (AAA) [Lim26a].

`Coco` [HB21] est un outil de vérification utilisant l'approche SAT pour représenter et vérifier l'absence de fuite lors des calculs intermédiaires réalisés par une microarchitecture exécutant un programme ou par un accélérateur cryptographique. Le support pour les cœurs RISC-V Ibex et SweRV a fait l'objet de plusieurs publications [Gig+21 ; GPM21]. En exploitant la description matérielle d'un circuit ainsi qu'un étiquetage des données sensibles, une simulation concrète est réalisée pour extraire une trace de simulation ainsi qu'un modèle du circuit. `Coco` peut être utilisé pour vérifier la présence de transitions et de glitches en exploitant la stabilité mais uniquement pour une granularité bit (modèle `RR d-bit-probing`). Une unique propriété de sécurité, que nous identifions comme étant NI, est prise en charge par l'outil, pour les processeurs comme pour les accélérateurs matériels. `Coco` est publié sous licence libre, documenté et propose de nombreux exemples pour simplifier sa prise en main, ce qui justifie son utilisation fréquente dans la littérature.

`Closing the gap` [HHB24] est une approche s'appuyant sur des contrats pour vérifier des implémentations logicielles exécutées par une microarchitecture, sans support pour les accélérateurs cryptographiques. Les contrats expriment, au niveau du jeu d'instructions, toutes les sources de fuites possibles de la microarchitecture. Ce contrat est d'abord validé vis-à-vis de la description matérielle du cœur, puis les implémentations logicielles sont vérifiées par rapport au contrat. La première étape est complexe mais ne doit être réalisée qu'une seule fois pour la microarchitecture. Pour l'instant, seul un contrat pour l'Ibex a été proposé par les auteurs et les mémoires ne sont pas modélisées. Les vérifications sont réalisées pour des modèles de fuite intégrant les transitions, les glitches et un support partiel de la stabilité. La stabilité doit être explicitement calculée au niveau du contrat, les éléments dont on souhaite prendre la stabilité en compte doivent donc apparaître dans le contrat ainsi que les éléments microarchitecturaux participant à son calcul. Les auteurs annoncent supporter des vérifications sur plusieurs bits mais pour des raisons de performance uniquement et non en considération des implications sur la propriété de sécurité. L'implémentation et le contrat de `Closing the gap` sont disponibles en ligne mais il n'est pas possible de lancer des vérifications, faute de documentation et de moyen de compiler le projet.

Élément de problématique 2 : La vérification formelle de microarchitectures exécutant des implémentations logicielles en considérant la largeur des supports, des modèles de fuite tels que le modèle `RR d-sw-probing`, et de propriétés de sécurité telles que NI ou SNI constitue un besoin non couvert par l'état de l'art.

Les méthodes de co-vérification logiciels-matériels ont pour objectif de guider les concepteurs vers des implémentations sécurisées. Les méthodes proposées se différencient par les garanties qu'elles apportent mais également par le degré d'explicabilité des fuites qu'elles identifient. Des informations précises permettent le développement et l'application de contre-mesures matérielles ou logicielles adaptées à une microarchitecture. La prochaine section présente un état de l'art des méthodes de renforcement automatique de la sécurité d'implémentations logicielles par l'insertion de contre-mesures logicielles ou par la transformation de programmes garantissant l'équivalence fonctionnelle mais supprimant les fuites lors de l'exécution de ceux-ci sur une microarchitecture.

3.5 État de l'art des méthodes de renforcement automatique de la sécurité de programmes par transformation logicielle

De nombreux travaux ont étudié le renforcement automatique du niveau de sécurité de logiciels s'exécutant sur des microarchitectures via des modifications au niveau logiciel. Des méthodes, présentées dans la sous-section 3.5.1, opèrent en amont de l'exécution, en appliquant automatiquement des contre-mesures de masquage ou de dissimulation sur les opérations sensibles. D'autres méthodes, présentées dans la sous-section 3.5.2, exploitent des simulations ou des exécutions réelles pour identifier les instructions causant des fuites puis appliquer des contre-mesures de dissimulation sans nécessiter d'informations sur le programme exécuté.

3.5.1 Sécurisation en amont de l'exécution

`Compiler Assisted Masking` [Mos+12] propose une méthode de masquage automatique à la compilation de programmes non-masqués. Cette méthode prend en entrée un programme dont les entrées sensibles sont annotées comme *publiques* ou *secrètes* et infère la nature des variables. Le code du compilateur n'est pas fourni et aucun exemple de code compilé n'est donné.

`SC Masker` [EW14] est un outil basé sur LLVM [LA04] proposant le masquage automatique de programmes `C` non sécurisés. Les entrées sont annotées et une méthode basée sur un solveur SMT est utilisée pour garantir que les programmes masqués proposés sont parfaitement masqués et sont toujours corrects vis-à-vis du calcul réalisé.

A Code Morphing Methodology to Automate Power Analysis Countermeasures [ABP12] propose d'intégrer automatiquement des contre-mesures de dissimulation sur des programmes dont les opérations sensibles sont annotées. Les contre-mesures intégrées sont le polymorphisme de code dynamique, le réordonnancement d'opérations ainsi que la permutation des accès aux tableaux. L'implémentation de la méthode n'est pas rendue disponible par les auteurs.

De nombreuses autres approches, conceptuellement similaires, ont été proposées au cours de la dernière décennie [Bel+20 ; Cou+16 ; TS24 ; Ama+11 ; RLT15].

`PoMMES` [ZM24] est un compilateur de programmes `C` masqués visant à produire des implémentations `Assembleur` sécurisées pour un modèle abstrait de processeur utilisant le jeu d'instructions `ARMv6-M`. À partir d'une analyse statique des registres utilisés pour conserver des variables sensibles tout au long du programme, le compilateur génère des instructions de nettoyage et garantit que des éléments permettant de reconstituer des secrets ne sont jamais simultanément conservés dans les registres. Le modèle de fuite utilisé prend en compte les transitions et les glitches, mais la fiabilité des programmes sécurisés dépend de la pertinence de ce modèle. L'outil est accessible sous licence libre avec des exemples de programmes.

Les méthodes présentées sécurisent des programmes soit en implémentant des contre-mesures de dissimulation ou de nettoyage, soit en intégrant le masquage mais de manière insuffisante face aux modèles de fuite de l'état de l'art qui prennent en compte la microarchitecture. Aucune méthode de sécurisation en amont ne permet, à notre connaissance, de considérer la microarchitecture complète sur laquelle sera exécuté le programme sécurisé.

3.5.2 Sécurisation en aval de l'exécution

A First Step Towards Automatic Application of Power Analysis Countermeasures [Bay+11] propose une méthode de protection automatique de programmes. Le programme source est compilé et de nombreuses mesures sont réalisées pendant son exécution avec différentes valeurs d'entrées sensibles. Une métrique basée sur l'information mutuelle est utilisée pour déterminer les instructions qui présentent des fuites. Enfin, l'instruction identifiée comme présentant une fuite est remplacée par la même instruction mais avec le registre de destination préchargé avec une valeur aléatoire. Il s'agit donc de contre-mesures de

dissimulation, automatiquement appliquées lors de l'identification de fuites sur une cible réelle.

ROSITA [She+21b][3] et ROSITA++ [She+21a][4] sont deux piles logicielles de réécriture de code, capables de sécuriser respectivement des implémentations masquées au premier ordre et aux ordres supérieurs. Elles exploitent la détection des instructions présentant des fuites au sein de traces fournies par le générateur de traces de consommation ELMO* [She+21b]. Les contre-mesures appliquées sont des contre-mesures de dissimulation utilisant un registre dédié. Elles sont basées sur l'instruction qui est identifiée comme source de la fuite. En particulier, les auteurs identifient quatre types de fuite (mémoire, registres, rotations et opérandes) et définissent une contre-mesure pour chaque type. Les outils sont disponibles sous licence libres.

Les méthodes en aval de l'exécution présentées proposent d'exploiter des simulateurs de traces de consommation ou des traces réelles pour identifier les instructions à l'origine de fuites. Elles proposent ensuite de corriger automatiquement ces fuites en exploitant les informations sur les instructions exécutées à l'instant de la fuite. Le défaut principal de cette méthodologie est que sur des processeurs pipelinés, l'instruction actuellement en cours d'exécution n'est pas nécessairement celle à l'origine de la fuite. De plus, la nature (*secret* ou *masque* par exemple) des éléments manipulés à l'instant des fuites identifiées n'est pas connue et mène donc à des contre-mesures génériques sans compréhension des causes des fuites.

> Élément de problématique 3 : La sécurisation automatique de programmes s'exécutant sur des microarchitectures reste une question ouverte. Les contre-mesures actuellement proposées par les méthodes qui considèrent la microarchitecture complète ne permettent pas de produire des programmes sécurisés dans les modèles de fuite de l'état de l'art en exploitant les informations de la microarchitecture.

3. Disponible sur github : https://github.com/0xADE1A1DE/Rosita
4. Disponible sur github : https://github.com/0xADE1A1DE/Rositaplusplus

4 Problématique et plan

Les sections précédentes ont permis d'identifier et de contextualiser la problématique de ces travaux.

Problématique

La vérification de microarchitectures et d'accélérateurs cryptographiques implémentant des contre-mesures de masquage logiciel et/ou matériel est au cœur de ces travaux. Le choix des approches formelles repose sur l'importance des garanties qu'elles apportent quant à la sécurité finale d'une implémentation, à condition d'utiliser un modèle de fuite pertinent. Les méthodes actuelles de vérification formelle ne supportent pas la vérification dans le modèle le plus pertinent en prenant simultanément en compte la largeur des supports (modèle `RR d-sw-probing`) (Élément de problématique 1). Par ailleurs, aucune méthode ne permet de vérifier formellement, avec une granularité variable, à la fois des gadgets pour des propriétés de sécurité variables et des logiciels s'exécutant sur des microarchitectures (Élément de problématique 2). De plus, certaines de ces méthodes nécessitent des informations précises sur les microarchitectures étudiées, qui doivent être définies manuellement, ou ne passent pas à l'échelle sur des circuits ou des programmes complexes. D'autre part, la conception de programmes sécurisés pour les microarchitectures est une tâche fastidieuse. Les méthodes actuellement proposées pour réaliser automatiquement ces sécurisations ne permettent pas la prise en compte d'une microarchitecture complète ou ne permettent pas d'identifier avec certitude les instructions responsables des fuites. Sans cette certitude et sans information sur la nature des éléments manipulés, les contre-mesures proposées ne permettent pas la sécurisation dans des modèles de fuites prenant en compte la microarchitecture (Élément de problématique 3).

Pour répondre à cette problématique, le chapitre 5 est dédié à la formalisation d'une méthode de représentation et de vérification automatique et efficace de microarchitectures exécutant des implémentations logicielles ainsi que d'accélérateurs cryptographiques. Cette méthode est implémentée dans `aLEAKator`, un nouvel outil supportant la vérification prenant en compte la largeur des supports et toutes les combinaisons des modèles de fuites du modèle `d-sw-probing` au modèle `RR d-sw-probing`. Cette méthode supporte les

propriétés de sécurité t-NI et t-TPS pour toutes les cibles matérielles ainsi que la propriété t-SNI pour les accélérateurs cryptographiques. Les verdicts de vérification obtenus avec aLEAKator sont comparés en détail à des mesures réelles, permettant de conforter le choix des modèles de fuite de l'état de l'art ainsi que la pertinence de la modélisation.

Le chapitre 6 propose ensuite une étude approfondie des sources de fuites dans les microarchitectures non-sécurisées, des moyens logiciels à disposition pour éviter ces fuites, puis d'une méthode pour sécuriser automatiquement des programmes exécutés sur une microarchitecture. Cette méthode est une preuve de concept exploitant la richesse des informations fournies par les méthodes de simulation et de vérification formelles comme aLEAKator, dont la pertinence est démontrée en réalisant automatiquement la sécurisation d'une implémentation C de l'AES masqué avec le schéma de Herbst [HOM06] pour le processeur ARM Cortex-M4 et dans le modèle RR 1-sw-probing.

Enfin, la partie III conclut ce manuscrit et propose des pistes à explorer pour poursuivre ces travaux.

Ces travaux s'inscrivent dans le cadre du projet ANR *IDROMEL* [1] (Improving the Design of secure systems by a Reduction Of Micro-architectural Effects on side-channeL attacks). Les objectifs du projet sont d'investiguer l'impact des microarchitectures pour la sécurisation d'implémentations face aux canaux auxiliaires et de contribuer à la conception de systèmes sécurisés.

Les objectifs du projet sont :

— Concevoir des systèmes sécurisés contre les attaques par canaux auxiliaires basées sur l'analyse de la consommation électrique et des émissions électromagnétiques, pour des architectures allant des objets connectés aux téléphones.
— Évaluer l'impact des microarchitectures de processeurs sur ces attaques, afin de développer des outils automatisés d'évaluation de la sécurité et de nouveaux processeurs intégrant des contre-mesures matérielles.

1. Page du projet disponible sur le site de l'ANR : https://anr.fr/Projet-ANR-20-CE39-0010

Deuxième partie

Contributions

5 Modélisation et vérification de micro-architectures et d'accélérateurs matériels

Ce chapitre est dédié à la définition d'une nouvelle méthode permettant de vérifier qu'un accélérateur matériel ou une microarchitecture exécutant un logiciel, au cours de l'entièreté de son exécution, ne présente aucun défaut de masquage dans un modèle de fuite donné et pour une propriété de sécurité donnée. Le contenu est adapté de l'article « aLEAKator : HDL Mixed-Domain Simulation for Masked Hardware & Software Formal Verification » [Ami+26], publié à TCHES (Transactions on Cryptographic Hardware and Embedded Systems) dans le second volume de 2026. L'article a reçu le badge « Artifact Functional », attestant de la disponibilité des sources sous licence libre ainsi que de la possibilité d'utiliser l'implémentation d'`aLEAKator`.

Lors de la conception de circuits, il est usuel de réaliser des simulations logiques. Les simulations `HDL` ont pour but de simuler, dans le domaine booléen [1], l'évolution des signaux composant un matériel lorsqu'il est soumis à des séquences d'entrées. Ces simulations sont généralement réalisées pour valider le fonctionnement d'un circuit et permettent de générer des traces visualisables de l'évolution de chaque signal du circuit à travers le temps, aussi appelées *VCD* (pour *V*alue *C*hange *D*umps). La *simulation en domaines mixtes* correspond à l'enrichissement d'une telle simulation avec des domaines supplémentaires pour chaque signal, permettant d'exposer pendant la simulation les informations nécessaires à la vérification dans les modèles de fuite définis dans la partie précédente. La simulation en domaines mixtes proposée dans ce manuscrit associe de nouvelles informations aux signaux :

— Une expression symbolique, décrivant le calcul réalisé sur des masques, shares, secrets ou valeurs publiques (constantes ou abstraites)

— Un ensemble de fuite, nommé *LeakSets*, qui représente toutes les observations possibles en présence de glitches

— Une valeur de stabilité, pour le raffinage éventuel du *LeakSets* et nécessaire pour le modèle `RR 1-sw-probing`

1. Certaines simulations proposent d'enrichir les informations booléennes avec des états supplémentaires pour les valeurs inconnues ou non initialisées par exemple. Ces états ne sont pas considérés dans ces travaux et les circuits sont supposés initialisés en tout points.

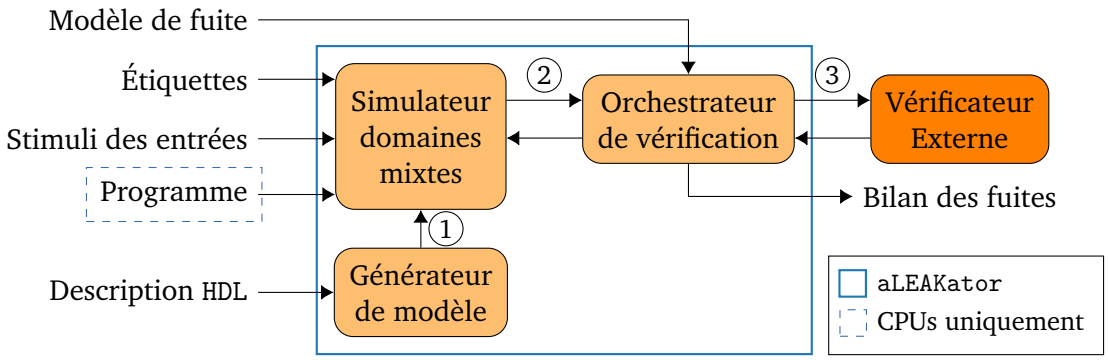


Fig. 5.1 : Vue d'ensemble d'`aLEAKator`.

Les valeurs booléennes, appelées *valeurs concrètes* dans cette étude, sont aussi calculées au cours de la simulation à des fins de débogage et de vérification de la cohérence de l'état symbolique. La simulation en domaines mixtes n'est donc plus réalisée comme usuellement sur le domaine booléen mais à travers quatre domaines interdépendants : le domaine concret, le domaine symbolique, le domaine des ensembles de fuite et le domaine de la stabilité.

Une vue globale de la méthode est présentée dans la figure 5.1. Les entrées sont la description `HDL` de l'accélérateur ou de la microarchitecture, une séquence de stimuli pour chaque signal d'entrée à chaque cycle (par exemple une horloge ou un signal de reset) ainsi que des étiquettes qui font le lien entre les signaux et la nature des variables, c'est-à-dire *share*, *secret*, *masque*, *publique* ou concret. Dans le cas des microarchitectures, un programme à exécuter, donné sous la forme d'un binaire, est aussi utilisé. Enfin, le modèle de fuite (pouvant aller jusqu'au `RR d-sw-probing`) est donné pour indiquer les vérifications à réaliser.

À partir de la description `HDL` de l'accélérateur ou de la microarchitecture, un modèle exposant toutes les valeurs intermédiaires internes au matériel, qui doivent être vérifiées, est généré (1). Ce modèle est ensuite simulé (2) cycle par cycle par l'orchestrateur de vérification, en exploitant les stimuli d'entrée et les étiquettes qui précisent la nature des variables associées aux signaux. L'orchestrateur, après chaque cycle de simulation, détermine les éléments qui doivent être vérifiés, en correspondance avec le modèle de fuite donné en entrée et la propriété de sécurité à valider. Un ensemble réduit, mais couvrant tous les éléments à vérifier, est construit puis donné à un vérificateur externe (3). Le verdict de vérification est raffiné avec les informations disponibles dans l'orchestrateur pour déterminer précisément les causes des fuites rencontrées. Les fuites sont ensuite indiquées à l'utilisateur avec des informations précises, notamment : le cycle de simulation, la nature des fuites (les expressions et ensembles de fuite) ainsi que les informations spatiales (fichier `HDL` source, numéro de ligne et nom du signal). Ces informations doivent ensuite être

exploitées par le concepteur du logiciel ou du matériel pour corriger les fuites. Les étapes ② et ③ sont réalisées de manière entrelacée jusqu'à ce que la séquence complète des stimuli d'entrée ait été simulée.

5.1 Modélisation et simulation de composants matériels

Cette première section est séparée en deux parties. Une première consacrée à la modélisation des composants matériels et une seconde qui exploite cette modélisation pour réaliser des simulations puis des simulations en domaines mixtes.

5.1.1 Modélisation des composants matériels

Les accélérateurs matériels et les microarchitectures sont des circuits, potentiellement séquentiels, pouvant être modélisés comme des machines de Mealy fonctionnelles. Dans le cas des microarchitectures, le programme binaire est chargé dans des mémoires. Ces mémoires, qui contiennent les données et les instructions, font partie intégrante du circuit.

La modélisation des circuits proposée dans cette section fait abstraction de l'utilisation qui sera faite à terme du modèle (simulation en domaines mixtes), et peut être utilisée pour formaliser tous types de simulations de circuits, notamment une simulation concrète. La valeur d'un signal à un instant donné représente l'ensemble des domaines de la simulation et doit être projetée, par exemple dans le domaine concret, pour obtenir son évaluation dans ce domaine.

Définition 5 (Circuit)**.** Un circuit $\mathcal{C}$ est représenté par le tuple $\langle W, G, init\rangle$ avec G l'ensemble des portes logiques, qui peuvent être combinatoires ($\in C$) ou séquentielles ($\in R$) et avec $init$ le vecteur des valeurs initiales des éléments de R. W, l'ensemble des signaux, est composé de I, l'ensemble des entrées primaires, O, l'ensemble des sorties primaires et de l'ensemble des autres signaux internes ($\in G \times G$). Chaque signal $w \in W$ a une largeur définie avec $|w|$.

Définition 6 (Porte combinatoire)**.** Une porte combinatoire $g \in C$ est définie par le tuple $\langle f_g, W_g^{in}, w_g^{out}\rangle$ avec f_g sa fonctionnalité, W_g^{in} l'ensemble des signaux en entrée et w_g^{out} son unique signal de sortie. Les entrées peuvent être multiples mais la sortie est unique.

Définition 7 (Porte séquentielle)**.** Une porte séquentielle est une porte r définie par le tuple $\langle w_r^{in}, w_r^{out}\rangle$ avec w_r^{in} et w_r^{out} son unique signal d'entrée et son unique signal de sortie. Les portes séquentielles comprennent les registres et les cellules mémoires individuelles.

Définition 8 (Machine de Mealy fonctionnelle)**.** La machine de Mealy fonctionnelle $\mathcal{M}(\mathcal{C})$ associée à un circuit $\mathcal{C}$ est décrite par le tuple $\langle \Sigma_I, \Sigma_O, \Sigma_R, \delta, \lambda, \alpha_R^0 \rangle$ avec Σ_I, Σ_O, Σ_R respectivement les ensembles de toutes les valeurs sur I, O et les signaux de sortie des éléments dans R, ce dernier étant aussi nommé ensemble des états du circuit. Dans le contexte d'une simulation booléenne, les valeurs possibles sont $\Sigma_I = 2^{|I|}$, $\Sigma_O = 2^{|O|}$, $\Sigma_R = 2^{|R|}$.

Dans la définition 8, $\alpha_R^0 \in \Sigma_R$ est l'état initial du circuit $\mathcal{M}(\mathcal{C})$. Cet état correspond au vecteur $init$ du circuit $\mathcal{C}$. Plus généralement, $\alpha^t \in \Sigma_W$ dénote les valeurs de tous les signaux de $\mathcal{C}$ à un cycle t et α_X^t la restriction de α^t aux signaux de l'ensemble de signaux X. Ainsi, α_R^t correspond à l'état du circuit $\mathcal{C}$ au cycle t ; α_I^t et α_O^t dénotent respectivement les valeurs des signaux d'entrée et de sortie du circuit $\mathcal{C}$ au cycle t. $\delta : \Sigma_I \times \Sigma_R \to \Sigma_R$ est la fonction de transition et $\lambda : \Sigma_I \times \Sigma_R \to \Sigma_O$ est la fonction de génération. Leurs applications calculent respectivement l'état du circuit pour le prochain cycle et les valeurs des signaux de sortie pour le cycle courant, à partir des valeurs courantes des entrées et de l'état courant tel que défini dans l'équation 5.1.

Exemple 5. La représentation d'une machine de Mealy fonctionnelle est donnée dans la figure 5.2. La combinatoire est partagée entre les fonctions de transition et de génération et les portes séquentielles sont rassemblées en une barrière synchrone.

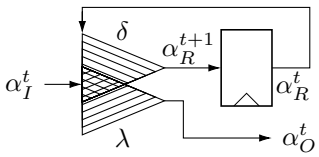


Fig. 5.2 : Représentation graphique des machines de Mealy fonctionnelles.

$$\forall t \geq 0 \begin{cases} \alpha_R^{t+1} = \delta(\alpha_R^t, \alpha_I^t) \\ \alpha_O^t \ = \lambda(\alpha_R^t, \alpha_I^t) \end{cases} \tag{5.1}$$

Les fonctions δ et λ sont invariantes dans le temps car la structure du circuit est figée. De plus, comme elles calculent une valeur pour chacun des éléments de R et O respectivement, elles peuvent toutes deux être représentées comme des vecteurs de fonctions, avec un élément pour chaque signal (de R ou O) à calculer.

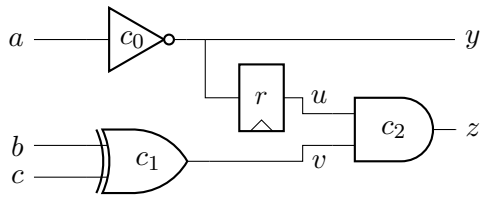
Les fonctions δ et λ sont construites en appliquant un parcours inductif de la structure du circuit $\mathcal{C}$ (algorithme 1) à partir respectivement des signaux en entrée de chacune des portes séquentielles (équation 5.2) et des sorties primaires (équation 5.3).

$$\delta = \left| \begin{array}{ll} \textsc{build}(w_{r_0}^{in}) & \\ \vdots & r_0 \ldots r_n \in R \\ \textsc{build}(w_{r_n}^{in}) & \end{array} \right. \tag{5.2}$$

$$\lambda = \left| \begin{array}{ll} \textsc{build}(w_0) & \\ \vdots & w_0 \ldots w_n \in O \\ \textsc{build}(w_n) & \end{array} \right. \tag{5.3}$$

Algorithme 1 : Parcours inductif de la structure d'un circuit $\mathcal{C}$ à partir d'un signal w.

Entrée : Un signal w, qui est soit une entrée primaire, soit la sortie d'une porte g
Sortie : La fonctionnalité associée à w jusqu'aux entrées primaires et aux portes séquentielles
Fonction BUILD(w) **:**
  si $w \in I$ **alors retourner** w
  sinon si $g \in R$ **alors retourner** w_g^{out}
  sinon retourner $f_g(\dots,\text{BUILD}(w_i),\dots)$ ▷ Avec w_i, les signaux de W_g^{in}



(a) Vue structurelle du circuit.

$$\delta = \left| \begin{array}{lll} \delta_r & \equiv f_{c_0}(\alpha_a) & \end{array} \right. \tag{5.4}$$

$$\lambda = \left| \begin{array}{lll} \lambda_y & \equiv f_{c_0}(\alpha_a) & \\ \lambda_z & \equiv f_{c_2}(\alpha_u, \quad \alpha_v \quad) & \\ & \equiv f_{c_2}(\alpha_r, \quad f_{c_1}(\alpha_b, \alpha_c) \quad) & \end{array} \right. \tag{5.5}$$

(b) Fonctions δ et λ correspondantes.

Fig. 5.3 : (a) Un circuit d'exemple avec $I = \{a, b, c\}$, $O = \{y, z\}$ et les signaux internes $\{u, v\}$. L'ensemble des portes combinatoires est $C = \{c_0, c_1, c_2\}$ et celui des portes séquentielles $R = \{r\}$. Dans cet exemple, f_{c_0} est une porte de fonctionnalité NON, f_{c_1} de fonctionnalité XOR et f_{c_2} de fonctionnalité ET. (b) Les fonctions de transition et de génération de la machine de Mealy fonctionnelle $\mathcal{M}(\mathcal{C})$.

Exemple 6. La figure 5.3b donne les fonctions δ (équation 5.4) et λ (équation 5.5) de la machine de Mealy associée au circuit d'exemple de la figure 5.3a et obtenues en utilisant respectivement les équations 5.2 et 5.3. Ce circuit est composé de portes combinatoires et séquentielles.

Définition 9 (Trace de simulation)**.** À partir d'un état initial $\alpha_R^0 \in \Sigma_R$, la machine de Mealy, pilotée par une séquence de configurations d'entrée $\alpha_I^0 \dots \alpha_I^{n-1} \dots$ génère, à travers des applications successives de δ et λ, respectivement une séquence d'états $\alpha_R^1 \dots \alpha_R^n \dots$ et une séquence de configurations de sortie $\alpha_O^0 \dots \alpha_O^{n-1} \dots$.

Une trace de simulation $\sigma(\alpha_R^0, \alpha_I^0 \dots \alpha_I^{n-1} \dots)$ est représentée comme l'entrelacement de ces suites d'états et de transitions.

$$\sigma(\alpha_R^0, \alpha_I^0 \dots \alpha_I^{n-1} \dots) = \alpha_R^0 \xrightarrow[\alpha_O^0]{\alpha_I^0} \alpha_R^1 \to \dots \to \alpha_R^{n-1} \xrightarrow[\alpha_O^{n-1}]{\alpha_I^{n-1}} \alpha_R^n \xrightarrow[\dots]{\dots} \dots$$

Les éléments nécessaires pour représenter des circuits ont été présentés dans cette section. La prochaine section propose l'enrichissement de cette modélisation, pour une application à la simulation en domaines mixtes.

5.1.2 Simulation en domaines mixtes

L'application des fonctions δ et λ associées au circuit $\mathcal{C}$ représente la simulation de ce circuit pendant un cycle d'horloge, et leurs applications successives représentent la simulation de ce circuit sur plusieurs cycles. La simulation usuelle des circuits est réalisée dans le domaine *concret* consiste à calculer la valeur logique associée à chaque signal.

Les propriétés de sécurité encapsulées dans les modèles de fuite présentés précédemment nécessitent de simuler les interactions exactes entre les données sensibles à travers les cycles et pour des effets physiques divers. Cette section introduit la *simulation symbolique*, qui rend compte de la contribution de toutes les variables sensibles à un calcul, à tout instant et en tout point du circuit. La simulation en domaines mixtes enrichit la simulation concrète en propageant des expressions symboliques qui représentent les calculs réalisés sur chacun des signaux vis-à-vis des valeurs sensibles, menant à une simulation dans le domaine *symbolique*. L'ensemble des expressions symboliques intermédiaires observables en présence de glitches sur les signaux est calculé dans le domaine des *ensembles de fuite* et les informations de stabilité des signaux sont calculées dans le domaine de la *stabilité*, pour optionnellement raffiner le domaine des ensembles de fuite.

Ces domaines ainsi que le calcul de la propagation des valeurs dans chaque domaine pour chaque type de porte logique pouvant être traversée sont formellement spécifiés. Cette spécification passe par la définition, pour chaque fonctionnalité de porte et chaque domaine, d'une fonction calculant la fonctionnalité dans le domaine.

1. **Domaine concret** : $\mathcal{B}$, l'ensemble des vecteurs booléens de toutes tailles, $\mathcal{B} = \bigcup_{i\in\mathbb{N}^+} \mathbb{B}^i$, avec $\mathbb{B} = \{0, 1\}$.
2. **Domaine symbolique** : $\mathcal{E}$, l'ensemble des expressions symboliques de toutes largeurs, $\mathcal{E} = \bigcup_{i\in\mathbb{N}^+} \mathbb{E}^i$, avec $\mathbb{E}^i$ l'ensemble des expressions e de largeur i définies comme :

$$e \in \mathbb{E}^i \Leftrightarrow e := \begin{cases} \texttt{CST}(x) & \text{une constante } x \in \mathbb{B}^i \\ \texttt{SYMB}(y) & y, \text{ une variable symbolique de largeur } i \\ \texttt{OP}(\dots, t_j, \dots) & \text{un opérateur } \texttt{OP}, \text{ de sortie de largeur } i \text{ et } t_j \in \mathcal{E} \end{cases} \tag{5.6}$$

Le tableau 5.1 résume les opérateurs pouvant être utilisés comme constructeurs d'expressions symboliques.

OP_XOR	OP_ADD	OP_LSL	OP_CONCAT
OP_AND	OP_MUL	OP_LSR	OP_EXTRACT
OP_OR	OP_POW	OP_ASR	OP_ZEXT
OP_NOT	OP_SUB	ARRAY	OP_SEXT

Tab. 5.1 : Liste des opérateurs de construction des expressions symboliques.

3. **Domaine des ensembles de fuite** : $\mathcal{L}$, l'ensemble des vecteurs de toutes tailles dont les éléments sont des ensembles d'expressions symboliques $\mathcal{L} = \bigcup_{i \in \mathbb{N}^+} \mathbb{L}^i$, avec $\mathbb{L} = \mathcal{P}(\mathcal{E})$.
4. **Domaine de la stabilité** : $\mathcal{S}$, l'ensemble des vecteurs booléens de toutes tailles, représentant la stabilité, $\mathcal{S} = \bigcup_{i \in \mathbb{N}^+} \mathbb{S}^i$, avec $\mathbb{S} = \{0, 1\}$.

Définition 10 (Valeur d'un signal)**.** La valeur d'un signal w de largeur $|w|$ à un cycle t est le tuple $\alpha_w^t = \langle {}^c\alpha_w^t, {}^e\alpha_w^t, {}^l\alpha_w^t, {}^s\alpha_w^t \rangle \in \mathbb{B}^{|w|} \times \mathbb{E}^{|w|} \times \mathbb{L}^{|w|} \times \mathbb{S}^{|w|}$, avec ${}^c\alpha_w^t$ la valeur concrète du signal au cycle t, ${}^e\alpha_w^t$ son expression symbolique, ${}^l\alpha_w^t$ son ensemble de fuite et ${}^s\alpha_w^t$ sa stabilité.

Définition 11 (Extraction de bit (ou slicing))**.** La représentation $w[i]$ est utilisée pour extraire le $i^{\text{ème}}$ bit du signal w, avec $i < |w|$.

La fonctionnalité de chaque porte g est définie pour chacun des domaines au travers des fonctions ${}^c f_g$, ${}^e f_g$, ${}^l f_g$ et ${}^s f_g$. Dans un objectif de clarté, seulement certaines de ces fonctions sont données dans chaque domaine (tableaux 5.2, 5.3 et 5.4). La liste complète est cependant donnée dans l'annexe B.

Exemple 7. Le circuit de la figure 5.4 est annoté avec les valeurs de chaque signal. Les expressions symboliques sont simplifiées (par exemple : $\texttt{OP_AND}(\texttt{SYMB}(x), \texttt{CST}(0))$ devient $\texttt{CST}(0)$) mais la démarche de simplification n'est présentée que plus tard, dans la section 5.3 (Génération et vérification d'expressions avec VerifMSI++) de ce manuscrit.

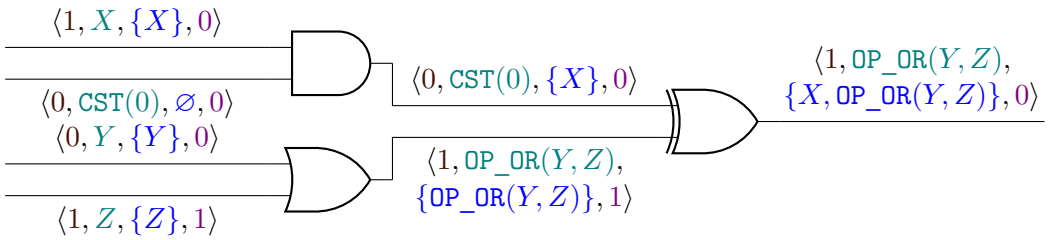


Fig. 5.4 : Valeurs des signaux du circuit, avec $X = \texttt{SYMB}(x)$, $Y = \texttt{SYMB}(y)$ et $Z = \texttt{SYMB}(z)$.

Pour un signal w à un cycle t, la valeur du signal α_w^t est calculée par l'application des fonctions d'évaluation dans chaque domaine, détaillées dans les sections concernant les valeurs concrètes et symboliques, les valeurs de stabilité et les ensembles de fuite. De plus, comme I représente l'ensemble des signaux qui sont des entrées primaires, une évaluation est fournie pour chaque domaine à travers la séquence d'entrée pour chaque cycle, conformément à l'équation 5.7. Une valeur concrète $\alpha_{i \in I}^t$ doit être donnée pour chaque entrée. Si aucune initialisation symbolique n'est spécifiée, la valeur symbolique

par défaut utilisée est construite à partir de la valeur concrète : $\texttt{CST}(\alpha^t_{i\in I})$. En présence de glitches, les valeurs symboliques des signaux d'entrée sont observables. L'expression symbolique de chaque bit des signaux d'entrée est donc utilisée par défaut dans le domaine des ensembles de fuite. Pour la stabilité, il n'est pas possible de déterminer si les signaux en amont des entrées sont stables ; aucune supposition ne doit donc être faite concernant leur stabilité : ils sont ainsi considérés instables par défaut.

$$\alpha^t_{i\in I} = \left\langle {}^c\alpha^t_i \in \mathbb{B}^{|i|}, {}^e\alpha^t_i = \begin{cases} \texttt{CST}({}^c\alpha^t_i) \\ \texttt{SYMB}(nom) \end{cases}, {}^l\alpha^t_i[j] = \{{}^e\alpha^t_i[j]\}\forall j < |i|, {}^s\alpha^t_i = \vec{0} \right\rangle \tag{5.7}$$

5.1.3 Évaluations concrètes et symboliques

Les fonctions d'évaluation, appliquées à un signal w à un cycle t, pour le domaine concret (équation 5.8) et pour le domaine symbolique (équation 5.9) sont respectivement notées ${}^c eval(w,\, t)$ et ${}^e eval(w,\, t)$. Elles sont définies inductivement, jusqu'aux signaux d'entrée ou jusqu'aux portes séquentielles incluses dans le cône de dépendance combinatoire du signal.

$${}^c eval(w,t) := \begin{cases} {}^c\alpha^t_w & \text{si } w \in I \\ t = 0\,?\ {}^c\alpha^0_w : {}^c\alpha^{t-1}_{w^{in}_r} & \text{si } \exists r \in R \text{ t.q. } w = w^{out}_r \\ {}^c f_g(\dots, {}^c eval(w_i,t), \dots) & \text{sinon, } \exists g \text{ t.q. } w = w^{out}_g, \forall w_i \in W^{in}_g \end{cases} \tag{5.8}$$

Pour ces deux domaines, si w est la sortie d'une porte séquentielle r, la valeur est celle de l'entrée de r au cycle précédent, ou nulle pour le cycle initial. Pour les autres portes g, la valeur est l'application de la fonction (tableau 5.2) ${}^c f_g$ ou ${}^e f_g$ exploitant toutes les valeurs des signaux d'entrée respectivement projetées dans $\mathbb{B}^n$ et $\mathbb{E}^n$. Pour le domaine symbolique, si w est une constante symbolique $\texttt{CST(x)}$, alors x doit correspondre à la valeur concrète de w au même cycle, ${}^c\alpha^t_w$.

$${}^e eval(w,t) := \begin{cases} {}^e\alpha^t_w & \text{si } w \in I \\ t = 0\,?\ {}^e\alpha^0_w : {}^e\alpha^{t-1}_{w^{in}_r} & \text{si } \exists r \in R \text{ s.t. } w = w^{out}_r \\ {}^e f_g(\dots, {}^e eval(w_i,t), \dots) & \text{sinon, } \exists g \text{ s.t. } w = w^{out}_g, \forall w_i \in W^{in}_g \end{cases} \tag{5.9}$$

Domaines	Entrées	Définition de ${}^{c}f_g$ et ${}^{e}f_g$
$\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$	$({}^{c}\alpha_{w1}, {}^{c}\alpha_{w2})$	${}^{c}f_g(\dots) = {}^{c}\alpha_{w1}$ AND/OR/XOR ${}^{c}\alpha_{w2}$
$\mathbb{E}^n \times \mathbb{E}^n \to \mathbb{E}^n$	$({}^{e}\alpha_{w1}, {}^{e}\alpha_{w2})$	${}^{e}f_g(\dots) =$ `OP_{AND/OR/XOR}` $({}^{e}\alpha_{w1}, {}^{e}\alpha_{w2})$

Tab. 5.2 : Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans les domaines $\mathbb{B}^n$ et $\mathbb{E}^n$.

5.1.4 Évaluation de stabilité

La fonction d'évaluation ${}^{s}eval(w, t)$ pour un cycle t est définie bit-à-bit dans l'équation 5.10, pour des raisons de lisibilité. Les conditions de terminaisons sont identiques à celles des fonctions d'évaluation concrètes et symboliques.

Lorsqu'un signal est une sortie de porte séquentielle, sa stabilité est l'équivalence bit-à-bit de ses expressions symboliques courantes et précédentes, ou $\vec{0}$ pour le cycle initial.

Dans la majorité des cas, la stabilité de la sortie d'une porte combinatoire ne dépend que de la stabilité de ses signaux d'entrée. Cependant, certaines valeurs symboliques d'entrée permettent, sous certaines conditions, d'assurer que la sortie sera stable ; la valeur symbolique est donc exploitée en addition de la valeur de stabilité pour l'évaluation de stabilité des portes combinatoires.

Exemple 8. Une porte AND de 1 bit ayant une entrée dont la valeur est stable et dont la valeur symbolique est `CST(0)` sera stable indépendamment de la stabilité et de la valeur de l'autre entrée [MM24]. Il en est de même pour une porte OR 1 bit dont une des entrées serait stable avec une valeur symbolique de `CST(1)`.

Ces valeurs symboliques stables, spécifiques à chaque porte, sont appelées *valeurs absorbantes*. Leur prise en charge implique que les fonctions dans le domaine de la stabilité ${}^{s}f_g$ prennent en entrée la valeur des signaux d'entrée projetées dans le domaine symbolique et dans le domaine de stabilité. Les cas des portes AND et OR sont illustrés dans le tableau 5.3 en proposant un comportement plus fin pour certaines valeurs spécifiques d'entrée symbolique.

$$
{}^{s}eval(w,t)[i] := \begin{cases} {}^{s}\alpha_w^t[i] & \text{si } w \in I \\ t = 0\,?\ {}^{s}\alpha_w^0[i] : {}^{e}\alpha_w^t[i] = {}^{e}\alpha_w^{t-1}[i] & \text{si } \exists r \in R \text{ t.q. } w = w_r^{out} \\ {}^{s}f_g(\dots, {}^{s}eval(w_j,t), \dots, & \text{sinon, } \exists g \text{ t.q. } w = w_g^{out}, \\ \quad \dots, {}^{e}eval(w_j,t), \dots)[i] & \forall w_j \in W_g^{in} \end{cases} \tag{5.10}
$$

f_g	Domaines	Entrées	Définition de ${}^s f_g[i]$
AND	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i])$ $\vee ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(0)))$ $\vee ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(0)))$
OR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i])$ $\vee ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(1)))$ $\vee ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(1)))$
XOR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	${}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i]$
NOT	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s\alpha_w[i]$

Tab. 5.3 : Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans le domaine $\mathbb{S}^n$.

5.1.5 Évaluation des ensembles de fuite

La fonction d'évaluation ${}^l eval(w, t)$ est aussi définie bit-à-bit pour des raisons de lisibilité, dans l'équation 5.11. Les ensembles de fuite sont des vecteurs d'ensembles d'expressions symboliques. Pour un signal d'entrée w de largeur $|w|$, la valeur initiale dans ce domaine est un vecteur de $|w|$ éléments dont chacun est le singleton valant l'expression du bit de même rang de l'expression symbolique du signal w. Si la valeur symbolique utilise le constructeur CST, un ensemble vide $\varnothing$ est utilisé à la place, ce cas est appelé *trivial* par la suite.

Pour les portes séquentielles, comme la valeur symbolique du cycle précédent et celle du cycle courant peuvent être observées en présence de glitches, l'ensemble de fuite pour un cycle t est défini bit-à-bit comme l'union des bits de même rangs de ces deux expressions. Pour le cycle initial, seule l'expression courante est considérée. Dans le cas où cette sortie est stable, toujours pour les portes séquentielles, les deux expressions sont nécessairement identiques.

Pour une porte combinatoire g, l'ensemble de fuite est calculé à partir des ensembles de fuite des entrées ainsi que sur les valeurs symboliques et de stabilité de la sortie de la porte g. Lorsque la sortie est instable, l'ensemble de fuite est calculé à partir de la fonction tableau 5.4 appliquée aux ensembles de fuite des signaux d'entrée. Cependant, lorsque la sortie est stable, l'ensemble de fuite résultant est le vecteur des singletons valant l'expression du bit de même rang de l'expression symbolique de la sortie.

Comme pour les entrées, pendant l'évaluation de la sortie d'une porte séquentielle ou combinatoire, si l'ensemble de fuite résultant n'est composé que d'expressions symboliques avec le constructeur CST, le calcul est trivial et remplacé par $\varnothing$.

Domaines	Entrées	Définition de ${}^{l}f_g$[i]
$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n$ $\times \mathbb{E}^n \to \mathbb{L}^n$	$({}^{l}\alpha_{w1}, {}^{l}\alpha_{w2},$ ${}^{s}\alpha_{w}, {}^{e}\alpha_{w})$	${}^{l}f_g(\dots)[i] = \begin{cases} {}^{l}\alpha_{w1}[i] \cup {}^{l}\alpha_{w2}[i] & \text{si } \neg {}^{s}\alpha_{w}[i] \\ \{{}^{e}\alpha_{w}[i]\} & \text{sinon} \end{cases}$

Tab. 5.4 : Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans le domaine $\mathbb{L}^n$.

$$ {}^{l}eval(w,t)[i] := \begin{cases} \{{}^{e}\alpha_w^t[i]\} & \text{si } w \in I \\ t = 0\,?\, {}^{l}\alpha_w^0[i] : \{{}^{e}\alpha_w^t[i], {}^{e}\alpha_w^{t-1}[i]\} & \text{si } \exists r \in R \text{ t.q. } w = w_r^{out} \\ {}^{l}f_g(\dots, {}^{l}eval(w_j,t), \dots, & \text{sinon, } \exists g \text{ t.q. } w = w_g^{out}, \\ \quad {}^{s}eval(w,t), {}^{e}eval(w,t))[i] & \forall w_j \in W_g^{in} \end{cases} \tag{5.11} $$

L'évaluation de la stabilité nécessite de connaître la valeur symbolique d'un signal tandis que l'évaluation de l'ensemble de fuite nécessite de connaître à la fois la valeur symbolique et la valeur de stabilité. Pour des raisons d'efficacité, le calcul de la valeur d'un signal doit donc être réalisé dans un ordre spécifique. Un tel ordre est proposé dans l'algorithme 2, qui réalise un parcours inductif pour un signal à un cycle donné et qui calcule la valeur dans tous les domaines à la fois. En conservant en mémoire, à travers les cycles, toutes les valeurs calculées pour les signaux internes, ce parcours calcule l'état pour le prochain cycle et les valeurs des signaux de sortie pour le cycle courant.

Algorithme 2 : Simulation en domaines mixtes d'un signal w à un cycle t.

Entrée : Un signal w, qui est soit une entrée primaire, soit la sortie d'une porte g
Sortie : La valeur α_w^t du signal w au cycle t
Fonction EVAL(w, t) **:**

si $w \in I$ **alors retourner** α_w^t
sinon si $g \in R$ **alors retourner** $(t == 0\,?\,\alpha_{w_g^{out}}^0 : \alpha_{w_g^{in}}^{t-1})$
sinon
/* Avec $w_{i1} \dots w_{in}$, les signaux dans W_g^{in} */
$\alpha_{w_{i1}}^t =$ EVAL(w_{i1}, t)
$\vdots$
$\alpha_{w_{in}}^t =$ EVAL(w_{in}, t)
$cv = {}^{c}f_g({}^{c}\alpha_{w_{i1}}^t, \dots, {}^{c}\alpha_{w_{in}}^t)$ ▷ Valeur concrète
$se = {}^{e}f_g({}^{e}\alpha_{w_{i1}}^t, \dots, {}^{e}\alpha_{w_{in}}^t)$ ▷ Expression symbolique
$st = {}^{s}f_g({}^{s}\alpha_{w_{i1}}^t, \dots, {}^{s}\alpha_{w_{in}}^t, {}^{e}\alpha_{w_{i1}}^t, \dots, {}^{e}\alpha_{w_{in}}^t)$ ▷ Stabilité
$ls = {}^{l}f_g({}^{l}\alpha_{w_{i1}}^t, \dots, {}^{l}\alpha_{w_{in}}^t, {}^{e}\alpha_w^t, {}^{s}\alpha_w^t)$ ▷ Ensemble de fuite
retourner $\langle cv, se, ls, st \rangle$

Cette section a proposé un cadre théorique pour la simulation en domaines mixtes de circuits pouvant être représentés comme des machines de Mealy. Cette simulation expose

les effets matériels se produisant pendant le fonctionnement de ces circuits et devant être captés pour une vérification dans le modèle `RR d-sw-probing` pour chacun des signaux internes.

La prochaine section est dédiée à l'établissement d'un cadre théorique de vérification de propriétés de sécurité sur des circuits simulés en domaines mixtes.

5.2 Orchestration de la vérification

Les vérifications sont réalisées indépendamment de la simulation du circuit. L'*orchestrateur de vérification* est responsable d'extraire les valeurs des signaux et de réaliser les vérifications cohérentes avec le modèle de fuite donné. L'interface entre cet orchestrateur et le vérificateur externe est une fonction $verif$, qui prend en entrée un ensemble d'expressions symboliques définies dans $\mathcal{E}$. Elle retourne vrai si le vérificateur a pu prouver que ces expressions ont une distribution jointe indépendante des valeurs sensibles et faux sinon. Le vérificateur n'a aucune information sur le modèle de fuite, il est donc de la responsabilité de l'orchestrateur de construire les expressions cohérentes avec le modèle, en utilisant les valeurs des signaux extraites de la simulation en domaines mixtes. La seule différence entre la vérification des propriétés de sécurité `NI` et `TPS` est la configuration du vérificateur externe pour des expressions manipulant des shares ou des secrets et masques. Pour les propriétés telles que `SNI` ou `PINI`, il est nécessaire de transmettre des informations supplémentaires (mais déjà disponibles dans la modélisation d'`aLEAKator`) telles que le nombre de signaux qui sont des sorties dans le n-uplet à vérifier et éventuellement les indices de sortie.

Cette section présente en premier lieu le cadre de construction des ensembles d'expressions à vérifier pour chaque modèle de fuite puis se concentre sur les opportunités de réduction du nombre d'ensembles d'expressions à vérifier à chaque cycle, en fonction du modèle de fuite.

5.2.1 Ensembles d'expressions à vérifier en cohérence avec un modèle de fuite

Le tableau 5.5 présente l'ensemble d'expressions construit pour un signal w à un cycle t pour chaque valeur de `g` et `t` dans le modèle `(g,t) 1-sw-probing`. La manière directe de vérifier un circuit est de construire puis de vérifier ces ensembles d'expressions pour chaque signal du circuit et pour chaque cycle simulé.

Modèle `(g,t) 1-sw-probing`	Ensemble d'expressions à vérifier
`(g=0, t=0)`	$\{{}^{e}\alpha_w^t\}$
`(g=0, t=1)`	$\{{}^{e}\alpha_w^t, {}^{e}\alpha_w^{t-1}\}$
`(g=1, t=0)`	$\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^t[i]$
`(g=1, t=1)`	$\{{}^{e}\alpha_w^{t-1}\} \cup (\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^t[i])$

Tab. 5.5 : Ensembles d'expressions construits pour un signal w à un cycle t pour chaque modèle de fuite.

Fuites en valeur. Pour le modèle `(0,0) 1-sw-probing`, le singleton composé de la valeur symbolique du signal w, ${}^{e}\alpha_w^t$, est envoyé au vérificateur.

Fuites en transition. Pour le modèle `(0,1) 1-sw-probing`, l'ensemble contenant les valeurs symboliques du signal w au cycle courant et au cycle précédent, $\{{}^{e}\alpha_w^t, {}^{e}\alpha_w^{t-1}\}$, est donné au vérificateur. Cet ensemble couvre toutes les combinaisons possibles des deux expressions et capte donc toutes les transitions possibles.

Fuites en valeur avec glitches. Pour le modèle `(1,0) 1-sw-probing`, l'ensemble des expressions à vérifier est l'aplatissement de l'ensemble de fuite du signal. Cet ensemble capture tous les glitches au niveau d'un signal.

Fuites en transition avec glitches. Pour le modèle `(1,1) 1-sw-probing`, l'ensemble d'expressions à envoyer au vérificateur est le singleton composé de la valeur symbolique du signal w au cycle précédent ainsi que l'aplatissement de l'ensemble de fuite du signal au cycle courant.

Il est intéressant de noter que l'ensemble d'expressions à vérifier pour le cas des transitions ou des valeurs avec glitches couvre celui d'une vérification en valeur. De même, celui construit pour une vérification en transition avec glitches couvre l'ensemble d'expressions construit pour tous les autres modèles présentés.

Ces ensembles d'expressions à construire sont les mêmes que la stabilité soit prise en compte ou non. Puisque la stabilité n'a d'impact qu'à la simulation, elle n'est pas mentionnée ici. Il convient néanmoins de distinguer les modèles de fuites réellement vérifiés selon qu'elle elle était active lors de la simulation ou non. Par exemple, en présence de stabilité à la simulation, le modèle `(1,1) 1-sw-probing` devient le modèle `RR 1-sw-probing`. De même, l'ordre de vérification ne change pas les expressions à construire pour chacun des signaux et n'est pas abordé ici.

La section suivante s'intéresse à la réduction du nombre d'ensembles d'expressions à construire pour effectuer la vérification d'un circuit, dans le but de passer plus aisément à l'échelle sur des circuits composés de nombreux signaux et lors de simulations sur de nombreux cycles.

Modèle (g,t) 1-sw-probing	Signaux à vérifier
(g=0,t=0)	$w \in W$ (Tous les signaux)
(g=0,t=1)	$w \in W$ (Tous les signaux)
(g=1,t=0)	$w \in W_r^{in}$ pour $r \in R$, $w \in O$, $w \in W^{split}$, $w \in W_g^{in}$ avec ${}^s\alpha^t_{w_g^{out}} \neq \vec{0}$, $w \in W_g^{in}$ avec g les multiplexeurs dont les sélecteurs sel respectent ${}^s\alpha^t_{sel} \neq \vec{0}$
(g=1,t=1)	$w \in W$ (Tous les signaux)

Tab. 5.6 : Signaux w à vérifier à un cycle t pour chaque modèle de fuite.

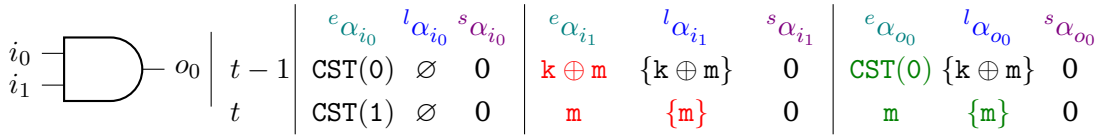


Fig. 5.5 : Exemple illustrant la nécessité de vérifier chaque signal dans les modèles (0,1) et (1,1) 1-sw-probing. L'entrée i_1 fuit k en transition au cycle t tandis que la sortie o_0 non.

5.2.2 Ensembles réduits d'expressions à vérifier

La section précédente suppose une stratégie gourmande impliquant de vérifier chaque signal à chaque cycle en envoyant au vérificateur d'expressions l'ensemble d'expressions associé au signal et adapté au modèle de fuite (tableau 5.5). En pratique, pour une quantité non négligeable des signaux, notamment ceux de contrôle, cet ensemble est vide ou composé uniquement d'expressions utilisant le constructeur CST. Ces ensembles sont considérés triviaux et ne doivent pas déclencher d'appel au vérificateur, au même titre que les ensembles d'expressions qui ne manipulent ni secrets ni shares.

Cette section est consacrée aux optimisations visant à exploiter le modèle de fuite pour ne pas vérifier des signaux dont les informations sont structurellement redondantes avec d'autres, dans un modèle de fuite donné. Le tableau 5.6 détaille, pour chaque modèle de fuite, les signaux à vérifier à chaque cycle. La stabilité est considérée active pour tous ces modèles de fuite, sauf mention contraire. Les optimisations proposées dans cette section s'appliquent aux vérifications avec les propriétés de sécurité NI et TPS au premier ordre. L'ajout d'optimisations pour l'ordre supérieur et pour les propriétés de sécurité supplémentaires reste une question ouverte.

Fuites en valeur et en transition sans glitch. L'ensemble d'expressions à construire dans les modèles (0,0) et (0,1) 1-sw-probing contient la valeur symbolique courante (et la précédente, pour les transitions) du signal. Il n'est pas possible de vérifier moins de signaux pour ces modèles car une expression générée en sortie d'une porte g peut présenter une fuite (par exemple avec l'expression $(k \oplus m) \oplus m$) tandis qu'une porte qui prend en entrée

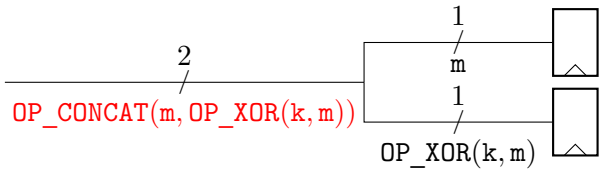


Fig. 5.6 : Exemple illustrant la nécessité de vérifier les signaux qui sont séparés et partiellement utilisés par une porte, avec $\texttt{k} = \texttt{SYMB}(K)$ et $\texttt{m} = \texttt{SYMB}(M)$. Le signal de 2-bits fuite k tandis qu'aucun des signaux d'un bit, qui sont les entrées de registres, ne fuit.

la sortie de g ne présente plus de fuite (par exemple $(m' \oplus ((k \oplus m) \oplus m))$. Ainsi, les sorties de toutes les portes doivent être vérifiées à chaque cycle car il n'y a aucune garantie que les portes logiques en aval présentent toujours les éventuelles fuites des portes logiques en amont.

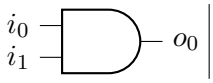
Exemple 9. La figure 5.5 illustre un tel cas avec la vérification dans le modèle `(0,1) 1-sw-probing` d'un circuit pour lequel un signal d'entrée (i_1) fuit en transition tandis que le signal de sortie de la porte (o_0) ne fuit pas.

Fuites en valeur avec glitches. Lorsque seuls les glitches sont considérés, il n'est pas nécessaire de vérifier tous les signaux. Pour le modèle `(1,0) 1-bit-probing` sans stabilité, les ensembles de fuite ne peuvent que croître en taille et ils deviennent maximaux en taille lorsqu'ils atteignent les sorties primaires du circuit ($w \in O$) ou les portes séquentielles ($w \in W_r^{in}$ for $r \in R$). Cependant, l'ajout de la stabilité invalide cette propriété, précisément parce que la stabilité permet de réduire la taille des ensembles de fuite associés aux sorties des portes logiques stables (cf. tableau 5.4). De plus, des portes qui permettent de réduire l'ensemble de fuite d'un signal d'entrée sans même que ce signal soit stable existent. C'est par exemple le cas des *multiplexeurs* dont le signal de sélection est stable car le signal sélectionné est reproduit sur la sortie et le signal non sélectionné est totalement mis de côté. C'est aussi le cas des signaux qui, sans traverser de porte, peuvent voir leur ensemble de fuite associé partiellement utilisé et donc ne plus présenter de fuite alors même que le signal initial en présentait. Ces signaux sont appelés signaux *split* ou *découpés* par la suite, et sont regroupés dans l'ensemble nommé W^{split}. Ainsi, les signaux d'entrée de toutes les portes qui présentent ce comportement doivent aussi être vérifiés pour prendre en compte la réduction éventuelle des ensembles de fuite lorsque la stabilité est considérée. En outre, signaux découpés doivent être vérifiés pour garantir que la propriété `sw` du modèle de fuite n'est pas relâchée lorsque tous les signaux ne sont pas vérifiés.

Exemple 10. La figure 5.6 illustre le besoin de vérifier les signaux dans W^{split}.

Ensemble d'expressions	$(\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t-1}[i]) \cup (\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t}[i])$.
Signaux à vérifier	$w \in W_r^{in}$ pour $r \in R$, $w \in O$, $w \in W^{split}$, $w \in W_g^{in}$ avec ${}^{s}\alpha_{w_g^{out}}^{t} \neq \vec{0}$ ou ${}^{s}\alpha_{w_g^{out}}^{t-1} \neq \vec{0}$, $w \in W_g^{in}$ avec g les multiplexeurs dont les sélecteurs sel respectent ${}^{s}\alpha_{sel}^{t} \neq \vec{0}$ ou ${}^{s}\alpha_{sel}^{t-1} \neq \vec{0}$

Tab. 5.7 : Signaux w à vérifier et ensembles d'expressions à construire pour une vérification au cycle t dans le modèle `(1,1) 1-sw-probing` lorsque la sur-approximation est considérée.



	${}^{e}\alpha_{i_0}$	${}^{l}\alpha_{i_0}$	${}^{s}\alpha_{i_0}$	${}^{e}\alpha_{i_1}$	${}^{l}\alpha_{i_1}$	${}^{s}\alpha_{i_1}$	${}^{e}\alpha_{o_0}$	${}^{l}\alpha_{o_0}$	${}^{s}\alpha_{o_0}$
$t-1$	`CST(0)`	$\varnothing$	1	`k` $\oplus$ `m`	{`k` $\oplus$ `m`}	0	`CST(0)`	$\varnothing$	1
t	`CST(1)`	$\varnothing$	0	`m`	{`m`}	0	`m`	{`m`}	0

Fig. 5.7 : Exemple illustrant la nécessité de vérifier, au cycle t, les entrées des portes qui étaient stables au cycle $t-1$. Dans le modèle `(1,1) 1-sw-probing`, l'entrée i_1 fuit `k` au cycle t tandis que la sortie o_0 ne fuit pas.

Fuites en transition avec glitches. Pour le modèle de fuite `(1,1) 1-sw-probing`, l'ensemble d'expressions à vérifier comprend l'expression symbolique du signal au cycle précédent (cf. tableau 5.5). Ainsi, comme pour l'exemple de la figure 5.5 mais dans le modèle `(1,1) 1-sw-probing`, tous les signaux doivent être vérifiés.

Le modèle `RR 1-sw-probing` est le plus précis ciblé par notre méthode, or structurellement, l'optimisation permettant de ne pas vérifier tous les signaux ne s'applique pas. Il est néanmoins possible de limiter le nombre de signaux à vérifier, même dans ce cas, à l'aide d'un compromis conservant la justesse de la vérification. En modifiant l'ensemble d'expressions envoyé au vérificateur (initialement dans le tableau 5.5) avec l'union des aplatissements des ensembles de fuite du signal w aux cycles t et $t-1$, l'expression symbolique du cycle précédent n'est plus nécessaire. L'expression devient donc $(\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t-1}[i]) \cup (\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t}[i])$. Cet ensemble d'expressions est une sur-approximation conservatrice des transitions avec glitches d'un signal, et permet à nouveau de ne pas vérifier tous les signaux, sous certaines conditions détaillées dans le tableau 5.7. Plus spécifiquement, les signaux d'entrée des portes stables qui devaient être vérifiés pour le cycle t sans les transitions doivent maintenant aussi être vérifiés si ces mêmes portes ont exploité la stabilité au cycle $t-1$.

Exemple 11. La figure 5.7 illustre un circuit pour lequel une fuite ne serait pas détectée si la sur-approximation était utilisée sans que la stabilité au cycle $t-1$ ne soit considérée.

Le cadre de vérification proposé dans cette section permet de considérer les variantes des modèles `sw-probing` avec la largeur des signaux définie dans la description matérielle des circuits. Cependant, certaines vérifications ne sont pertinentes que sur des signaux d'un bit, comme celles des accélérateurs matériels qui ont été conçus pour les modèles de

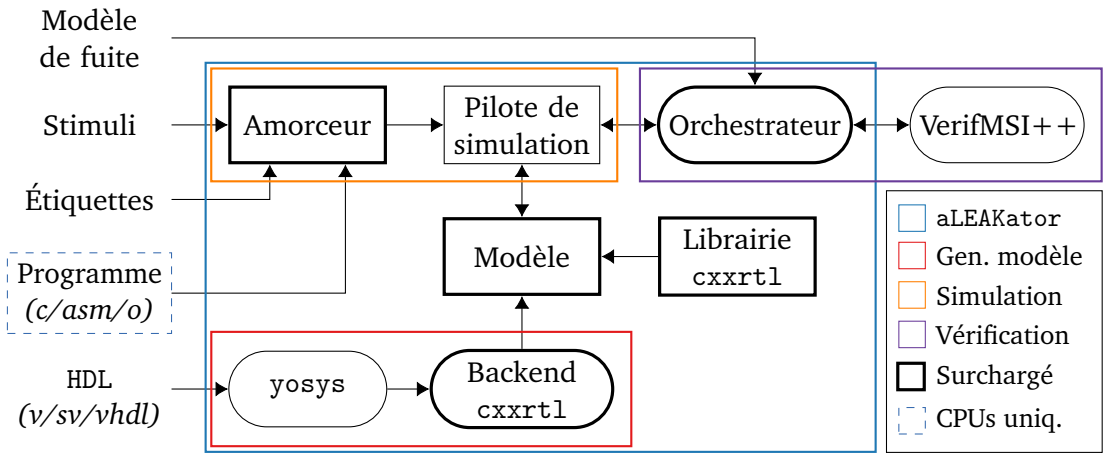


Fig. 5.8 : Implémentation d'aLEAKator.

fuite dans lesquels chaque bit fuit indépendamment. Toutes les réductions du nombre de signaux à vérifier présentées dans cette section sont toujours valides pour les modèles `(g,t) 1-bit-probing`. Le seul gain est que les signaux découpés ($w \in W^{split}$) n'ont plus besoin d'être vérifiés car ces modèles 1 bit ne considèrent pas la violation de la non-indépendance des masques comme une fuite. La vérification des signaux à la granularité bit impose d'envoyer chaque bit à vérifier au vérificateur externe, augmentant le coût en temps et en mémoire de la vérification.

5.3 Implémentation d'aLEAKator

Cette section détaille l'implémentation d'aLEAKator, la pile logicielle appliquant la méthode de simulation en domaines mixtes présentée dans la section 5.1. Dans un premier temps, le déroulement de la modélisation et de la vérification est illustré dans la figure 5.8. Les points saillants de la méthode, ainsi que les limitations, sont abordés ensuite.

5.3.1 Flux d'implémentation

Génération du modèle

Cette première étape est réalisée par `yosys` [Wol13] et par une version modifiée de son backend de simulation, `cxxrtl`. La simulation de ce modèle consiste en l'application d'une séquence d'entrées pour calculer les différents états et sorties ; telle que définie dans la définition 9. `Yosys` est appelé avec les sources `HDL`, qui peuvent être du `Verilog`, `SystemVerilog` ou du `VHDL` (via un outil externe nommé *GHDL*). En premier lieu, une

synthèse partielle du circuit est réalisée en appliquant toutes les étapes d'élaboration d'un circuit sauf l'association à une technologie. L'association à une technologie est ignorée dans l'objectif de ne pas remplacer les fonctionnalités de haut niveau du circuit par une composition équivalente de portes logiques 1-bit atomiques. Cela se justifie par deux raisons : la première est l'augmentation du nombre de portes (et de la complexité des expressions lors de leur traversée) par cette étape (dont un exemple est donné dans le tableau 5.8) et la seconde est que les opérateurs de haut niveau peuvent être implémentés de plusieurs manières, dépendant des choix de l'outil de synthèse. Il est donc plus judicieux de conserver des opérateurs de haut niveau et de les modéliser de manière conservatrice mais robuste aux différentes implémentations possibles. Par exemple, les opérateurs de multiplication sont conservés et modélisés de manière pessimiste dans la simulation en domaines mixtes (voir annexe B).

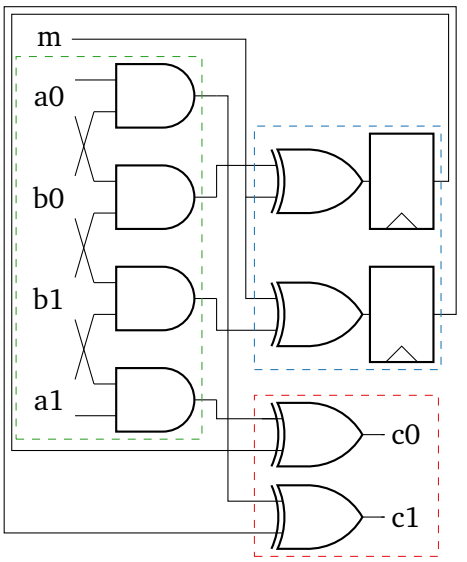
	♯ de signaux	♯ de bits	♯ de portes
Sans association	3	128	1
Avec association	766	34799	12460

Tab. 5.8 : Impact de l'association à une technologie pour un circuit composé uniquement d'un multiplicateur $\mathbb{B}^{32} \times \mathbb{B}^{32} \rightarrow \mathbb{B}^{64}$. L'association réalise notamment la transformation des portes complexes en portes basiques de la technologie (AND, MUX, XOR, ...).

Pour les microcontrôleurs, la synthèse partielle est réalisée deux fois, une première pour le processeur et une seconde avec tous les composants (processeur, mémoires, bus, périphériques, ...). Cela se rapproche de la manière dont sont fabriqués les microcontrôleurs, en garantissant que les interfaces du processeur restent inchangées (l'outil de synthèse ne peut donc pas optimiser les signaux à partir d'informations de part et d'autre de l'interface).

À l'issue de cette synthèse, la représentation interne de `yosys`, est une description dans un langage intermédiaire nommé `RTLIL`. Cette description est l'entrée du backend `cxxrtl`, qui opère comme une couche de traduction d'une description `RTLIL` du circuit en une description `C++` qui peut être simulée à l'aide d'une bibliothèque d'exécution nommée « *librairie `cxxrtl`* ». Cette bibliothèque, initialement développée pour réaliser des simulations concrètes, a été complètement réécrite pour intégrer les règles de propagation dans les différents domaines détaillées dans la section 5.1. Ce modèle `C++` doit être compilé, ce qui peut prendre jusqu'à 10 minutes pour le Cortex-M4. Cependant, une unique compilation du modèle est nécessaire, même lorsque l'implémentation logicielle doit changer, limitant l'impact de cette compilation.

Exemple 12. Le modèle `C++` généré pour le ET masqué avec le schéma `DOM` est illustré dans la figure 5.9. Un unique appel à la fonction `eval` ne permet pas de propager sur les sorties la valeur du ET logique, en raison des registres. Après le premier appel à `eval`, `c0 = a1 & b1` et `c1 = a0 & b0`, les sorties valides ne sont donc disponibles qu'après le second appel.



(a) Représentation en portes logiques agencées pour expliciter la machine de Mealy.

```
1  bool eval() {
2   p_a0b1 = and_uu<1>(p_a0, p_b1);
3   p_a1b0 = and_uu<1>(p_a1, p_b0);
4   p_a0b0 = and_uu<1>(p_a0, p_b0);
5   p_a1b1 = and_uu<1>(p_a1, p_b1);
6
7   p_a0b1m = xor_uu<1>(p_a0b1, p_m);
8   p_a1b0m = xor_uu<1>(p_a1b0, p_m);
9
10  p_c1 = xor_uu<1>(p_a0b0, p_reg1.curr);
11  p_c0 = xor_uu<1>(p_a1b1, p_reg0.curr);
12
13  if (posedge_p_clk) {
14   p_reg0.next = p_a0b1m;
15   p_reg1.next = p_a1b0m;
16  }
17  return true;
18 }
```

Listing 5.a : ET masqué avec le schéma DOM généré par `cxxrtl`.

Fig. 5.9 : Modèle pouvant être simulé du ET logique matériel, masqué avec le schéma `DOM` [GMK16] (description `Verilog` du listing 2.b de la partie I) et la machine de Mealy associée. La propriété suivante est respectée sur les sorties après propagation complète : `c0 ⊕ c1 = (a0 ⊕ a1) & (b0 ⊕ b1)`.

Bit-à-bit	Comparaison	Arithmétique	Décalage	Taille	Autre
bit_not	ucmp	add	shl	(r)trunc	is_zero
bit_and	scmp	sub	shr	(r)zext	is_neg
bit_or	equal	neg	sshr	sext	mem_write
bit_xor	not_equal	mul		blit	mem_read
				repeat	register

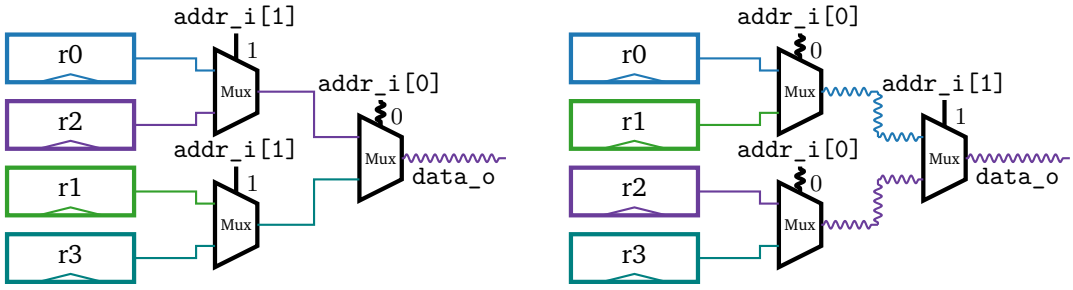
Tab. 5.9 : Fonctions permettant de simuler toutes les portes supportées par `aLEAKator`.

Les portes logiques supportées par `aLEAKator` sont simulées en utilisant les fonctions listées dans le tableau 5.9 afin de supporter l'ensemble des opérations qui peuvent être exprimées en `HDL` synthétisable. Il est possible d'utiliser `aLEAKator` pour simuler une netlist après association sur une technologie, au prix d'une forte augmentation du nombre de portes et de la définition d'une fonction calculant la fonctionnalité dans les différents domaines de chaque type de porte de la technologie.

Il est important de noter qu'une unique description `HDL` peut souvent mener à plusieurs circuits différents mais fonctionnellement équivalents. En effet, d'une part la syntaxe des langages de description de matériel permet souvent différents choix pour les programmes de synthèse et, d'autre part, les optimisations réalisées peuvent varier d'un programme de synthèse à un autre.

```
always @ (rd1_addr_i or reg0 or reg1 or reg2 or reg3)
 case (addr_i)
  2'b00 : data_o = r0;
  2'b01 : data_o = r1;
  2'b10 : data_o = r2;
  2'b11 : data_o = r3;
  default : data_o = 1'bx;
 endcase
```

Listing (5.b) Exemple de description `Verilog` décrivant un circuit qui réalise la lecture d'un registre parmi quatre.



(a) Premier exemple pouvant être généré. **(b)** Second exemple pouvant être généré.

Fig. 5.10 : Différents circuits pouvant être générés lors de la synthèse de la description donnée dans le listing 5.b, réalisant la lecture d'un registre parmi quatre. À titre d'exemple, le registre r2 est lu et l'adresse est bien `2'b10` dans les deux cas.

Les verdicts de vérification issus des outils qui prennent en entrée des descriptions `HDL`, `aLEAKator` compris, ne sont donc valables que pour l'interprétation de la description `HDL` réalisée par l'outil de synthèse et non pour la description `HDL` elle-même. Ainsi, même pour une description `HDL` identique à celle utilisée pour produire un circuit, des décalages entre les fuites observées et les verdicts de vérification peuvent éventuellement se produire. L'utilisation de la *netlist* exacte avant association à une technologie, générée au cours de la synthèse, élimine ce décalage mais ces netlists ne sont généralement pas accessibles pour des circuits « sur étagère ».

Exemple 13. La figure 5.10 illustre un exemple de description `HDL` et de deux circuits fonctionnellement équivalents mais dont le comportement en termes de fuite n'est pas le même en raison de l'ordre de traversée de multiplexeurs. Soient les registres r0 et r1 initialisés avec les deux shares qui composent un secret, le registre lu r2 et le bit `addr_i[1]` stable. Pour le circuit décrit dans la figure 5.10a, les ensembles de fuite des signaux d'entrée du multiplexeur piloté par `addr_i[1]` sont respectivement $\{r2\}$ et $\{r3\}$, ainsi même avec ce multiplexeur instable, l'ensemble de fuite en sortie présente pas de fuite. À l'inverse, dans la figure 5.10b, les ensembles de fuite en entrée du multiplexeur piloté par `addr_i[1]` sont respectivement $\{r0, r1\}$ et $\{r2\}$, la stabilité de ce multiplexeur n'empêchera donc pas ce glitch , qui se produit en aval.

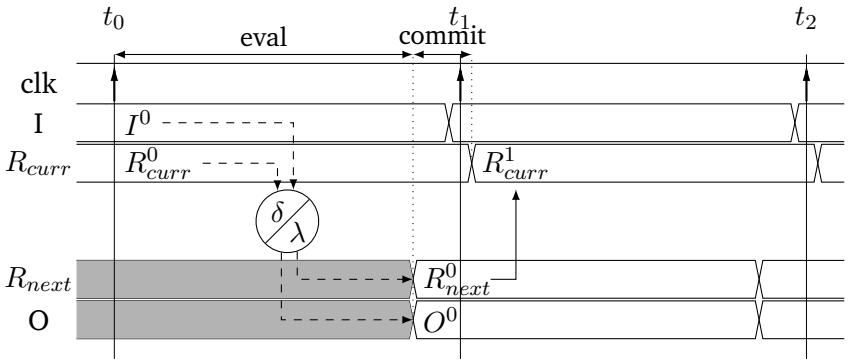


Fig. 5.11 : Temporalité des appels aux fonctions `eval` et `commit` de `cxxrtl`.

Simulation du modèle

Cette étape de simulation est réalisée en fournissant au modèle à simuler des séquences de stimuli d'entrée. Pour les microcontrôleurs, un programme doit être chargé à l'initialisation du modèle, comme illustré dans la figure 5.8. Chaque processeur disponible pour l'utilisation dans `aLEAKator` est fourni avec un *bootloader*, un programme minimal permettant d'initialiser le processeur pour ensuite lancer l'exécution d'un programme arbitraire. Le programme à simuler peut être un programme `C`, `Assembleur`, ou même un fichier objet issu d'une compilation. Après compilation avec le bootloader, les sections du binaire sont extraites pour générer un amorceur de mémoire qui les chargera aux bonnes adresses à l'initialisation de la simulation. Pour les microcontrôleurs comme pour les accélérateurs cryptographiques, l'utilisateur d'`aLEAKator` doit aussi fournir un étiquetage pour les valeurs données en entrée, en mémoire ou en registres. À cette fin, des fonctions d'aide sont fournies pour être surchargées par l'utilisateur. Elles sont appelées lorsque le matériel a terminé son initialisation, à chaque cycle ou lorsque le programme est terminé. Les programmes sont cross-compilés [2] pour l'architecture ciblée en utilisant l'infrastructure de compilation `llvm` [LA04], assurant une intégration aisée lors de l'ajout de nouvelles architectures à `aLEAKator`.

La simulation des modèles `cxxrtl` est réalisée en deux étapes qui sont illustrées dans la figure 5.11. Premièrement, la fonction `eval` propage les valeurs des entrées actuelles et des portes séquentielles du cycle précédent jusqu'aux sorties et portes séquentielles. Puis la fonction `commit` charge les nouvelles valeurs des portes séquentielles, qui seront utilisées pour le prochain cycle.

À partir de ces éléments, le pilote de simulation propage, cycle après cycle, les séquences d'entrée à travers le modèle, dans tous les domaines, au sens de la définition 9.

2. Compilés pour une architecture cible qui peut être différente de l'architecture depuis laquelle la compilation est lancée.

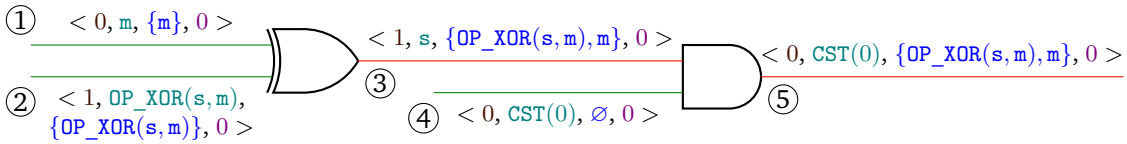


Fig. 5.12 : Identification de la source d'une fuite dans le cadre de la vérification en valeur avec glitches d'un circuit utilisant un masque `m` et un secret `s`. Les signaux présentant des fuites en valeur avec glitches sont indiqués en rouge et les autres en vert. Seul le signal ③ devrait être retourné pour simplifier l'identification des signaux importants pour corriger la fuite.

Vérification du modèle

Cette troisième étape est réalisée par l'orchestrateur de vérification, responsable d'extraire les valeurs dans les différents domaines des signaux de chaque cycle simulé du modèle. Ces valeurs sont conservées en cohérence avec le modèle de fuite, qui peut nécessiter de vérifier conjointement des valeurs issues de différents cycles. L'ensemble des expressions et ensembles d'expressions, réduit à l'aide des optimisations présentées, est construit pour le modèle de fuite et la propriété de sécurité donnés en entrée. Les étapes de simulation et de vérification sont répétées de manière entrelacée jusqu'à la fin du programme, des séquences d'entrée ou jusqu'à la première fuite selon la configuration d'`aLEAKator`. Si une fuite est rencontrée, les informations disponibles pour en comprendre l'origine sont données à l'utilisateur, notamment : le numéro de cycle de simulation, les expressions qui ont mené à cette fuite, l'emplacement des signaux participant à la fuite (le fichier et la ligne dans les sources `HDL` ainsi que le nom du signal).

`aLEAKator` peut optionnellement retrouver l'origine des fuites. Le besoin derrière cette option est que lorsqu'une fuite se produit dans un circuit comportant de nombreuses portes, il est probable qu'une fuite se propage dans une grande partie de l'arbre combinatoire issu du signal sur lequel elle apparaît. D'autre part, lorsque les optimisations réduisant le nombre de signaux à vérifier sont utilisées, les signaux sur lesquels `aLEAKator` relève des fuites ne sont pas nécessairement ceux qui sont à l'origine de la fuite. Pour ces deux raisons, `aLEAKator` peut optionnellement vérifier récursivement (en s'arrêtant aux registres et entrées primaires) les entrées des portes dont les sorties présentent des fuites dans le but de trouver les portes dont la sortie fuit mais pas les entrées. Ces portes sont indiquées comme à l'origine de la fuite et leur identification présente deux avantages : les informations associées à la fuite sont minimales (il n'existe pas d'ensemble de fuite ou d'expression plus simple pour expliquer cette fuite) et une fuite donnée n'est pas déclarée sur les signaux de l'arbre combinatoire aval du signal sur lequel apparaît cette première fuite.

Exemple 14. La figure 5.12 illustre une vérification concluant à la présence d'une fuite secrète en valeur avec glitches. Le circuit étant purement combinatoire et ne présentant aucun signal stable, seul le signal ⑤ doit être vérifié dans le cadre d'une vérification en

valeur avec glitches et avec les optimisations. D'autre part, si les optimisations n'étaient pas activées, les signaux ③ et ⑤ seraient identifiés comme présentant des fuites. Dans ces deux cas, l'identification de la source des fuites par `aLEAKator` permet d'indiquer au concepteur que le signal ③ est la source d'une fuite au lieu de n'indiquer que le signal ⑤ ou à la fois les signaux ③ et ⑤.

Génération et vérification d'expressions avec VerifMSI++

Pour la génération et la vérification externe des expressions, l'outil `VerifMSI++`[3], qui est une version améliorée de `VerifMSI` [MT23] est utilisé. Son interface `C++` en fait un candidat idéal pour l'intégration à `aLEAKator`. `VerifMSI++` permet la représentation et la simplification d'expressions symboliques complexes et propose différents algorithmes pour vérifier des propriétés comme `NI` et `TPS` sur ces expressions.

`VerifMSI++` permet la simplification des expressions symboliques à l'aide d'heuristiques mais n'offre aucune garantie d'optimalité des expressions. Les simplifications appliquées ne peuvent que réduire la taille d'une expression. Ces simplifications peuvent être la propagation des éléments construits avec `CST`, des simplifications booléennes, des factorisations, le rassemblement de n opérations identiques en une opération sur n éléments et l'élimination des redondances. `VerifMSI++` peut vérifier si deux expressions sont équivalentes, ce qui est nécessaire pour calculer la stabilité des registres chargés avec des valeurs symboliques.

Cet outil implémente la méthode par substitution pour prouver l'indépendance statistique des expressions face aux shares ou secrets. Les expressions sont modélisées sous la forme d'un graphe orienté dont les nœuds sont des opérateurs (parmi ceux du tableau 5.1) et les arêtes sont des opérandes. Les extrémités du graphe sont soit des constantes, soit des variables symboliques. La vérification d'une expression consiste en un remplacement itératif de sous-expressions par des expressions manipulant moins de variables symboliques, jusqu'à ce qu'aucun share ou secret n'apparaisse dans l'expression. L'algorithme 3 représente une version simplifiée de la vérification de la propriété de sécurité TPS sur une expression `nœudInitial`. Un nœud reliant, via un opérateur bijectif, un masque à une autre sous-expression qui ne fait pas intervenir ce masque peut être remplacé par celui-ci. Toutes les autres occurrences du masque dans l'expression globale doivent aussi être remplacées par la sous-expression masquée.

Exemple 15. Deux expressions et leurs remplacements démontrant que la première est indépendante des secrets mais pas la seconde sont illustrés dans la figure 5.13.

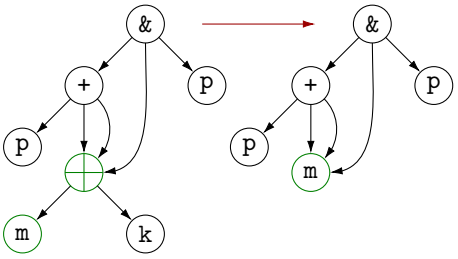
3. Disponible sur github : `https://github.com/quentin-meunier/verif_msi_pp`

Algorithme 3 : Vérification de la propriété TPS, adapté de [MT23].

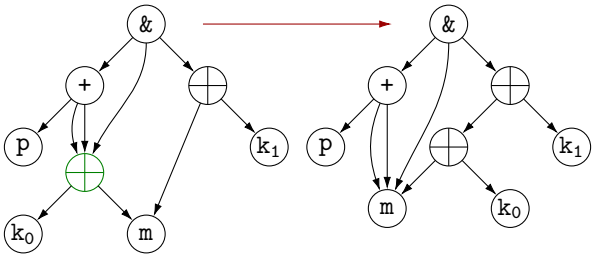
Entrée : nœudInitial : la racine de l'expression à vérifier
Sortie : verdict : le verdict d'indépendance statistique de la distribution de nœudInitial par rapport aux secrets

Fonction THRESHOLDPROBINGSECURITY(nœudInitial) :

```
nœud = simplifier(nœudInitial)
masquesPris = {}
tant que Vrai faire
    si var_secrètes_occurences(nœud) == 0 alors
        retourner Vrai // Plus aucun secret
    si est_masqué(nœud) alors
        retourner Vrai // Le nœud courant est masqué
    masque, nœudÀRemplacer = selectionner_masque(nœud, masquesPris)
    si masque == NULL alors
        retourner Faux // Il reste au moins un secret mais aucun masque
    masquesPris = masquesPris ∪ {masque}
    nœud = remplacer_graph(nœudInitial, masque, nœudÀRemplacer)
    nœud = simplifier(nœudInitial)
```



(a) Plus aucun secret, la propriété TPS est validée.



(b) Impossible de réutiliser `m` et des secrets sont encore présents dans l'expression, la propriété TPS n'est pas validée.

Fig. 5.13 : Vérification d'expressions via des substitutions avec `VerifMSI++`. Les expressions manipulent des variables symboliques de mêmes tailles : un masque `m`, une valeur publique `p` et des secrets `k`, k_0 et k_1

Il existe des cas pour lesquels `VerifMSI++` ne peut pas conclure. Ceux-ci sont indiqués comme des fuites par `aLEAKator`, pouvant causer des faux-positifs mais offrant l'avantage de ne jamais produire de faux-négatifs. Pour un nombre limité de variables symboliques dans l'expression, `VerifMSI++` permet l'énumération (illustrée dans la figure 3.3 de la partie I), garantissant dans ce cas l'absence de faux-négatifs.

`VerifMSI++` permet de décrire des circuits matériels pour vérifier différentes propriétés telles que NI, SNI, PINI, o-PINI, etc. La description de ces circuits ne permet cependant que de vérifier des circuits *stateless* (en opposition à *stateful*), c'est-à-dire dont les valeurs des entrées ne peuvent pas évoluer au cours du temps. Par conséquent, chaque signal est associé à une expression symbolique statique et ne nécessite donc pas de simulation.

5.3.2 Modifications apportées à cxxrtl

De nombreuses modifications de cxxrtl ont été nécessaires, dans le backend qui génère les modèles comme dans la bibliothèque d'exécution, qui implémente désormais les règles de propagation par domaine.

Adaptation des modèles. Pour réaliser la vérification de chaque sortie de porte, il est nécessaire de configurer cxxrtl pour exposer chaque signal de la netlist. Pour ce faire, chaque traversée de porte dans le modèle généré doit se traduire en une affectation à un signal nommé dans le modèle C++. Les lignes en HDL telles que `assign a = b & c & d;` sont donc traitées en créant des signaux aux noms uniques `__1__ = and_uu<1>(b, c);` puis en réalisant l'affectation `a = and_uu<1>(__1__, d);`. Ainsi, toutes les portes traversées sont bien matérialisées comme des signaux individuels, accessibles par l'orchestrateur de vérification. De même, les portes implicites comme les multiplexeurs, normalement traduites en C++ comme `assign a = b ? c : d` sont explicitées par une nouvelle porte `a = mux<1>(b, c, d);`.

Surcharge des opérations mémoires. Les lectures et écritures en mémoire sont surchargées de manière à, lorsque configuré par l'utilisateur, exécuter une fonction réalisant la traduction de l'adresse mémoire accédée à l'aide d'une propriété symbolique donnée. Cela permet la simulation symbolique en présence d'accès mémoires à des indices symboliques, comme cela peut arriver dans les implémentations cryptographiques tabulées masquées qui utilisent une propriété de traduction d'un indice symbolique vers une lecture mémoire (comme dans l'AES masqué avec le schéma de Herbst par exemple, voir annexe A).

Restrictions des circuits supportés. Les circuits supportés par aLEAKator sont plus restreints que ceux supportés par cxxrtl. Ainsi, seuls les circuits pouvant être représentés comme des machines de Mealy et qui sont exempts de boucles combinatoires sont supportés. Il existe donc, pour tout circuit supporté, un ordonnancement de l'évaluation des portes qui permet de déterminer les valeurs des sorties et des portes séquentielles en une unique traversée. Par ailleurs, seuls les circuits sensibles à un unique front d'horloge sont considérés, ce qui permet de ne simuler que ce front. cxxrtl est partiellement limité vis-à-vis de ce qui est considéré comme une boucle combinatoire : l'outil considère que l'utilisation d'un bit i d'un signal pour calculer un autre bit $j \neq i$ de ce même signal est une boucle combinatoire même sans dépendance réelle entre ces bits. Une passe supplémentaire a été ajoutée au backend de cxxrtl pendant la génération des modèles afin de séparer les bits des signaux qui présentent ce problème. La considération conjointe pour la vérification des bits qui ont été séparés pour la génération du modèle est faite, afin de conserver la preuve dans une granularité de la largeur des signaux telle qu'indiquée dans le HDL. L'orchestrateur de vérification est informé de ces signaux et de la structure du circuit via un fichier généré par

`yosys`, nécessaire pour identifier des éléments spécifiques pendant la vérification tels que les entrées non-sélectionnées des multiplexeurs ou les bits séparés (figure 5.6).

Interface d'extraction des évaluations. L'orchestrateur de vérification, tel que décrit dans la sous-section 5.2.2, est en charge de l'extraction des évaluations dans les différents domaines pour tous les signaux. À cette fin, une interface permettant d'extraire ces informations du modèle a été développée ainsi qu'une solution de suppression des valeurs temporaires inutilisées pour améliorer les performances de la simulation.

Cache des verdicts. Pour améliorer les performances globales, un mécanisme de cache a été implémenté pour ne vérifier les expressions ou ensembles d'expressions identiques qu'une unique fois. Cette optimisation améliore grandement les performances d'`aLEAKator` et est discutée dans la sous-section 5.4.2.

5.4 Résultats expérimentaux

Cette section est dédiée à différentes expériences pratiques démontrant l'intérêt et l'efficacité d'`aLEAKator`.

La sous-section 5.4.1 présente les verdicts de vérification pour des accélérateurs cryptographiques de l'état de l'art qui ont déjà été vérifiés par d'autres méthodes. L'objectif est de montrer que des verdicts concordants peuvent être obtenus en un temps réduit. La sous-section 5.4.2 présente la vérification formelle d'implémentations logicielles masquées sur cinq processeurs différents, démontrant la versatilité et la robustesse de la méthode. Finalement, les verdicts d'`aLEAKator` sont validés expérimentalement pour deux programmes sur deux processeurs à l'aide de mesures réelles de consommation. Ces résultats montrent que lorsqu'`aLEAKator` ne relève aucune fuite dans le modèle `RR 1-sw-probing`, aucune fuite n'est observée en pratique.

L'environnement expérimental pour toutes les expériences de cette section est une machine du cluster Dalek [CAB25] embarquant un processeur AMD Ryzen 9 7945HX avec 92Go de RAM et 512Go de mémoire swap sur SSD.

5.4.1 Accélérateurs cryptographiques masqués

L'analyse des accélérateurs matériels est réalisée sur trois cibles matérielles, desquelles huit expériences sont dérivées. Les résultats sont regroupés dans le tableau 5.10 et comparés avec les outils `Coco` et `Prover` (présentés dans la sous-section 3.2.2) et `Prolead` (présenté dans la sous-section 3.2.1). Les accélérateurs considérés ont été créés pour être sécurisés

Accélérateur	Vérif/Cycles Totaux	aLEAKator		Coco		Prover	Prolead
		Verdict	RAM (Go)	Verdict	RAM (Go)	Verdict	Verdict
SBox Present-TI NU	2/2	✗ <1s	0.01	✗ <1s	0.03	✗	✗
SBox Present-TI U	2/2	✓ <1s	0.01	✗* <1s	0.03	✓	✓
Present-TI NU	3/544	✗ <1s	0.01	-	-	-	✗
Present-TI U	3/544	✓ <1s	0.01	-	-	-	✓
AES-DOM	21/216	✓ 30s	12.3	✓ 26m30	3	-	-
Prince-TI	3/25	✗ 4.5s	2.2	✗ 4.6s	0.4	-	-
Prince-TI-Coco	3/25	✓ 5s	2.7	✓ 29s	0.4	-	-
Prince-TI-Coco	5/25	✗* 16s	8.0	✓ 3m45	1.7	-	-

Tab. 5.10 : Résultats de vérification obtenus par aLEAKator, Coco, Prover et Prolead. Les résultats marqués ✗* correspondent aux faux-positifs (le circuit est sécurisé mais le verdict de l'outil est « non-sécurisé »).

dans des modèles ne prenant pas en compte la largeur des supports ; les vérifications sont donc réalisées dans le modèle RR 1-bit-probing.

Present est un algorithme de chiffrement par bloc pensé pour des implémentations matérielles et composé de 31 rondes. Shahmirzadi *et al.* [SM21b] proposent Present-TI, une version de Present masquée avec le schéma TI. Deux variantes sont disponibles, une utilisant une SBox non-uniforme (NU) [End+17] et une seconde utilisant une SBox uniforme (U) [Pos+11]. Les auteurs de Prover ont montré que Coco produit des faux-positifs sur les deux versions de SBox tandis que Prover arrive à vérifier que seule la SBox uniforme est sécurisée. Les auteurs de Prolead ont vérifié les deux SBox ainsi que les implémentations complètes de Present, avec des verdicts sécurisés pour les versions uniformes et non-sécurisés pour les autres. aLEAKator a recours à l'énumération pour confirmer que ce verdict est un faux-positif et que seule la SBox uniforme est sécurisée. La version complète de l'algorithme est aussi vérifiée et donne des résultats concordants. À partir du troisième cycle, l'énumération est utilisée et permet de vérifier le quatrième cycle en 26 minutes. En raison des faux-positifs sur les SBox, les résultats de Coco pour l'algorithme complet ne sont pas donnés.

Prover ne permet pas de vérifier les circuits stateful (dont les entrées peuvent varier au cours du temps). Les résultats pour Present, AES-DOM et Prince ne sont donc pas donnés.

AES-DOM est une version masquée en VHDL avec le schéma DOM de l'AES-128 [RD01], proposée par Gross *et al.* [GMK16] avec un ordre de protection configurable. Une version nommée *non perfectly interleaved* avec une SBox pipelinée à huit étages a été vérifiée par Prolead dans le modèle (1,1) 1-probing (indisponible sur le dépôt git de la nouvelle version de l'outil, Prolead-V3). Cette version n'est pas utilisée dans le cadre des comparaisons avec aLEAKator car la configuration est différente. La version nommée *perfectly interleaved* avec une SBox pipelinée à 5 étages a été vérifiée par Coco jusqu'au 21ème

cycle. aLEAKator reproduit ces résultats dans une configuration identique (tableau 5.10). L'accélérateur utilise 38 bits de masque par cycle et les shares de clé et d'entrée à chiffrer sont concaténés octet-par-octet pendant les 16 premiers cycles. À partir du 21ème cycle, Coco échoue à conclure en un temps raisonnable (24h) tandis qu'aLEAKator peut vérifier jusqu'au 23ème cycle. Sur cette expérience, aLEAKator est 50 fois plus rapide que Coco, pour une consommation en mémoire 5,5 fois plus élevée.

Prince [Bor+12] est un algorithme de chiffrement par bloc plus petit que Present et ne nécessitant que 11 rondes. Božilov *et al.* [BKN22] proposent Prince-TI, une version masquée avec le schéma TI de l'algorithme Prince. Les auteurs de Coco ont montré qu'en raison d'un glitch sur un signal de contrôle, cette version n'est pas sécurisée. aLEAKator identifie le même défaut, sur le même multiplexeur que celui identifié par Coco. Une version corrigée a été proposée par les auteurs de Coco et vérifiée jusqu'au cycle 7. aLEAKator n'arrive plus à conclure à partir du cycle 5 et, comme l'énumération n'est pas possible en raison du nombre d'entrées symboliques, nous interprétons ce résultat comme un faux-positif. Les résultats corrects sont néanmoin obtenus jusqu'à 6 fois plus vite par aLEAKator. Prolead a été utilisé pour vérifier Prince Null-Fresh [SM21a], qui est donc un accélérateur matériel différent, avec lequel aucune comparaison n'est réalisée.

En résumé, aLEAKator est capable de reproduire des résultats connus de vérification, malgré d'éventuels faux-positifs. Pour les circuits non-triviaux, aLEAKator est jusqu'à 50 fois plus rapide que Coco, au coût d'une consommation mémoire plus élevée.

5.4.2 Implémentations logicielles masquées exécutées sur des processeurs

Cette section détaille les résultats de vérification d'implémentations logicielles sur cinq processeurs.

Configuration des processeurs

Cette sous-section détaille les différents processeurs ainsi que les configurations vérifiées par la suite.

Ibex [Con18] [4] est un cœur implémentant le jeu d'instructions 32 bits RISC-V, avec les extensions IEMCB. Initialement développé dans le cadre de la plateforme PULP [PUL26],

4. Disponible sur github : https://github.com/lowRISC/ibex.

le cœur est désormais maintenu et développé par *lowRISC*. Ce cœur est hautement configurable et permet de choisir entre un pipeline de deux ou trois étages (avec un étage de write-back). Il permet aussi de choisir parmi différents multiplicateurs et optionnellement d'utiliser un prédicteur de branchement. Dans ces travaux, la version avec trois étages est utilisée. En outre, l'*ibex simple system* fourni avec le cœur est utilisé. Il comprend un bus et un sous-système de mémoire.

Coco-Ibex [Gig+21] [5] est un fork (une divergence) de l'Ibex sécurisé par les auteurs de `Coco`. Additionnellement à la configuration précédemment détaillée, le cœur embarque un banc de registres sécurisé contre les glitches en écriture et en lecture, une RAM sécurisée et des mécanismes de masquage des entrées des sous-systèmes (appelé *gating* par la suite) tels que le multiplicateur, l'additionneur, l'opérateur de décalage ou encore les registres de contrôle et de statut. Chaque fonctionnalité de sécurité est indépendante et peut être activée/désactivée lors de la synthèse du cœur. Le bus utilisé pour accéder à la ROM non-sécurisée contenant les instructions est l'*ibex simple system*. Les vérifications de ces travaux sont réalisées avec une configuration identique à celle des auteurs de `Coco` [Gig+21], dans la configuration avec deux étages (sans le mécanisme de write-back). La mémoire sécurisée du Coco-Ibex est implémentée avec des flip-flops au lieu de mémoires traditionnelles. Cela permet une vérification plus précise des éléments mémorisants que pour les autres cœurs qui utilisent des blocs mémoires en boîte noire. Certaines expériences, comme l'`AES-Herbst`, ne sont pas données pour le Coco-Ibex car elles nécessitent plus que 512 octets de RAM et que l'augmentation à 4096 octets de la RAM sécurisée doublerait le nombre de portes nécessaires pour synthétiser le cœur. Pour réaliser nos expériences, plusieurs correctifs de bugs matériels qui faisaient dysfonctionner des instructions usuelles comme le `c.jal` ont été ajoutés au cœur. Comme le Coco-Ibex implémente des contre-mesures, certaines sources de fuite sont supprimées au niveau matériel. En suivant les recommandations des auteurs concernant le placement des données sensibles en mémoire, l'implémentation de programmes sécurisés est plus aisée que pour les autres cœurs. Il demeure toutefois nécessaire de vérifier les implémentations logicielles pour ce cœur car toutes les sources de fuites ne sont pas corrigées par le matériel.

CV32E40P [Gau+17] [6] est un cœur sous licence libre implémentant le jeu d'instructions 32 bits RISC-V (extension IM[F|Zfinx]C) avec un pipeline de 4 étages. Les deux objectifs du projet sont la performance et l'efficacité énergétique. Initialement maintenu par la plateforme PULP [PUL26], le cœur est maintenant maintenu et développé par le groupe *OpenHW*. Les expériences sur ce cœur ont été réalisées avec le même sous-système de mémoire et de bus que pour l'Ibex.

5. Disponible sur github : `https://github.com/isec-tugraz/coco-ibex`.
6. Disponible sur github : `https://github.com/openhwgroup/cv32e40p`.

Cortex-M3 et **Cortex-M4** sont deux processeurs adaptés pour être embarqués dans des microcontrôleurs. Les deux processeurs implémentent le jeu d'instructions `ARMv7-M` et utilisent des pipelines de trois étages. Leurs implémentations `Verilog` implémentent des mécanismes de *gating* visant à réduire la consommation. Les descriptions `Verilog` sont rendues disponibles via le programme Arm Academic Access (AAA) [Lim26a]. Additionnellement, un bus implémentant le protocole AHB-lite est utilisé pour nos vérifications. Les fonctionnalités de débogage, de journalisation des opérations et la sonde JTAG sont désactivées lors de la synthèse car elles ne sont pas utiles pour les expériences de ces travaux et sont supposées désactivées sur des circuits en production.

Programmes vérifiés

Pour chaque programme, deux variantes sont considérées, une variante non-sécurisée et une variante sécurisée. Les variantes sécurisées sont obtenues par une méthode itérative à partir d'une variante non-sécurisée exploitant le verdict et les informations fournies par `aLEAKator` jusqu'à l'obtention d'un verdict sécurisé dans le modèle de fuite ciblé. Les programmes ARM et RISC-V ne sont pas identiques car ces langages d'assemblage sont incompatibles.

`DOM-AND` [GMK16] est une implémentation du ET logique sur 32 bits, masquée avec le schéma `DOM`. La variante non-sécurisée (`DOM-AND-nsec`) est une version assembleur prouvée au niveau du jeu d'instructions comme étant sécurisée en considérant les transitions[7]. Une variante `DOM-AND-sec-rr` a été créée pour être sécurisée dans le modèle `RR 1-sw-probing`.

`Secmult` [RP10] est une implémentation masquée de la multiplication dans le corps fini (ou corps de Galois) GF(2^8). Pour les cœurs ARM, la variante non-sécurisée (`Secmult-nsec`) est obtenue via la compilation avec optimisations (`-O3`) d'une implémentation `C` masquée. Pour les processeurs RISC-V, la variante non-sécurisée (`Secmult-nsec`) est une version `Assembleur` manuellement créée et sécurisée au niveau du jeu d'instructions en valeur sans considération des transitions. Finalement, la variante sécurisée (`Secmult-sec-rr`) a été créée pour être sécurisée dans le modèle `RR 1-sw-probing`.

`AES-Herbst` [HOM06] est une implémentation tabulée de l'AES masquée avec le schéma de `Herbst`. Cette version utilise six masques d'un octet chacun et comprend l'étape de « key schedule » qui réalise la création des clés de rondes. Une explication de l'algorithme est donnée dans l'annexe A. La variante non-sécurisée est une version compilée avec optimisations

7. L'implémentation ARM de `DOM-AND-nsec` était donnée en exemple dans la figure 2.11 de la partie I pour montrer l'importance de considérer la microarchitecture dans les modèles de fuite dans la figure 2.12 de la partie I.

(-O3) d'une implémentation C de l'algorithme. Comme la création d'une version sécurisée dans le modèle RR 1-sw-probing serait une tâche chronophage et fastidieuse, la variante sécurisée (AES-Herbst-sec-v) est produite pour le modèle (0,0) 1-sw-probing. Cette variante est issue de la compilation d'une version protégée et renforcée en C. Cette version sécurisée a tout de même été vérifiée avec aLEAKator dans le modèle RR 1-sw-probing et présente comme attendu de nombreuses fuites. Cela permet de montrer qu'aLEAKator peut être utilisé pour des vérifications d'algorithmes complexes dans les modèles de fuite les plus avancés. En pratique, la vérification d'expressions sécurisées est moins coûteuse en temps et en mémoire que la vérification d'expressions qui ne le sont pas. Il est donc généralement plus judicieux d'arrêter la vérification à la première fuite identifiée.

Cette implémentation est tabulée en exploitant des indices symboliques. Pour en réaliser la vérification, la fonctionnalité de surcharge du système de mémoire (présentée dans la sous-section 5.3.2) a été utilisée pour chacun des processeurs, permettant de convertir des accès mémoires symboliques en valeurs symboliques lues sur le bus. Plus particulièrement, le schéma de masquage Herbst nécessite de calculer la table SBox', initialisée comme SBox'[i $\oplus$ m] = SBox[i] $\oplus$ m' pour toutes les valeurs de i $\in [0, 255]$ et avec m et m', deux masques du schéma. Ainsi, les lectures dans la SBox' sont converties, pour leur partie symbolique, en une lecture masquée de SBox. Il est important de noter que seule la partie correspondant à l'indice est symbolique. Il est donc possible de déterminer, en exploitant uniquement les informations concrètes de l'adresse, qu'il s'agit d'une lecture dans SBox', ce qui garantit la complétude d'une telle vérification.

Résultats de co-vérification matérielles/logicielles

Les résultats de vérification sont présentés dans le tableau 5.11. La propriété de sécurité utilisée pour ces programmes est 1-TPS. Pour chaque processeur, les variantes des trois programmes, à l'exception de l'AES-Herbst sur le Coco-Ibex, sont vérifiées.

Comme l'ordonnanceur de vérification élimine les vérifications triviales et utilise des caches, le nombre d'expressions envoyées au vérificateur externe est bien plus faible que le nombre de signaux des processeurs multiplié par le nombre de cycles vérifiés. De plus, afin d'illustrer l'effet de la sur-approximation proposée dans la sous-section 5.2.2 pour réduire le nombre de signaux à vérifier, deux colonnes sont données pour le nombre d'expressions. La première, *Expr. à vérifier*, représente le nombre d'ensembles d'expressions qui devraient être envoyées au vérificateur en l'absence de la sur-approximation (mais toujours en exploitant les caches et en supprimant les vérifications triviales). La seconde, *Expr. vérifiées*, est le nombre d'ensembles d'expressions envoyés en pratique. La différence entre ces deux colonnes est donc le nombre d'ensembles d'expressions uniques, pour lesquels la sur-approximation a

	SW	Var.	F	Modèle de fuite	Cycles	# expr. à vérifier	# expr. vérifiées	Cycles fuite	Temps	RAM (Go)
Cortex-M3	DOM-AND	nsec	5.14a	Valeur	48	630	630	0	1.8s	0.6
				RR	48	1 092	733	33	3.8s	1.0
		sec-rr	5.14b	RR	67	1 095	779	0	4.5s	1.0
	Secmult	nsec	5.16a	Valeur	209	2 916	2 916	53	5.1s	0.5
				RR	209	4 591	3 469	185	13.5s	1.4
		sec-rr	5.16b	RR	660	4 494	3 186	0	34.0s	1.7
	AES-Herbst	nsec		Valeur	6 924	74 707	74 707	3 050	3m46	37.1
		sec-v		Valeur	10 113	90 053	90 053	0	5m04	43.7
				RR	10 113	192 812	148 409	8 329	18m53	208.0
Cortex-M4	DOM-AND	nsec	5.15a	Valeur	48	649	649	0	1.9s	0.6
				RR	48	1 112	762	33	4.3s	1.1
		sec-rr	5.15b	RR	67	1 112	799	0	5.0s	1.0
	Secmult	nsec	5.17a	Valeur	203	2 998	2 998	36	5.7s	0.5
				RR	203	4 669	3 419	179	14s	1.4
		sec-rr	5.17b	RR	660	4 680	3 255	0	39s	1.8
	AES-Herbst	nsec		Valeur	6 924	77 227	77 227	3 050	4m13	38.0
		sec-v		Valeur	10 113	93 279	93 279	0	5m48	45.3
				RR	10 113	194 470	147 766	8 329	20m14	211.5
Ibex	DOM-AND	nsec		Valeur	37	705	705	0	5.1s	3.4
				RR	37	881	350	18	2.7s	1.0
		sec-rr		RR	58	833	355	0	2.1s	0.7
	Secmult	nsec		Valeur	638	12 703	12 703	0	11.4s	2.8
				RR	638	17 609	4 527	236	21.3s	4.5
		sec-rr		RR	646	17 270	4 427	0	20.5s	4.3
	AES-Herbst	nsec		Valeur	6 294	159 261	159 261	2 963	14m37	203.1
		sec-v		Valeur	8 869	185 508	185 508	0	21m41	226.8
				RR	8 869	230 463	90 050	6 651	22m08	301.0
CV32E40P	DOM-AND	nsec		Valeur	37	785	785	0	10.5s	9.9
				RR	37	860	299	17	5.8s	1.1
		sec-rr		RR	60	822	231	0	7.5s	0.8
	Secmult	nsec		Valeur	639	13 824	13 824	0	15.8s	3.4
				RR	639	12 461	2 886	234	1m14	5.0
		sec-rr		RR	651	12 228	2 783	0	1m15	4.8
	AES-Herbst	nsec		Valeur	6 294	176 150	176 150	2 762	21m38	267.6
		sec-v		Valeur	8 869	199 405	199 405	0	43m44	423.5
				RR	8 869	207 301	86 259	6 705	39m01	333.7
Coco-Ibex	DOM-AND	nsec		Valeur	51	537	537	0	1.5s	0.5
				RR	51	1 223	651	9	6.8s	3.5
		sec-rr		RR	61	1 194	961	0	7.6s	3.8
	Secmult	nsec		Valeur	724	2 033	2 033	0	34.3s	8.3
				RR	724	4 639	3 583	10	1m30	43.6
		sec-rr		RR	739	4 622	3 533	0	1m30	43.7

Tab. 5.11 : Résultats de vérification détaillés d'aLEAKator. La colonne « F » référence les figures de comparaison avec des mesures réelles, présentées dans la section courante (Comparaison avec des mesures réelles).

permis d'éviter une vérification dans le modèle `RR 1-sw-probing`. Pour les vérifications en valeur (dans le modèle `(0,0) 1-sw-probing`), ces colonnes sont toujours identiques car tous les signaux doivent être vérifiés.

La colonne *Cycles fuite* représente le nombre de cycles, parmi le nombre de cycles d'exécution (colonne *Cycles*), pour lesquels au moins un ensemble d'expressions n'a pas pu être prouvé comme étant sécurisé par le vérificateur.

Les consommations en mémoire et en temps sont aussi rapportées pour chaque vérification. Une forte variation est observable entre les différents processeurs, même pour des programmes similaires. Cette différence peut être expliquée par la présence de gating des entrées des opérateurs inutilisés par certains processeurs. C'est le cas, par exemple, des processeurs ARM Cortex-M3 et Cortex-M4 mais pas des processeurs comme l'Ibex et le CV32E40P. L'absence de gating des entrées des opérateurs impose de calculer, à tous les cycles, la multiplication, l'addition, le décalage, etc. des éléments en entrée de l'ALU, ce qui engendre la génération de nombreuses expressions complexes et non nécessaires au calcul. L'écart de consommation de mémoire RAM est aussi expliqué par le nombre de portes logiques des circuits et le nombre de fuites déjà identifiées au cours d'une vérification. La consommation en mémoire pour les programmes complexes tels que l'`AES-Herbst` est élevée, mais nous la considérons acceptable puisque de la mémoire swap est utilisée.

Les optimisations implémentées dans `aLEAKator` sont un point essentiel de la méthode. Au cours de la vérification de la variante `AES-Herbst-sec-v` sur le Cortex-M3, plus de 80 millions de vérifications triviales sont évitées et le cache économise environ 2 millions de vérifications, réduisant de manière significative l'impact d'une telle vérification. La sur-approximation permet de réduire d'un facteur moyen de 2,2 et pouvant aller jusqu'à 4,4 (pour le `Secmult` sur le CV32E40P) le nombre d'ensembles d'expressions uniques à vérifier.

`aLEAKator` permet donc la vérification d'algorithmes nécessitant des milliers de cycles de calcul tels que l'`AES-Herbst` dans le modèle `RR 1-probing`, en un intervalle de 19 à 39 minutes, selon le processeur ciblé. Dans un modèle de fuite plus simple et sans le calcul des ensembles de fuites, la vérification peut être réalisée en seulement 5 minutes sur le Cortex-M3.

Les résultats notables de ces vérifications sont que les variantes `sec-rr` et `sec-v` des programmes qui ne présentent aucun cycle de fuite selon `aLEAKator` sont désormais garanties d'implémenter correctement le masquage, en considérant la microarchitecture et le modèle de fuite utilisé. Il s'agit, à notre connaissance, de la première vérification dans les modèles `(0,0) 1-(sw/bit)-probing` et `RR 1-(sw/bit)-probing` d'une version sécurisée dans le modèle `(0,0) 1-probing` de l'`AES` masqué avec le schéma de Herbst. `aLEAKator` permet de reproduire cette vérification pour quatre processeurs différents. Cela motive la création d'une implémentation sécurisée dans le modèle `RR 1-sw-probing`, afin d'évaluer la capacité d'`aLEAKator` à mener une telle vérification.

Comparaison avec des mesures réelles

Afin de valider les résultats obtenus avec `aLEAKator` et de confirmer la pertinence des modèles de fuite de l'état de l'art, cette section est dédiée à la comparaison des verdicts de vérification avec des mesures réelles. Les programmes comparés sont le `DOM-AND` et le `Secmult`, dans leurs variantes `nsec` et `sec-rr` et pour les processeurs ARM Cortex-M3 et Cortex-M4. Ces deux processeurs sont respectivement embarqués sur les microcontrôleurs interchangeables `STM32F1` et `STM32F3` de la `ChipWhisperer` CW1200 [OC14], dont une photographie est présentée dans la figure 2.6.

Comparaison des verdicts d'`aLEAKator` avec des mesures de t-test. Les fuites détectées par `aLEAKator` se produisent dans différents composants des processeurs, incluant l'unité de lecture/écriture mémoire (LSU, pour Load-Store Unit), le banc de registres, les ports et les opérateurs de l'ALU (ou UAL, pour Unité Arithmétique et Logique). Les figures 5.14, 5.15, 5.16 et 5.17 présentent l'évolution de la valeur du t-test pour chaque entrée secrète sur 500 000 traces et pour la durée complète du programme. Le test est réalisé sur l'écart de moyenne entre deux groupes ; ces groupes sont créés en fonction du poids de Hamming de l'entrée secrète considérée. Ces vérifications sont réalisées avec un échantillonnage de 4 mesures de consommation par cycle. La correspondance avec les vérifications présentées dans la section précédente est visible dans la colonne « F » du tableau 5.11. Nous utilisons comme seuil de rejet de H_0 (l'hypothèse d'indépendance statistique) la valeur de 4,5 du t-test comme détaillé dans la section 3.2 (sous-sous-section 3.2.1).

Ces figures illustrent que les variantes non-sécurisées présentent des valeurs de t-test dépassant le seuil tandis que toutes les variantes sécurisées restent en dessous de celui-ci. Ces résultats indiquent que lorsqu'`aLEAKator` ne trouve pas de fuite, aucune fuite n'est observable en pratique.

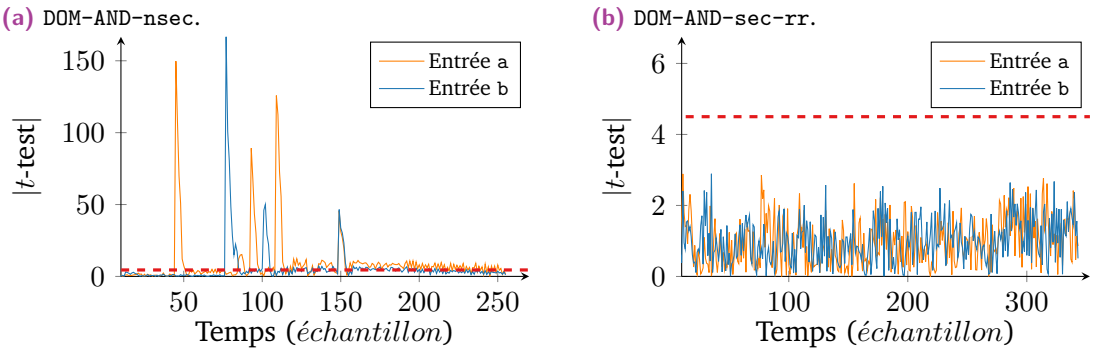


Fig. 5.14 : Valeurs de t-test : variantes `nsec` (a) et `sec-rr` (b) du `DOM-AND` sur le Cortex-M3.

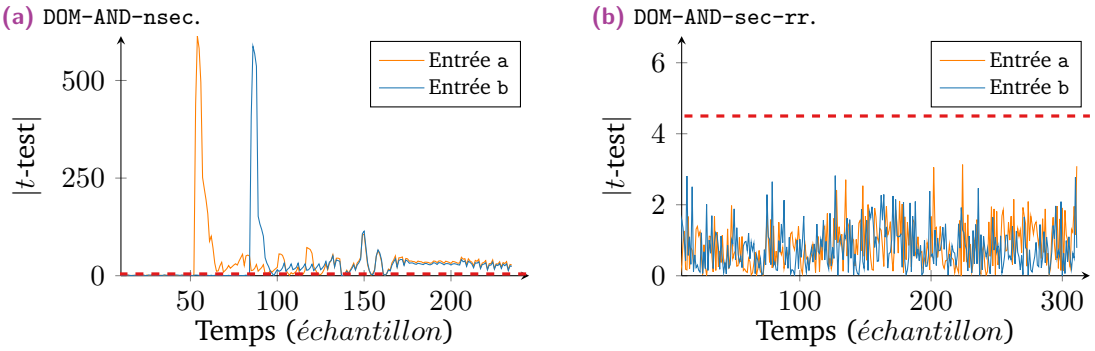


Fig. 5.15 : Valeurs de t-test : variantes `nsec` (a) et `sec-rr` (b) du `DOM-AND` sur le Cortex-M4.

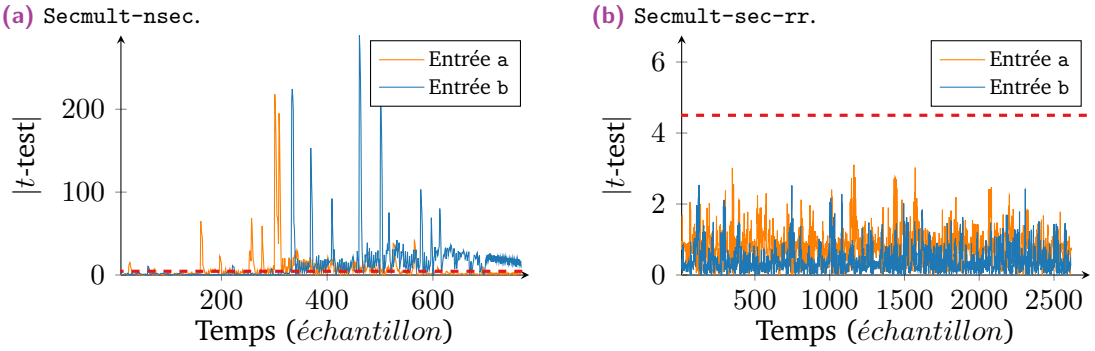


Fig. 5.16 : Valeurs de t-test : variantes `nsec` (a) et `sec-rr` (b) du `Secmult` sur le Cortex-M3.

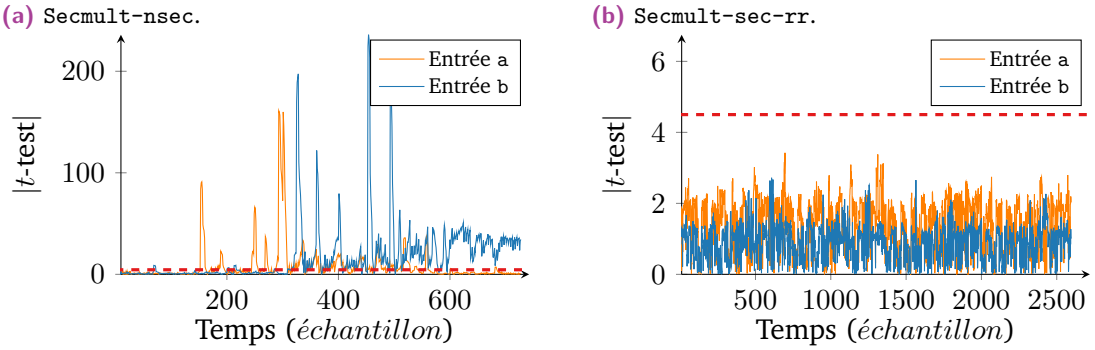


Fig. 5.17 : Valeurs de t-test : variantes `nsec` (a) et `sec-rr` (b) du `Secmult` sur le Cortex-M4.

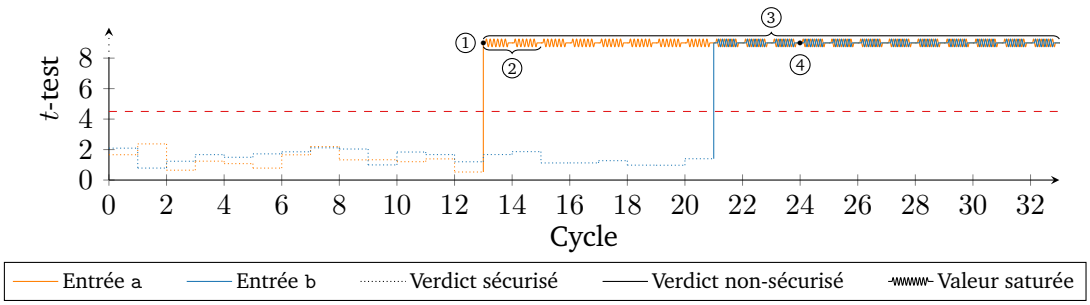


Fig. 5.18 : Valeur maximale du t-test par cycle pour chaque secret et verdict d'`aLEAKator` associé dans le modèle `RR 1-sw-probing` pour le `DOM-AND-nsec` sur le Cortex-M4.

Comparaison cycle-à-cycle des verdicts de fuite avec des mesures réelles

Cette sous-section présente des comparaisons cycle-à-cycle entre les verdicts d'`aLEAKator` et des t-tests calculés sur des mesures réelles. Les sorties d'`aLEAKator` sont exploitées pour expliquer les fuites observées en pratique, en faisant le lien entre la microarchitecture et le programme. Trois programmes sont étudiés sur le `Cortex-M4` : le `DOM-AND-nsec` (figure 5.18), le `Secmult-nsec` (figure 5.21) et un micro-programme additionnel nommé `Refresh` (figure 5.20).

Alignement des traces de verdict d'`aLEAKator` avec les traces de t-test. Pour chaque cycle, quatre valeurs de t-test sont calculées, une par échantillon. Les résultats d'`aLEAKator` étant précis au cycle, la valeur de t-test considérée pour un cycle est donc la valeur maximale du t-test parmi les quatre valeurs de ce cycle. De plus, comme les circuits réels peuvent être différents des modèles `Verilog` fournis par ARM (hiérarchie mémoire et délais associés par exemple), des décalages sur les cycles correspondant aux instructions peuvent se produire entre les traces simulées et les traces mesurées. Par exemple, sur les circuits réels, les mémoires peuvent prendre des délais différents de ceux simulés par `aLEAKator`. Les traces de cette section sont donc resynchronisées par décalage avec bégaiement des traces mesurées et les traces simulées. Cette resynchronisation concerne au maximum 5% des traces présentées. De la même manière que dans la sous-section précédente, les valeurs de t-test supérieures à 4,5 sont considérées comme des fuites. Ces résultats renforcent les résultats précédents en montrant qu'`aLEAKator` associe bien un verdict de fuite à chaque cycle qui présente un dépassement du seuil de t-test en pratique.

DOM-AND-nsec. Dans cette expérience, chaque fuite identifiée par `aLEAKator` correspond à un dépassement du seuil de t-test en pratique. L'analyse des sources de fuites est centrée sur quatre sources de fuites, numérotées sur la figure 5.18.

① La première fuite a lieu au 13ème cycle. `aLEAKator` signale que la flip-flop `HRDATAS_RAM` est source d'une fuite en transition sans glitch et indique la ligne `HDL` qui correspond à

cette flip-flop. En exploitant le PC (pour *P*rogram *C*ounter) et les valeurs symboliques du signal `HRDATAS_RAM` au cycle précédent et au cycle courant rapportées par `aLEAKator`, les instructions données dans le listing 5.c sont identifiées comme source de la fuite. Elles sont responsables du chargement des deux shares du secret `a` dans le banc de registres.

```
1 ldr r2, =a0  // Lecure relative au PC: @a0 dans r2
2 ldr r0, [r2] // Lecture de a0 dans r0
3 ldr r2, =a1  // Lecture relative au PC: @a1 dans r2
4 ldr r1, [r2] // Lecture de a1 dans r1
```

Listing 5.c : Chargement des shares `a0` et `a1` dans le `DOM-AND-nsec`.

Le premier et le troisième `ldr` sont des lectures mémoires relatives au PC. Elles permettent de lire les adresses des shares `a0` et `a1` qui sont sauvegardées dans la section de code du programme. Ces valeurs transitent sur le bus de la mémoire de programme et non sur le bus de la mémoire de données. Le second et le quatrième `ldr` sont des lectures dans la mémoire de données, qui transitent sur le bus de données, nommé `HRDATAS_RAM`. Comme le troisième `ldr` ne transite pas sur le même bus, la donnée du premier share présente sur `HRDATAS_RAM` persiste jusqu'à l'exécution du quatrième `ldr` qui charge le second share, causant donc une transition du premier share vers le second.

② La fuite en transition sans glitch signalée au 13ème cycle (①) est aussi signalée comme étant une fuite en valeur avec glitches (déductible du cas concernant les registres de l'équation 5.11) et comme une fuite en transition avec glitches (déductible de la ligne *Ensemble d'expressions* dans le tableau 5.7) aux cycles 13 et 14.

③ `aLEAKator` signale des fuites en valeur avec glitches à partir du 13ème cycle, qui réalise l'écriture du second share du secret `a` dans le banc de registres. Ces fuites sont liées au signal `rd1_data` du module `u_cm4_dpu_regbank`, qui est la sortie de l'arbre de multiplexeurs correspondant au premier port de lecture du banc de registres. Le premier port de lecture du banc de registres du Cortex-M4 est illustré dans la figure 5.19. Une analyse des ensembles de fuite relevés par `aLEAKator` indique que les deux shares, sauvegardés dans des registres différents, peuvent glitcher sur le port de lecture du banc. Ces éléments indiquent que si le signal de sélection du registre à lire (`rd1_addr[0]`) n'est pas totalement stable, l'arbre de lecture peut notamment causer une observation conjointe des contenus des registres `r0` et `r1`. Cela explique la raison pour laquelle l'ensemble de fuite associé au signal indiqué par la marque ● contient à la fois les ensembles de fuites de `r0` ({`a0`}) et de `r1` ({`a1`}), causant une éventuelle observation en glitches du secret `a`.

④ Au 24ème cycle, `aLEAKator` identifie une fuite liée au banc de registre et à l'exécution de l'instruction `xor` suivante : `eor r4,r4, r3 // r4 = (a0 and b1), r3 = m`. La source identifiée par `aLEAKator` est la flip-flop correspondant au registre `r1` dans le banc de registres. Les ports d'écriture du banc de registres, illustrés dans la figure 5.19, peuvent

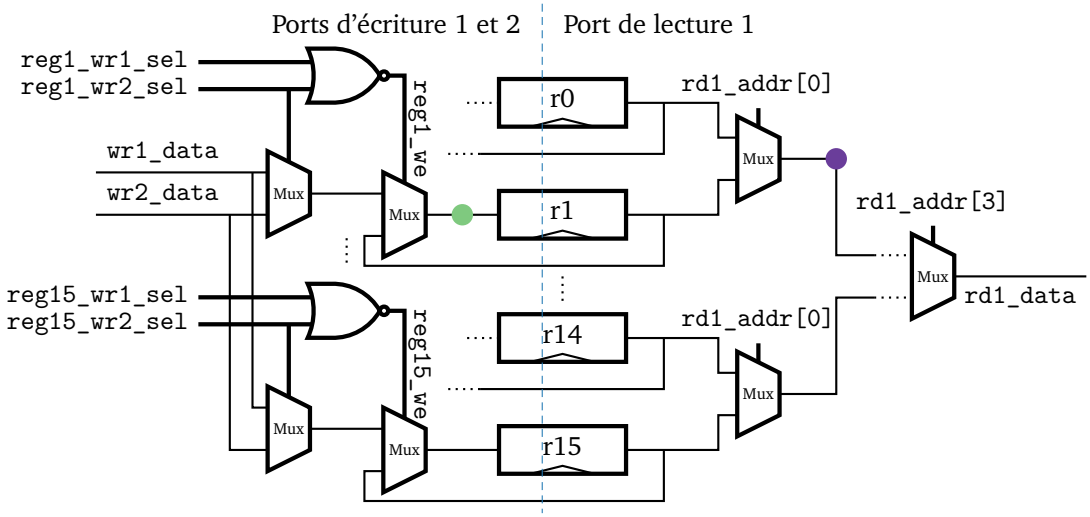


Fig. 5.19 : Vue simplifiée du banc de registres du processeur ARM Cortex-M4.

présenter des glitches entre les données en registre et les données à stocker, si le signal qui régit l'écriture (appelé write_enable) est instable. Lors de l'écriture en registre du résultat de l'instruction `eor`, le signal `reg1_we` est instable. Pour cette raison, l'ensemble de fuite du signal marqué comme ● contient à la fois l'ensemble de fuite du signal du premier port d'écriture (`wr1_data`), celui du second port d'écriture (`wr2_data`) et l'ensemble de fuite du registre `r1`. Dans le cas précis du cycle 24, le port d'écriture 1 fuit la sortie de l'ALU ($\{$`a0 & b1`,`m`$\}$) et l'ensemble de fuite du registre `r1` est $\{$`a1`$\}$, résultant en une fuite en glitches sur le signal ● du secret `a`.

Des phénomènes similaires se produisent pour l'entrée `b`, à partir du chargement du second share de ce secret au cycle 21.

Programme Refresh. Le programme `Refresh`, présenté dans le listing 5.4.2, réalise le chargement d'un share `a0` d'un secret `a`, le masque avec un masque `m` puis le sauvegarde en mémoire. Ensuite, le programme réalise un nettoyage du registre, du bus mémoire et de l'ALU avant de lire le second share du même secret `a1`, de le masquer avec le même masque puis de l'écrire en mémoire.

```
1  // r0=@a0,r1=@a1,r2=@m,r3=@blk
2  ldr r4, [r0] // a0 dans r4
3  ldr r5, [r2] // m dans r5
4
5  eor r4, r4, r5 // Remasquage a0
6  str r4, [r0]
7
8  // Nettoyage LSU en écriture
9  str r3, [r3]
10 eor r4, r4, r4 // Nettoyage r4
11 mov r4, 0x0 // Nettoyage ports ALU ?
12
13 // LSU en lecture déjà nettoyée
14 ldr r6, [r1] // a1 dans r12
15
16 eor r6, r6, r5 // Remasquage a1
17 str r6, [r2]
18 mov r6, 0x0
```

Listing 5.d : Programme `Refresh`. Les `ldr` et `mov` sont encodés en 16 bits et toutes les autres instructions en 32 bits.

L'architecture ARMv7-M, implémentée dans le Cortex-M4, permet d'utiliser des instructions encodées en 16 bits ou en 32 bits. De nombreuses instructions du jeu d'instructions ont plusieurs encodages différents mais fonctionnellement équivalents. Par exemple, l'instruction `ldr` possède deux encodages (un de 16 bits et un de 32 bits) fonctionnellement équivalents mais dont les effets microarchitecturaux ne sont pas strictement identiques [GHM22]. L'instruction `mov` peut être encodée de trois manières différentes (deux versions fonctionnellement différentes en 32 bits[8] et une version 16 bits). Pour les instructions `mov` et `ldr`, la différence de comportement micro-architectural est que le port d'entrée de l'ALU écrit selon l'encodage n'est pas le même.

Dans le programme `Refresh`, le nettoyage des ports de lecture et d'écriture de la mémoire de données est correctement réalisé. `aLEAKator` indique que la fuite est liée à l'ALU, l'évolution des ports d'entrée de l'ALU intervenant dans la fuite sont résumés dans le tableau 5.12. L'instruction `eor` qui nettoie le registre `r4` a pour effet de bord d'écrire la valeur de ce registre, $\texttt{a0} \oplus \texttt{m}$, sur les deux ports d'entrée de l'ALU. L'instruction `mov` qui est exécutée ensuite est encodée au format 32 bits. Cette version de l'instruction écrit uniquement le port `A` de l'ALU (tandis que la version 16 bits aurait uniquement écrit le port `B`). Le `ldr` exécuté ensuite est encodé au format 16 bits. Dans ce format, le port `A` de l'ALU est utilisé pour placer l'adresse à lire, ne modifiant pas la valeur du port `B` qui vaut toujours $\texttt{a0} \oplus \texttt{m}$. Finalement, lors de l'exécution du dernier `eor`, le port `A` de l'ALU effectue la transition de la valeur nettoyée vers le share `a1` ($\texttt{0} \rightarrow \texttt{a1}$) et le port `B` effectue la transition du premier share remasqué vers le masque ($\texttt{a0} \oplus \texttt{m} \rightarrow \texttt{m}$). Aucune de ces transitions ne représente une fuite en transition faisant apparaître le secret `a`. Elles causent cependant des transitions dans les signaux internes de l'ALU, notamment dans les opérateurs multi-cycles. Effectivement, des signaux combinatoires de l'ALU qui utilisaient ces deux ports réalisent une transition d'expressions manipulant $\texttt{a0} \oplus \texttt{m}$ à des expressions manipulant `a1` et `m`, or l'ensemble d'expressions $\{\texttt{a1}, \texttt{a0} \oplus \texttt{m}, \texttt{m}\}$ représente une fuite du secret `a`. Ces transitions sans glitch sont observables en pratique au 18ème cycle, comme illustré dans la figure 5.20. `aLEAKator` relève ces fuites en transition sans glitch et des fuites en transition avec glitches pour ces cycles.

Ces fuites subtiles peuvent être évitées soit en nettoyant le registre `r4` sans le réécrire sur les deux ports de l'ALU (par exemple `eor r4, r13, r13`), soit en forçant l'usage de l'encodage 16 bits pour le `mov` ou de l'encodage 32 bits pour le `ldr`.

Cet exemple illustre l'importance de l'utilisation d'outils automatiques tels qu'`aLEAKator` ou `Coco` : le nettoyage de chemins de données est une tâche complexe, qui nécessite des

8. Une première qui accepte que le registre destination soit *PC*, *SP* ou *LR* et et une seconde, qui doit être préférée et n'accepte pas ces registres de destination `https://developer.arm.com/documentation/ddi0406/cb/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/MOV--immediate-`.

Instruction décodée	Port A	Port B
(10) `eor r4,r4, r4`	a0⊕m	a0⊕m
(11) `mov r4,0x0`	0x0	a0⊕m
(14) `ldr r6,[r1]`	@a1	a0⊕m
(16) `eor r6,r6, r5`	a1	m

Tab. 5.12 : Évolution des expressions symboliques des ports d'entrée `A` et `B` de l'ALU après décodage des instructions 10 à 16 du listing 5.4.2.

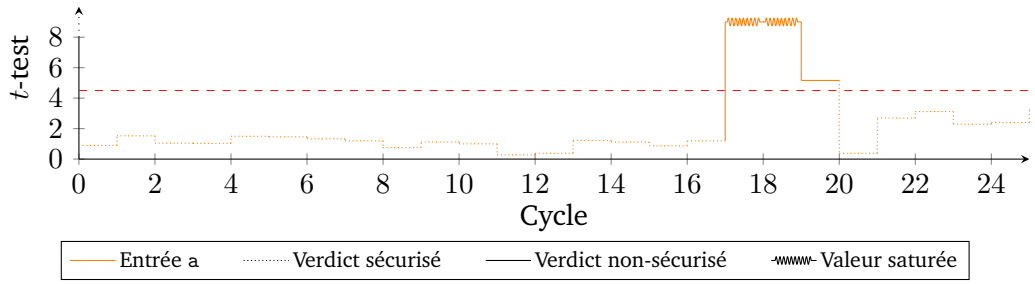


Fig. 5.20 : Valeur maximale du t-test par cycle pour chaque secret et verdict d'`aLEAKator` associé dans le modèle `RR 1-sw-probing` pour le `Refresh` sur le Cortex-M4.

informations avancées sur le fonctionnement de la microarchitecture et qui nécessite de prendre en compte tous les effets de bord des instructions utilisées.

Secmult-nsec. La figure 5.21 montre que tous les cycles présentant un dépassement du seuil de t-test sont correctement identifiés par `aLEAKator`. De plus, de nombreux cycles ne correspondant pas à un dépassement du seuil sont identifiés comme présentant des fuites par `aLEAKator`. Les raisons à cela sont multiples :

1. Une analyse du programme compilé montre que les quatre shares de `a` et `b`, d'un octet chacun, sont sauvegardés dans un unique mot mémoire de 32 bits. Naturellement `aLEAKator` identifie cela comme une fuite en valeur sans glitch (cycles 4-18 dans la figure 5.21), même pour les lectures qui ne lisent qu'un octet de donnée, le mot complet transitant sur le bus mémoire. Ces cycles ne produisent pas de dépassement du t-test, ce qui peut être expliqué par deux raisons :
   a) Le t-test réalisé est de premier ordre alors que la non-indépendance au sein d'un mot est une fuite de second ordre.
   b) Du point de vue d'un seul des deux secrets, les shares de l'autre secret au sein du mot agissent comme du bruit, éventuellement suffisant à masquer les fuites entre elles.
2. Les autres fuites identifiées mais non observables en pratique sont associées à la présence simultanée des shares dans le banc de registres. Comme illustré dans

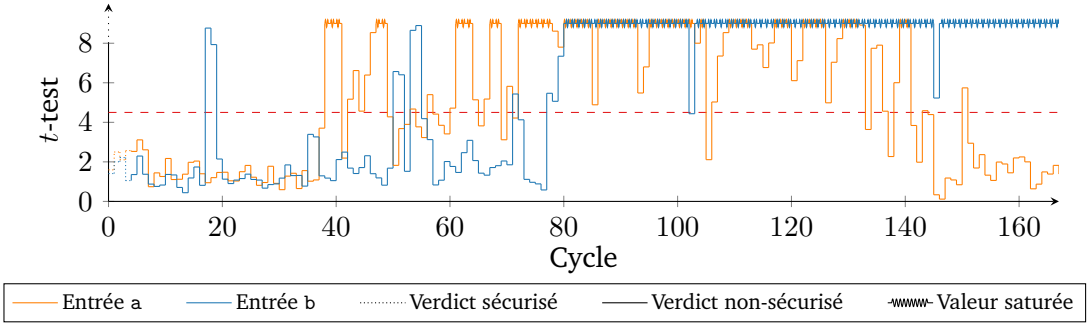


Fig. 5.21 : Valeur maximale du t-test par cycle pour chaque secret et verdict d'aLEAKator associé dans le modèle `RR 1-sw-probing` pour le `Secmult-nsec` sur le Cortex-M4.

la figure 5.19, si tous les shares sont présents en registres et que l'adresse de registre à lire est instable, des glitches les révélant peuvent se produire et se répercuter ensuite dans toute la combinatoire du processeur. Ces glitches ne sont pas nécessairement observables si leur influence sur la consommation globale est masquée par le bruit.

3. L'absence de fuite en pratique pour ces mêmes cycles peut aussi être liée à la sur-approximation réalisée lors de l'estimation des glitches, dont seul un sous-ensemble se produit en pratique.
4. Il n'y a pas de garantie que la microarchitecture vérifiée soit strictement identique à celle du processeur réel, ce qui peut être la cause de décalages entre les verdicts d'aLEAKator et les mesures.

Cette analyse cycle-à-cycle des verdicts d'aLEAKator permet de mettre en évidence les sources des fuites observées en pratique. Les versions sécurisées des programmes vérifiés dans les sections précédentes ont été développées à l'aide de barrières mémoires, de réorganisation des données et des instructions aux points clés des programmes à l'aide des sorties d'aLEAKator.

5.4.3 Ordre supérieur et autres propriétés de sécurité

Les résultats présentés jusqu'à présent correspondent à des vérifications au premier ordre de la propriété de sécurité TPS. Cependant aLEAKator supporte les vérifications à l'ordre supérieur et des propriétés de sécurité alternatives, pour lesquels cette sous-section apporte des résultats de vérification.

Vérifications à l'ordre supérieur. Plusieurs manières de considérer l'ordre supérieur existent. Comme évoqué dans la section 3.1, les combinaisons de d-éléments peuvent être spatiales, temporelles ou spatio-temporelles [HB21 ; MM22]. Quelle que soit la méthode choisie, les informations sont déjà simulées par aLEAKator et l'ordre de vérification n'est

considéré que par l'orchestrateur de vérification. aLEAKator permet actuellement les combinaisons spatiales et les combinaisons temporelles mais n'offre pas d'optimisations autres que les caches pour limiter le nombre d'ensembles d'expressions envoyés au vérificateur. L'ajout d'optimisations est un sujet qui reste à explorer pour permettre la vérification de circuits plus complexes à l'ordre supérieur. En effet, le nombre de d-uplets à vérifier croît rapidement avec le nombre p de signaux[9], le nombre de cycles à simuler, ou le produit des deux pour l'ordre supérieur spatio-temporel.

Le tableau 5.13 donne des résultats de vérification reproduits avec aLEAKator de verdicts existants par des outils de l'état de l'art sur les gadgets suivants pour l'ordre supérieur spatial : DOM-KECCAK [GSM17] (Coco et Prover) et DOM-AND [GMK16] (VerifMSI). Les verdicts d'aLEAKator sont cohérents avec les verdicts des outils de l'état de l'art. Le nombre de d-uplets explose rapidement ; la vérification à l'ordre 3 du DOM-KECCAK protégé à l'ordre 4 nécessiterait la vérification de plus de 34 milliards de triplets.

Matériel masqué	Cycles	Ordre de sécurisation	Ordre de Vérification	Verdict aLEAKator	# d-uplets à vérifier
DOM-AND	2/2	1	2	✗	240
DOM-AND	2/2	2	2	✓	666
DOM-AND	2/2	2	3	✗	15 540
DOM-AND	2/2	5	4	✓	450 662 52
DOM-KECCAK	2/2	1	1	✓	918
DOM-KECCAK	2/2	2	2	✓	1 047 552
DOM-KECCAK	2/2	2	3	✗	356 866 048
DOM-KECCAK	2/2	3	2	✓	22 028 942

Tab. 5.13 : Résultats de vérification à l'ordre supérieur d'aLEAKator.

Autres propriétés de sécurité. aLEAKator a initialement été créé pour la vérification de la propriété de sécurité TPS, plus commune pour les implémentations logicielles. L'outil peut cependant être utilisé pour les propriétés NI et SNI [Bar+15 ; Bar+16]. À nouveau, seul l'orchestrateur prend en compte la propriété de sécurité ; elle n'a aucune influence sur la simulation. À ce titre, le support de propriétés de sécurité additionnelles telles que PINI [CS20] ne nécessite que la récupération des informations utiles et leur envoi au vérificateur externe. Cependant, l'orchestrateur est à nouveau responsable des optimisations pour limiter la complexité des vérifications. Actuellement, les vérifications de propriété NI exploitent les mêmes optimisations que pour la propriété TPS mais ces optimisations ne sont pas valides pour la propriété SNI. Le développement d'optimisations pour SNI et le support de PINI sont donc des objectifs d'amélioration d'aLEAKator à court terme.

9. ou de bits de signaux pour les modèles de fuite en bit

Matériel masqué	Cycles	Ordre de sécurisation	Ordre de vérification	Propriété de sécurité	Avec glitches	Verdict aLEAKator
DOM-AND	2/2	2	2	NI	✗	✓
DOM-AND	2/2	2	2	NI	✓	✓
DOM-AND	2/2	2	2	SNI	✗	✓
DOM-AND	2/2	2	2	SNI	✓	✗
ISW-AND	2/2	2	2	NI	✗	✓
ISW-AND	2/2	2	2	NI	✓	✗
ISW-AND	2/2	2	2	SNI	✗	✓
ISW-AND	2/2	2	2	SNI	✓	✗

Tab. 5.14 : Résultats de vérification des propriétés NI et SNI sur les gadgets DOM-AND et ISW-AND, à l'ordre supérieur.

Pour illustrer le support d'aLEAKator de ces propriétés de sécurité, le tableau 5.14 recense des verdicts de vérifications qui reproduisent les résultats de l'état de l'art. Des vérifications de propriétés NI et SNI sont données à l'ordre supérieur spatial pour les gadgets ISW-AND [ISW03] et DOM-AND [GMK16] au second ordre avec et sans glitch. Le DOM-AND est réalisé avec des registres supplémentaires sur les sorties pour prendre en compte la présence de glitches lorsque les entrées changent. Tous ces résultats corroborent les verdicts fournis par les auteurs de VerifMSI [MT23].

5.4.4 Discussion

Cette sous-section expose les différents points forts et éléments limitants d'aLEAKator, identifiés lors de son développement et de son utilisation.

Concrétisations. Le flot de contrôle des implémentations vérifiées est supposé indépendant des valeurs d'entrée symboliques (secrets, masques, shares et variables publiques). Au cours de la simulation en domaines mixtes, cela se traduit par la contrainte que tous les signaux de contrôle doivent avoir une expression symbolique exprimée sous la forme d'une CST et que leurs ensembles de fuites doivent être équivalents à $\varnothing$. Ce n'est cependant généralement pas le cas pour les processeurs, par exemple en raison de recombinaisons microarchitecturales comme le calcul des drapeaux de statut de contrôle. Lorsque de tels cas sont rencontrés par aLEAKator, le choix a été fait, pour conserver la complétude de la preuve, de propager des expressions symboliques qui ne sont pas construites avec des CST, en partant du principe que le programme final n'en dépendra pas. De même, pour les signaux de contrôle, les ensembles de fuite associés peuvent ne pas être $\varnothing$, afin de représenter correctement que des données peuvent glitcher sur les signaux de contrôle.

Une manière alternative serait de concrétiser ces signaux, c'est-à-dire utiliser les valeurs concrètes pour fixer la valeur des expressions symboliques, ce qui aurait pour effet de rendre la preuve de sécurité incomplète. Le choix réalisé pour la conception d'`aLEAKator` s'est révélé suffisant pour toutes nos expériences mais certains opérateurs pourraient, par exemple, rendre symbolique le PC, ce qui rendrait la simulation de cycles supplémentaires impossible. Un tel exemple serait la multiplication d'un share sur le Cortex-M3, pour lequel l'exécution dépend microarchitecturalement de si la valeur manipulée vaut 0 en raison de la présence d'un raccourci d'exécution pour les multiplications triviales [10]. L'estimation de si une variable symbolique est égale à 0 rendrait le PC symbolique, mettant fin à la simulation.

Passage à l'échelle. Comme pour les autres méthodes formelles présentées dans la section 3.4 et avec lesquelles `aLEAKator` est comparé, il existe des limites sur les circuits et logiciels pouvant être vérifiés. Les structures de données nécessaires pour obtenir une simulation et une vérification efficaces sont complexes et occupent beaucoup de mémoire. Le coût en temps et en mémoire dépend grandement de la structure du circuit et de la nature des expressions générées. Cela peut être observé dans la section 5.4, avec de petits accélérateurs matériels plus complexes à vérifier que des processeurs simulant des programmes complets.

Faux-positifs. Notre méthode n'est pas exempte de faux-positifs puisqu'un verdict non-sécurisé d'`aLEAKator` indique soit que `VerifMSI++` a réussi à prouver qu'une expression n'était pas sécurisée, soit qu'au contraire l'outil n'a pas réussi à obtenir la preuve qu'une expression est sécurisée (hors recours à l'énumération). Les autres méthodes formelles telles que `Coco` souffrent aussi de faux-positifs, comme démontré par les auteurs de `Prover` [ZCF24].

Connaissances limitées des processeurs. `aLEAKator` a été utilisé dans ces travaux sur cinq processeurs différents, sans nécessiter d'informations sur leur fonctionnement interne. Les sorties précises d'`aLEAKator` facilitent la compréhension des fuites, même si leur correction pour créer des programmes sécurisés nécessite tout de même une expertise de conception logicielle ou matérielle.

Intérêt des expressions symboliques. L'utilisation d'expressions symboliques pour la simulation et la vérification offre de nombreux avantages. Premièrement, cela permet de réaliser le calcul de stabilité en exploitant la valeur du signal pour les portes qui le permettent (notamment les OU, ET ainsi que les MUX). Ce calcul fin de la stabilité permet une réduction de la sur-approximation réalisée lors de l'estimation des glitches. Deuxièmement, cela permet la surcharge de la mémoire avec des fonctions dépendant des

10. La multiplication n'étant donc pas exécutée en temps constant, elle est à proscrire pour des applications cryptographiques.

indices, qui ne serait pas possible avec des méthodes qui ne conservent pas l'interaction exacte des variables secrètes au sein du calcul. Troisièmement, les expressions permettent le recours à l'énumération pour éviter les faux-positifs. Finalement, il est possible, à tout instant de l'exécution, de concrétiser les éléments symboliques d'une expression pour comparer son résultat avec le résultat concret. Cela permet de confirmer que le calcul réalisé est le bon et que l'état symbolique est cohérent avec l'état concret.

Ces avantages ne sont pas accessibles en utilisant `Coco`, qui exploite des *ensembles de corrélation* qui permettent de déterminer quelles variables secrètes participent à un calcul mais pas d'en connaître l'interaction exacte.

5.4.5 Conclusion

Ce chapitre a été consacré à la définition d'une nouvelle méthode de représentation, de simulation et de vérification des accélérateurs matériels et des microarchitectures exécutant des implémentations logicielles. Cette méthode, implémentée dans l'outil `aLEAKator`, a été validée sur des résultats existants de la littérature. De plus, des résultats comparés et détaillés, cycle-à-cycle, peu courants pour les méthodes formelles de vérification similaires ont été présentés. Finalement, `aLEAKator` permet la vérification de ces implémentations dans des modèles de fuite de l'état de l'art, en considérant la granularité de vérification et pour des ordres et des propriétés de sécurité variées.

Plusieurs implémentations sécurisées ont été créées manuellement pour présenter ces résultats. Différentes sources de fuite subtiles ont été identifiées et détaillées, mettant en évidence la complexité de la création d'implémentations logicielles sécurisées. Le chapitre 6 est dédié à une analyse des sources communes de fuites dans l'objectif de proposer des méthodes automatiques de sécurisation des logiciels pour des microarchitectures données.

6 Sources de fuites, contre-mesures et résultats d'élimination des fuites

Les chapitres précédents ont présenté une formalisation des modèles de fuite, ainsi que l'outil `aLEAKator` et son fonctionnement, permettant notamment de détecter précisément les fuites de secret lors de l'exécution d'un programme sur un processeur.

Les sources possibles de fuite de secret au sein des processeurs non sécurisés sont multiples. Le travail d'implémentation de programmes masqués sécurisés est fastidieux, même avec l'utilisation d'outils formels tels qu'`aLEAKator` qui indiquent l'origine et la nature des fuites. Effectivement, en présence de programmes de plusieurs centaines d'instructions, la correction manuelle des fuites devient chronophage. Des méthodes de résolution automatique de fuites au sein de programmes exécutés sur des processeurs existent et ont été présentées dans la section 3.5. Ces méthodes ne permettent cependant pas d'exploiter les informations de la microarchitecture mais uniquement la localité temporelle des fuites et les instructions qui y sont associées, sans information sur les données actuellement présentes dans l'état du circuit. Elles ne peuvent donc pas identifier les glitches, détecter quel composant du circuit est à l'origine d'une fuite, ni proposer de corrections informées par l'état courant du circuit. Ces méthodes ne permettent pas de conclure systématiquement à la résolution des fuites identifiées. Comme elles n'exploitent pas d'outils formels de détection, elles sont limitées aux détections des méthodes par analyse statistique. Pour ces raisons, les contre-mesures sont soit limitées à de la dissimulation (comme dans *A First Step Towards Automatic Application of Power Analysis Countermeasures* [Bay+11]), soit à des contre-mesures de nettoyage uniquement informées par l'instruction source de la fuite (comme dans `ROSITA` et `ROSITA++` [She+21b; She+21a]). Le nettoyage uniquement guidé par l'instruction qui fuit n'est pas suffisant dans le cas général d'un processeur pipeliné car il ne prend pas en compte le fait que l'instruction qui est la source de la fuite, notamment en raison des registres intermédiaires de pipeline, n'est pas nécessairement l'instruction courante.

La conception de programmes masqués sécurisés est donc une étape complexe, chronophage, source d'erreurs. Les outils proposant une correction automatique, dans l'objectif de passer à l'échelle pour des programmes complexes, ne permettent pas une résolution systématique et ne fournissent pas de garantie formelle de sécurité du résultat.

Cependant, certaines sources de fuite sont communes à tous les processeurs n'implémentant pas de contre-mesures matérielles contre les attaques par canaux auxiliaires. Cela permet la généralisation de ces sources avec l'objectif de produire des processeurs sécurisés face à celles-ci ou des contre-mesures logicielles. `aLEAKator` dispose de toutes les informations sur l'état d'un circuit au cours de sa simulation ainsi que sur la nature et l'origine des fuites lorsqu'elles sont détectées. C'est donc un support adapté pour l'analyse des sources communes de fuite ainsi que pour la proposition de contre-mesures logicielles contre celles-ci. Ce chapitre est dédié à l'établissement d'une nouvelle méthode permettant, à l'aide des informations d'`aLEAKator`, d'éliminer les fuites de manière automatique. Ces travaux n'ont pas encore fait l'objet d'une publication, mais ont vocation à être publiés prochainement.

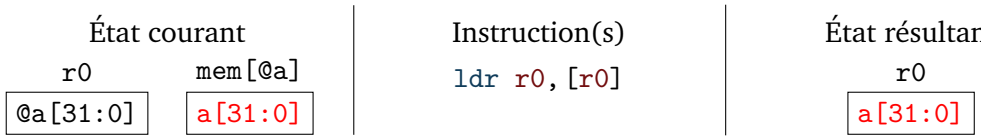
Ce chapitre est structuré en deux parties : un inventaire des fuites courantes dans les processeurs, accompagné d'exemples de contre-mesures évaluées expérimentalement, suivi de l'étude et de la mise en œuvre d'une méthode de réécriture de programme basée sur un système expert exploitant les informations d'`aLEAKator` pour éliminer itérativement les fuites d'un programme.

6.1 Sources communes de fuite et leurs contre-mesures

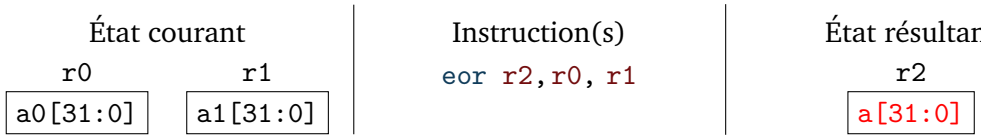
Cette section vise à détailler les sources communes de fuite, qu'elles soient dues à des erreurs d'implémentation d'un schéma de masquage, aux transformations réalisées par la compilation ou plus généralement à des manipulations usuelles des données par la microarchitecture.

Les exemples de cette section sont illustrés avec un masque `m` et un secret `a`, séparé en deux shares `a0` et `a1` tel que `a = a1` $\oplus$ `a0`. Les éléments présentant des fuites sont représentés en rouge.

Les exemples présentant un code `C` et différents codes `Assembleur` associés montrent différents résultats de compilation possibles pour ce même programme source. Tous les codes assembleur présentés sont issus de compilations réelles réalisées par les compilateurs `clang` ou `gcc`, avec différents niveaux d'optimisation. Sauf indication contraire, les t-tests de ce chapitre ont été calculés sur un million de traces de consommations mesurées sur le processeur ARM `STM32F303`.



(a) *Absence de masquage* : le secret `a` apparaît directement dans `r0`, l'état courant (mémoire) ainsi que l'état résultant (registre) fuient.



(b) *Erreur d'implémentation du schéma de masquage* : le *xor* de deux shares est réalisé, permettant de recomposer le secret `a` dans `r2`.

Fig. 6.1 : Sources de fuite en valeur granularité bit (modèle `1-bit-probing`).

6.1.1 Fuites en valeur

Fuites en valeur - Granularité bit

Les fuites en valeur en granularité bit (`1-bit-probing`) indiquent une *absence de masquage* (tel qu'illustré dans la figure 6.1a) ou une *erreur d'implémentation du masquage* (tel qu'illustré dans la figure 6.1b).

La résolution de ces fuites dépend du niveau de représentation du programme. Si le programme (implémenté en assembleur ou en langage de plus haut niveau) présente des défauts de conception, la seule contre-mesure est la correction de l'implémentation du masquage. Bien que la résolution soit propre au programme, les expressions détaillées ainsi que les signaux à l'origine des fuites fournis par `aLEAKator` aident grandement à cette tâche.

À contrario, si ces fuites résultent d'un programme implémentant correctement un schéma de masquage dans un langage compilé (en `C` par exemple), alors l'étape de compilation en est la cause. En effet, les compilateurs peuvent altérer un programme source de plusieurs manières, soit en réordonnançant certaines opérations (figure 6.2), soit en supprimant des opérations fonctionnellement inutiles (figure 6.3). Les compilateurs peuvent légitimement réaliser ces transformations, tant que le code qui en résulte est fonctionnellement équivalent à la description initiale, sans tenir compte du masquage.

Il est cependant possible de contraindre localement les compilateurs à ne pas réordonnancer ni supprimer d'opérations. Les *barrières mémoire* (`__asm__ __volatile__ ("": : : "memory");`), forcent les compilateurs à sauvegarder en mémoire les résultats des calculs intermédiaires réalisés avant leur traversée, imposant donc un ordre d'exécution lorsque plusieurs barrières sont utilisées et empêchant la détection d'opérations pouvant être optimisées au-delà de celles-ci. Cette contrainte vient au coût de l'invalidation de toutes les

```
uint32_t x = (a1 ^ mt) ^ m // Avec a1 = a ^ m
```

Listing 6.a : Remasquage de `a` $\oplus$ `m` par un masque `mt`, pour obtenir `x = a` $\oplus$ `mt`.

```
// r0=@a1, r1=@mt, r2=@m, r3=@x
ldr r4, [r0] // r4<-a1
ldr r5, [r1] // r5<-mt
ldr r6, [r2] // r6<-m

// Masquage par mt en premier
eor r7, r4, r5 // r7<-a1^mt
eor r8, r7, r6 // r8<-a1^mt^m

str r8, [r3] // r8->x
```

Listing 6.b : Version sécurisée, `mt` est ajouté avant d'enlever le masquage par `m`.

```
// r0=@a1, r1=@mt, r2=@m, r3=@x
ldr r4, [r0] // r4<-a1
ldr r5, [r1] // r5<-mt
ldr r6, [r2] // r6<-m

// Démasquage par m en premier
eor r7, r4, r6 // r7<-(a^m)^m
eor r8, r7, r5 // r8<-a1^m^mt

str r8, [r3]
```

Listing 6.c : Version exposant le secret a, `m` est enlevé avant le masquage par `mt`.

Fig. 6.2 : Altération du compilateur : réordonnancement d'opérations menant à des fuites dans le modèle `1-bit-probing`.

```
uint32_t o; // Variable globale
uint32_t x, y, z; // Variables locales
x = a0 ^ m;
y = x ^ a1; // y<-a0^m^a1
z = y ^ mt; // z<-a0^m^a1^mt
o = z ^ m;  // o<-a0^a1^mt
```

Listing 6.d : Recombinaison des deux shares d'un secret au sein d'une même variable et remasquage par `mt`.

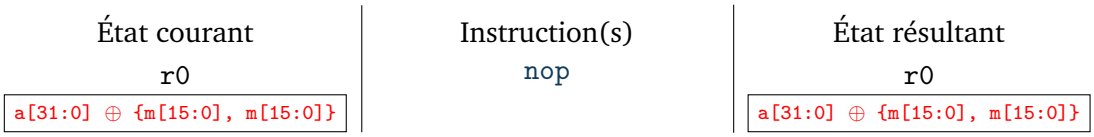
```
// r0=@a0, r1=@a1, r2=@m, r3=@mt,
    r4=@o
ldr r4, [r0] // r4<-a0
ldr r5, [r1] // r5<-a1
ldr r6, [r2] // r6<-m
ldr r7, [r3] // r7<-mt

eor r8, r4, r6 // r8<-a0^m
eor r8, r8, r5 // r8<-a1^a0^m
eor r8, r8, r7 // r8<-a1^a0^m^mt
eor r8, r8, r6 // r8<-a1^a0^mt

str r8, [r9]
```

Listing 6.e : Version sécurisée, toutes les opérations décrites en `C` sont effectuées.

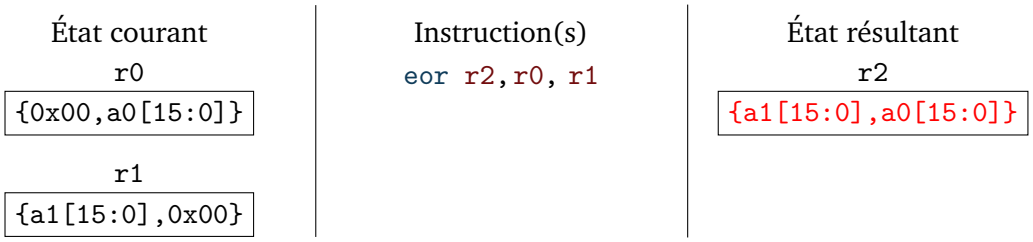
```
// r0=@a0, r1=@a1, r2=@mt, r4=@o
ldr r4, [r0] // r4<-a0
ldr r5, [r1] // r5<-a1
ldr r6, [r2] // r6<-mt

eor r4, r4, r5 // r4<-a1^a0
eor r4, r4, r6 // r4<-a1^a0^mt

str r4, [r3]
```

Listing 6.f : Version non sécurisée, les opérations liées au masque `m` ne sont pas effectuées, exposant a.

Fig. 6.3 : Altération du compilateur : suppression d'opérations menant à des fuites dans le modèle `1-bit-probing`.



(a) *Non indépendance d'un masque* : seulement 16 bits de masque sont utilisés pour masquer les 32 bits de secrets dans `r0`, l'état courant ainsi que l'état résultant fuient.



(b) *Recombinaison dans un mot* : les deux shares sont présents ensemble dans `r2`, l'état résultant fuit en granularité support.

Fig. 6.4 : Sources de fuite en valeur granularité support (modèle `1-sw-probing`).

valeurs lues depuis la mémoire actuellement en registre à chaque traversée de barrière. Ces valeurs devront être rechargées si elles sont réutilisées par la suite.

Fuites en valeur - Granularité support

Les fuites en valeur liées à la granularité support, telle que définie dans la section 3.3, peuvent être causées par la *non-indépendance de masques* (tel qu'illustré dans la figure 6.4a) ou par la *recombinaison de shares* au sein d'un mot (tel qu'illustré dans la figure 6.4b).

La distinction entre les différents niveaux d'implémentation des programmes est à nouveau faite. Pour les programmes `Assembleur`, le concepteur doit revoir les étapes du programme qui violent l'indépendance des masques, éventuellement assisté d'`aLEAKator`.

Pour une implémentation de plus haut niveau, en `C` par exemple, correctement masquée, l'origine des fuites d'un programme qui ne présente pas de fuite en granularité bit mais qui en présente en granularité support se trouve dans les optimisations du compilateur. Une première source est le placement des éléments en mémoire, où deux shares de taille octet qui composent un secret peuvent être placés au sein d'un même mot de 32 bits qui transitera sur le bus de lecture de la mémoire, même dans le cas d'une lecture d'un seul octet, comme illustré dans la figure 6.5. Une autre optimisation pouvant être utilisée par le compilateur et causant ce type de fuites est le *SLP* (pour *S*uperword *L*evel *P*arallelism), permettant de vectoriser, au sein d'une même instruction assembleur opérant sur des mots, plusieurs opérations de haut niveau réalisées sur des octets. La figure 6.6 illustre ce cas dans lequel les opérations sur des octets sont bien sécurisées mais la manipulation vectorisée ne l'est pas.

```
uint8_t a0, a1, m0, m1, m2, m3;
```

Listing 6.g : Déclaration d'octets de shares et masques.

```
.data
.align 2
// Premier mot de 32 bits
a0: .byte
m0: .byte
m1: .byte
m2: .byte
// Second mot de 32 bits
a1: .byte
m3: .byte
```

Listing 6.h : Version sécurisée, les deux mots mémoire de 32 bits ne présentent pas de fuite.

```
.data
.align 2
// Premier mot de 32 bits
a0: .byte
a1: .byte
m0: .byte
m1: .byte
// Second mot de 32 bits
m2: .byte
m3: .byte
```

Listing 6.i : Version non-sécurisée, le premier mot mémoire de 32 bits fuit le secret a.

Fig. 6.5 : Altération du compilateur : organisation mémoire menant à des fuites dans le modèle `1-sw-probing`.

```
char x[4] = {a0,p,p,p}, y[4] = {p,a1,p,p}, z[16]; // Avec var. publique p
// Sortie z: z[0] = a0 ^ p; z[4] = p ^ a1; z[8]=z[12]=0
for (int i = 0; i < 4; i++)
    z[i*4] = x[i] ^ y[i];
```

Listing 6.j : Calcul avec des indices spécifiques entre deux tableaux, dont chaque manipulation d'octet est sécurisée au niveau de représentation `C`.

```
// r4=@x, r5=@y, r6=@z
ldrb r1, [r4] // r1<-a0
ldrb r3, [r5] // r3<-p
ldrb r2, [r4, #1] // r2<-p

// z[0]<-a0^p
eor  r1, r3, r1 // r1<-a0^p
strb r1, [r6]

// z[4]<-p^a1
ldrb r1, [r5, #1] // r1<-a1
eor  r1, r1, r2 // r1<-p^a1
strb r1, [r6, #4]

// Ommission de z[8] et z[12]
```

Listing 6.k : Version sécurisée, le compilateur n'a pas eu recours à la vectorisation et réalise les mêmes opérations que décrites en `C`.

```
// r3=@x, r4=@y, r5=@z
ldr  r1, [r3] // r1<-{p, p, p, a0}
ldr  r2, [r4] // r2<-{p, p, a1, p}

// Vectorisation
// r2<-{0, 0, a1^p, a0^p}
eor  r2, r1, r2

// z[0]=a0^p
strb r2, [r5, #0]

// z[4]=p^a1
// r12<-{0, p, p, a1^p}
lsr  r12, r2, #8
strb r12, [r5, #4]
```

Listing 6.l : Version non-sécurisée, la vectorisation place `a0` et `a1` au sein d'un même mot de 32 bits, exposant le secret a.

Fig. 6.6 : Altération du compilateur : vectorisation au sein d'un mot menant à des fuites dans le modèle `1-sw-probing`.

La modification des types de données permet de forcer les compilateurs à ne pas placer au sein d'un même mot plusieurs octets. Remplacer les variables de largeur octet ou demi-mot en mots de 32 bits permet d'éviter ce regroupement, au prix de la réservation inutile de plusieurs octets en mémoire. Les barrières mémoire logicielles empêchent le compilateur de vectoriser au sein d'un mot, avec les mêmes surcoûts que lorsqu'elles sont utilisées pour conserver toutes les opérations ou empêcher leur réordonnancement.

6.1.2 Fuites liées à la mémoire

Les processeurs communiquent avec des mémoires via des bus pour récupérer des instructions à exécuter ou des données sur lesquelles réaliser des opérations ainsi que pour sauvegarder les résultats de ces opérations. Le concepteur d'un microcontrôleur peut choisir d'utiliser une unique mémoire pour les instructions et les données ou de séparer l'organisation mémoire en une *ROM* pour les instructions et données constantes et une *RAM* pour les données. Les mémoires sont liées aux processeurs via un ou plusieurs ports de lecture et d'écriture. Le choix de la hiérarchie mémoire a une influence sur les performances du microcontrôleur mais aussi sur les fuites liées à la mémoire.

Pour la suite de cette étude, tous les processeurs sont supposés utiliser deux bus mémoire distincts, un pour les instructions et données constantes (section `.text`) et un autre pour les données (sections `.data` et `.bss`).

Les bus respectent les hypothèses suivantes pour les opérations mémoire alignées (dont l'adresse de la donnée lue ou écrite est un multiple de la taille de cette donnée) :

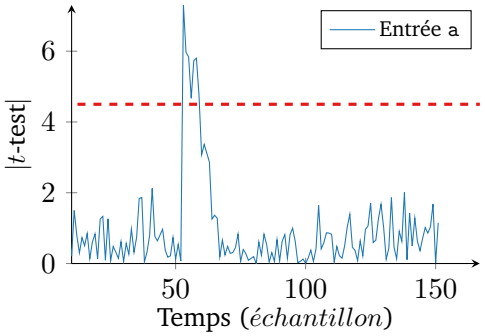
1. Les opérations mémoire de granularité plus petite que le mot (par exemple octet ou demi-mot) entraînent le transit sur le bus d'un mot complet de 32 bits.
2. La conversion de la valeur à *lire* depuis la mémoire vers la granularité demandée par l'instruction est réalisée par le processeur (dans la *LSU*, pour *L*oad *S*tore *U*nit).
3. La conversion de la valeur à *écrire* dans la mémoire vers la granularité demandée par l'instruction est réalisée par le contrôleur de mémoire.
4. La dernière valeur ayant transité sur un bus de lecture est maintenue tant qu'une autre valeur n'est pas lue sur ce même bus.

Toutes ces hypothèses ne sont pas nécessaires pour chaque processeur, mais il est conservatif de les considérer.

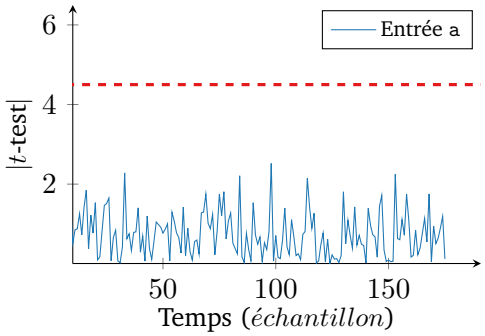
La figure 6.7 illustre par ailleurs que les implantations mémoires ont une influence sur les fuites à l'aide d'un programme assembleur créé pour causer une fuite en transition dans une cellule mémoire. L'exécution de ce programme sur deux microcontrôleurs embarquant des processeurs ARM Cortex-M4 ne donne pas lieu aux mêmes fuites.

```
.data
s0:.word 0 // a0
s1:.word 0 // a1
.text // r0=s0, r1=s1
ldr r2, [r0] // r2<-a0
str r2, [r1] // s1<-a0
```

Listing 6.m : Réécriture d'un share en mémoire par le second share du même secret.



(a) Exécution du listing 6.m sur le `STM32F303`, une fuite est observée.



(b) Exécution du listing 6.m sur le `STM32F405`, aucune fuite n'est observée.

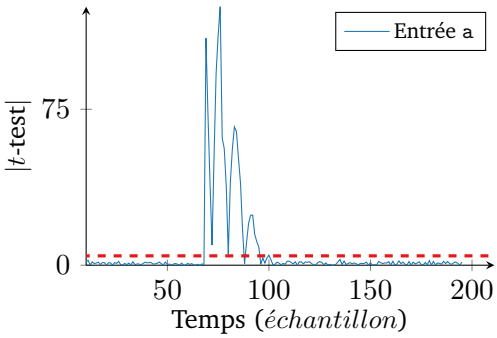
Fig. 6.7 : Différence de comportement pour un million de traces entre les microcontrôleurs `STM32F303` et `STM32F405`, embarquant tous les deux un processeur ARM Cortex-M4, lors de la réécriture en mémoire d'un share par un autre. Le premier share n'est jamais manipulé dans le processeur ni sur les bus mémoire.

Fuites liées à la mémoire - Bus de lecture

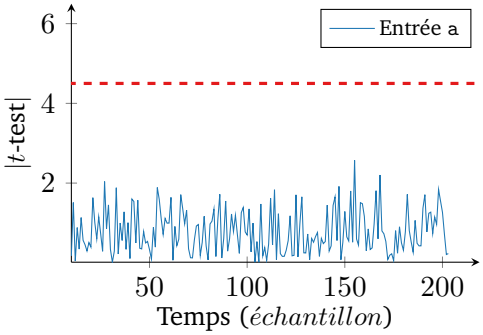
Lorsque deux lectures mémoire sont réalisées, la transition entre les deux valeurs lues est observable, même si elles ne sont pas consécutives dans le programme (cf. hypothèse 4). Le programme donné dans le listing 6.n présente une fuite, visible dans la figure 6.8a. Le bus de lecture étant matérialisé par un registre (qui conserve la dernière valeur lue), des glitches peuvent aussi se produire et faire apparaître dans un même ensemble de fuite la dernière valeur du bus ainsi que la nouvelle, ce qui correspond toujours à la transition sans glitch.

Ces fuites sur le bus de lecture de la mémoire sont fréquentes dans les programmes compilés. Le code produit par les compilateurs pour les architectures permettant de pipeliner les opérations mémoire favorise l'apparition de ces fuites en regroupant au maximum les opérations mémoire pour des raisons de performance mais ces fuites restent très communes sur tout type de programmes.

La contre-mesure principale consiste à nettoyer le bus de lecture afin d'empêcher ces transitions. Lire un mot ne contenant pas de valeur sensible (aucun masque, share ou secret) réalise ce nettoyage. Cette solution est implémentée dans le listing 6.o, et son efficacité est illustrée dans la figure 6.8b.



(a) Exécution du listing 6.n, non-sécurisé.



(b) Exécution du listing 6.o, sécurisé.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  .text // r0=s0, r1=s1
5  ldr r2, [r0] // r2<-a0
6
7  mov r2, #0 // Nett. r2
8  and r2, r2, r2 // Nop (×3)
9
10 ldr r3, [r1] // r3<-a1
```

Listing 6.n : Lecture sans nettoyage préalable: le bus de lecture mémoire réalise `a0` → `a1`.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r1=s1, r9=blk
6  ldr r2, [r0] // r2<-a0
7
8  mov r2, #0 // Nett. r2
9  and r2, r2, r2 // Nop (×3)
10 ldr r9, [r9] // Nett. bus lecture
11
12 ldr r3, [r1] // r3<-a1
```

Listing 6.o : Lecture avec nettoyage préalable: le bus de lecture mémoire réalise `a0` → `blk` → `a1`.

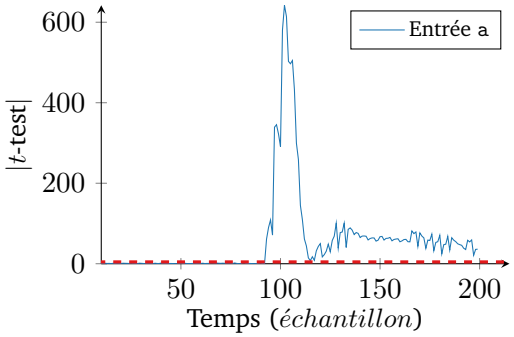
Fig. 6.8 : Vecteur démonstratif : fuite en transition lors d'une lecture mémoire.

Fuites liées à la mémoire - Bus d'écriture

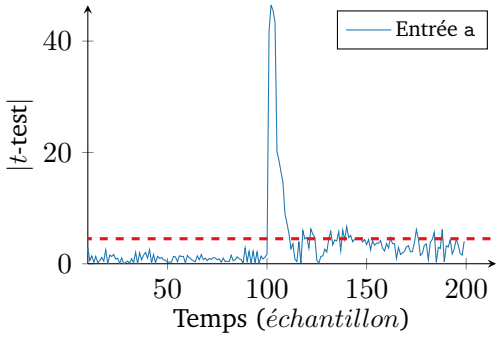
Les écritures consécutives des shares en mémoire sont sources de multiples transitions, notamment sur le bus d'écriture de la mémoire. Le programme donné dans le listing 6.p présente des fuites qui sont illustrées dans la figure 6.9a.

Comme précisé dans la section sur les fuites du bus de lecture, les instructions mémoire sont souvent regroupées par les compilateurs pour être exécutées consécutivement. Les transitions sur le bus d'écriture sont donc aussi fréquente, notamment dans les programmes compilés.

Le nettoyage du chemin d'écriture à l'aide de l'écriture d'une valeur non-symbolique est une contre-mesure efficace. Cependant, l'écriture consécutive de shares implique que ces shares sont présents dans l'état du processeur, ce qui peut mener à des fuites supplémentaires dans d'autres modules du processeur que le bus d'écriture mémoire. Ces autres sources de fuite sont décrites ultérieurement dans cette section. Cette solution est implémentée dans le listing 6.q, et son efficacité est illustrée dans la figure 6.9b.



(a) Exécution du listing 6.p, non sécurisé.



(b) Exécution du listing 6.q, plus sécurisé.

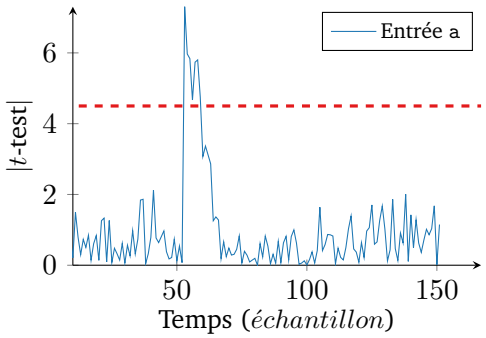
```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk: .word blk
5  .text // r0=s0, r1=s1, r9=blk
6  ldr r2, [r0] // r2<-a0
7  ldr r9, [r9] // Nett. bus lecture
8  and r9, r9, r9 // Nop
9  ldr r3, [r1] // r3<-a1
10 ldr r9, [r9] // Nett. bus lecture
11
12 and r9, r9, r9 // Nop (×4)
13
14 str r2, [r0] // a0->s0
15 str r3, [r1] // a1->s1
```

Listing 6.p : Écritures mémoire sans cycle d'attente: le bus d'écriture mémoire réalise a0 → a1.

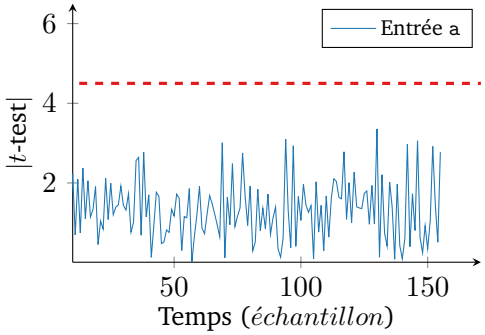
```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r1=s1, r9=blk
6  ldr r2, [r0] // r2<-a0
7  ldr r9, [r9] // Nett. bus lecture
8  and r9, r9, r9 // Nop
9  ldr r3, [r1] // r3<-a1
10 ldr r9, [r9] // Nett. bus lecture
11
12 and r9, r9, r9 // Nop (×4)
13
14 str r2, [r0] // a0->s0
15 str r9, [r9] // Nett. bus écriture
16 str r3, [r1] // a1->s1
```

Listing 6.q : Écritures mémoire avec cycle d'attente: le bus d'écriture mémoire réalise a0 → blk → a1.

Fig. 6.9 : Vecteur démonstratif : fuite en transition lors d'une écriture mémoire. Le correctif ne concerne que le bus mémoire, des fuites dans d'autres modules du cœur persistent.



(a) Exécution du listing 6.r, non sécurisé.



(b) Exécution du listing 6.s, sécurisé.

```
1 .data
2 s0:.word 0 // a0
3 s1:.word 0 // a1
4 .text // r0=s0, r1=s1
5 ldr r2, [r0] // r2<-a0
6 str r2, [r1] // s1<-a0
```

Listing 6.r : Écriture sans nettoyage préalable: une cellule mémoire réalise a1 → a0.

```
1 .data
2 s0:.word 0 // a0
3 s1:.word 0 // a1
4 .text // r0=s0, r1=s1
5 ldr r2, [r0] // r2<-a0
6 str r0, [r1] // Nett. cellule s1
7 str r2, [r1] // s1<-a0
```

Listing 6.s : Écriture avec nettoyage préalable: une cellule mémoire réalise a1 → @a0→ a0.

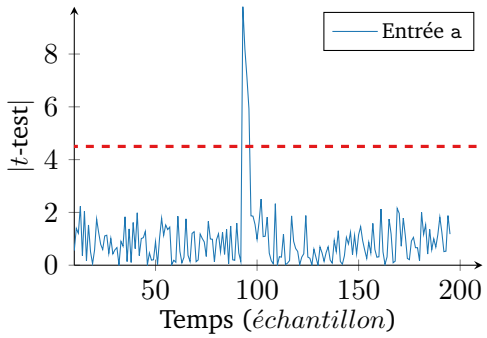
Fig. 6.10 : Vecteur démonstratif : fuite en transition dans une cellule mémoire.

Fuites liées à la mémoire - Cellules mémoire en écriture

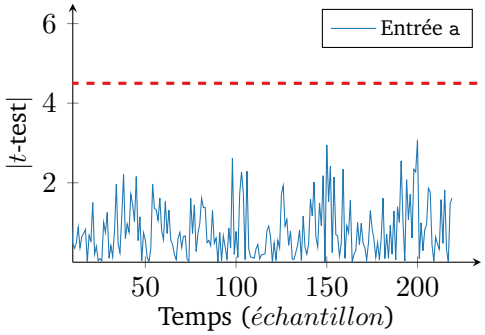
L'écriture d'un share dans une cellule mémoire contenant l'autre share d'un même secret (par réécriture ou par concaténation) peut causer une fuite en transition (en valeur, pour la concaténation) sans glitch dans la mémoire. L'expérience illustrée dans la figure 6.7 a montré que ces fuites ne se produisent pas pour tous les microcontrôleurs. Bien que ce comportement ne soit pas observable pour toutes les configurations ou technologies mémoire, il est nécessaire de présenter l'approche de protection dans ce cas. Les expériences liées à la réécriture de cellules mémoire sont réalisées sur le processeur ARM STM32F303, qui exhibe bien ces fuites. Le programme donné dans le listing 6.r présente une fuite, visible dans la figure 6.10a. La réécriture par une valeur non-symbolique d'une cellule mémoire contenant un share peut théoriquement provoquer une fuite si le bus d'écriture glitche l'autre share (par exemple via le banc de registres). Cependant, nos expérimentations n'ont pas permis de concevoir un programme permettant de visualiser ce cas.

Une source commune d'apparition de ces fuites dans les programmes compilés est l'écriture en pile (de variables locales d'une fonction par exemple) alors que la pile avait été préalablement utilisée pour sauvegarder d'autres variables sensibles.

La contre-mesure principale consiste à nettoyer la case mémoire écrite afin d'empêcher ces transitions. L'écriture d'un mot ne contenant pas de valeur sensible (aucun masque



(a) Exécution du listing 6.t, non sécurisé.



(b) Exécution du listing 6.u, sécurisé.

```
1  .data
2  s0:.quad 0 // 0x00,a0,0x00
3  s1:.quad 0 // 0x00,a1,0x00
4  blk:.word blk, blk
5  .text // r0=s0, r1=s1, r9=blk
6  add r0, r0, 2 // Désalignement
7  add r1, r1, 2 // Désalignement
8
9  ldr r2, [r0] // r2<-a0
10 mov r2, #0 // Nett. r2
11 and r2, r2, r2 // Nop (×3)
12
13 ldr r9, [r9] // Nett. bus lecture
14
15 ldr r3, [r1] // a1 in r3
```

Listing 6.t : Lecture non-alignée sans nettoyage préalable, le registre interne réalise a0 → a1 (avec un désalignement égal sur ces variables).

```
1  .data
2  s0:.quad 0 // 0x00,a0,0x00
3  s1:.quad 0 // 0x00,a1,0x00
4  blk:.word blk, blk
5  .text // r0=s0, r1=s1, r9=blk
6  add r0, r0, 2 // Désalignement
7  add r1, r1, 2 // Désalignement
8
9  ldr r2, [r0] // r2<-a0
10 mov r2, #0 // Nett. r2
11 and r2, r2, r2 // Nop (×3)
12
13 ldr r9, [r9] // Nett. bus lecture
14
15 add r9, r9, 2
16 ldr r9, [r9] // Nett. désaligné
17
18 ldr r3, [r1] // a1<-r3
```

Listing 6.u : Lecture non-alignée avec nettoyage préalable, le registre interne réalise a0 → blk→ a1 (avec un désalignement égal sur ces variables).

Fig. 6.11 : Vecteur démonstratif : fuite en transition lors d'une lecture non-alignée.

ni secret) réalise ce nettoyage. Cette solution est implémentée dans le listing 6.s, et son efficacité est illustrée dans la figure 6.10b.

Fuites liées à la mémoire - Lectures non-alignées

De nombreux processeurs prennent en charge les accès mémoire en lecture non-alignés, c'est-à-dire, pour l'architecture ARM, des lectures dont l'adresse lue n'est pas un multiple de la taille de la donnée en nombre d'octets (par exemple `ldr r0, [pc, 0x1]`, qui lit un mot de 4 octets qui s'étend sur deux cellules mémoire). Ces lectures sont gérées spécialement par les processeurs, puisqu'elles nécessitent de réaliser une première lecture, de décaler le résultat puis de compléter ce résultat avec une seconde lecture et un masque. Cette étape peut avoir lieu dans la LSU (Ibex, CV32E40P) ou dans un module dédié (*bus_matrix* pour les Cortex-M3 et Cortex-M4). Dans les deux cas, seul le résultat final de la lecture

est conservé dans le banc de registres ; un registre interne (dans la LSU ou dans un autre module) est donc réservé à cet effet. Le programme donné dans le listing 6.t présente une fuite, visible dans la figure 6.11a.

Les compilateurs limitent les apparitions de tels accès en raison de leurs coûts, mais ceux-ci peuvent tout de même être générés à la compilation. Des transitions peuvent avoir lieu sur le registre interne et, selon l'implémentation, celui-ci peut n'être réécrit que lors des accès non-alignés. De même, des glitches liés à des parties de la dernière valeur non-alignée lue peuvent persister dans les cycles après la lecture.

La réalisation d'une lecture non-alignée et non-symbolique permet de nettoyer le registre interne. Un exemple de l'application de cette contre-mesure est donné dans le listing 6.u et son efficacité est illustrée dans la figure 6.11b.

Fuites liées à la mémoire - Écritures non-alignées

De manière similaire aux lectures non-alignées, les écritures non-alignées sont réalisées à l'aide d'un registre interne. Ce registre, qui n'est pas nécessairement affecté à tous les cycles selon les architectures, peut aussi être à l'origine de transitions et de glitches difficilement prévisibles. Le programme donné dans le listing 6.v présente une fuite, visible dans la figure 6.12a.

L'apparition de ces fuites est, comme pour les lectures, peu fréquente, mais possible dans les codes compilés. Les fuites qui y sont liées sont difficilement prévisibles, puisqu'un concepteur de programme doit être conscient de la valeur de ce registre pour éviter que celui-ci soit à l'origine d'une fuite.

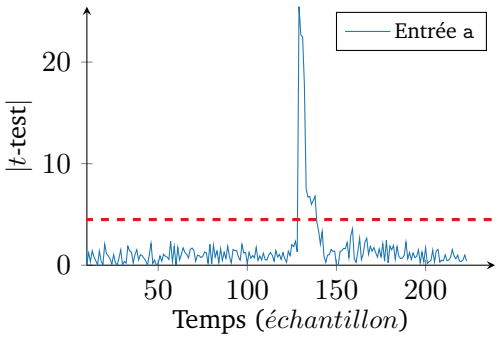
La réalisation d'une écriture non-alignée et non-symbolique permet de nettoyer le registre interne. Un exemple de l'application de cette contre-mesure est donné dans le listing 6.w et son efficacité est illustrée dans la figure 6.12b.

Fuites liées à la mémoire - Divers

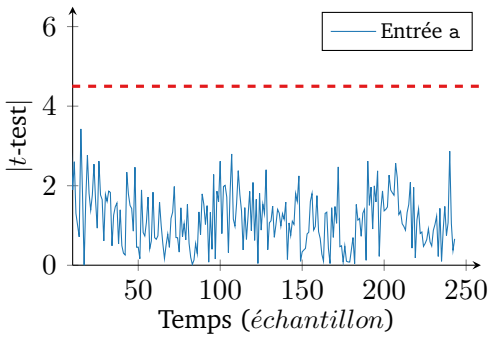
D'autres fonctionnalités des processeurs peuvent être à l'origine de fuites liées à la mémoire. Les deux principales identifiées pour les cœurs ARM Cortex-M3 et Cortex-M4 sont les accès *bit-band* et le *forwarding* d'opérandes.

Les accès bit-band sont des lectures et écritures mémoire dans une zone spécifique de la hiérarchie qui sont traduits [1] en opérations sur des bits, alors qu'usuellement seuls les

1. La notion de *bit-banding* est expliquée en détail dans la documentation ARM : `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Bit-banding`



(a) Exécution du listing 6.v, non sécurisé.



(b) Exécution du listing 6.w, sécurisé.

```
1  .data
2  s0:.quad 0 // 0x00,a0,0x00
3  s1:.quad 0 // 0x00,a1,0x00
4  blk:.word blk, blk
5  .text // r0=s0, r1=s1, r9=blk
6  ldr r2, [r0] // r2<-a0
7  add r0, r0, 2 // Désalignement
8  str r2, [r0] // Écriture
9
10 and r2, r9, r9 // Nett. r2
11 ldr r9, [r9] // Nett. bus lecture
12 and r9, r9, r9 // Nop(×4)
13
14 ldr r3, [r1] // r3<-a1
15 add r1, r1, 2 // Désalignement
16 str r3, [r1] // Écriture
```

Listing 6.v : Écriture non-alignée sans nettoyage préalable, le registre interne réalise a0 → a1 (avec un désalignement égal sur ces variables).

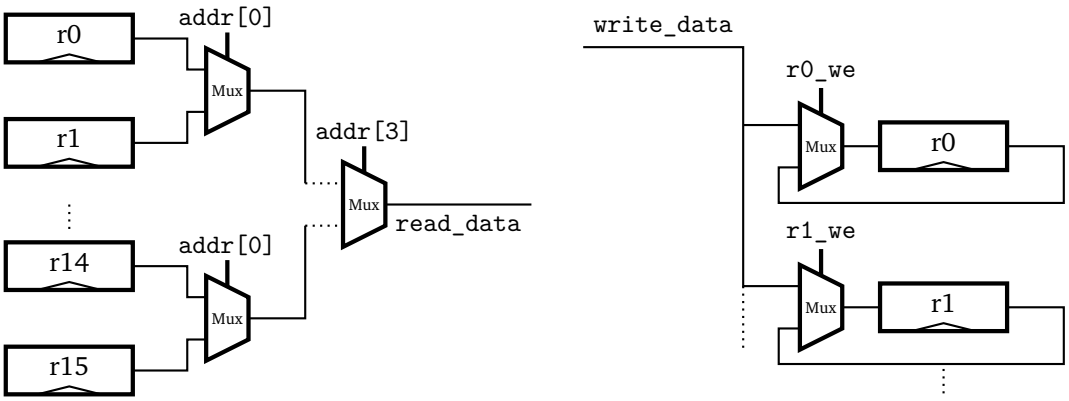
```
1  .data
2  s0:.quad 0 // 0x00,a0,0x00
3  s1:.quad 0 // 0x00,a1,0x00
4  blk:.word blk, blk
5  .text // r0=s0, r1=s1, r9=blk
6  ldr r2, [r0] // r2<-a0
7  add r0, r0, 2 // Désalignement
8  str r2, [r0] // Écriture
9  and r2, r9, r9 // Nett. r2
10
11 ldr r9, [r9] // Nett. bus lecture
12
13 add r9, r9, 2
14 str r9, [r9] // Nett. désaligné
15
16 and r9, r9, r9 // Nop(×4)
17
18 ldr r3, [r1] // r3<-a1
19 add r1, r1, 2 // Désalignement
20 str r3, [r1] // Écriture
```

Listing 6.w : Écriture non-alignée avec nettoyage préalable, le registre interne réalise a0 → blk → a1 (avec un désalignement égal sur ces variables).

Fig. 6.12 : Vecteur démonstratif : fuite en transition lors d'une écriture non-alignée.

octets sont individuellement adressables. Le sous-système qui remplit cette fonction pour les cœurs ARM est le *bus_master*, qui gère aussi les accès mémoire non-alignés. Certains registres sont partagés entre la fonction de bit-banding et celle pour les accès non-alignés. Les contre-mesures pour nettoyer les registres lors des accès non-alignés peuvent donc être nécessaires même lorsqu'aucun accès non-aligné n'a été réalisé, si une fuite liée au registre interne du module *bus_master* est identifiée.

Dans un objectif de performance, les processeurs pipelinés peuvent implémenter des *bypass*, qui sont des connexions entre différents étages du pipeline et qui ont pour but de récupérer plus rapidement des données plutôt que de créer des cycles de gel de pipeline. Les conditions d'utilisation de ces bypass sont complexes et dépendent de l'état du pipeline et des instructions précédemment exécutées. Certains processeurs utilisent des bypass,



(a) Exemple de port de lecture. **(b)** Exemple de port d'écriture.

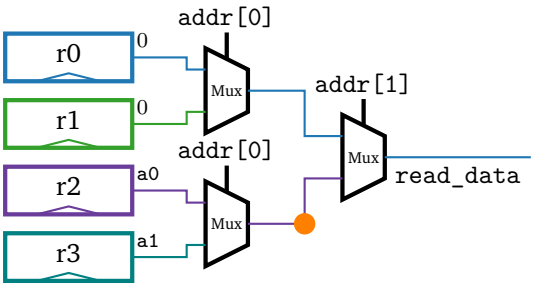
Fig. 6.13 : Exemples de ports de lecture et d'écriture d'un banc de registres.

notamment pour la LSU, de manière à récupérer directement une donnée lue depuis la mémoire pour l'exécution d'une instruction sans repasser par l'écriture et la lecture en registre ou pour utiliser directement un résultat d'ALU comme adresse. Ces signaux sont sources de glitches puisqu'ils lient les opérandes d'instructions à différents étages du pipeline. La contre-mesure contre les glitches liés aux bypass, notamment ceux de la LSU, est l'ajout d'instructions ne manipulant aucune donnée symbolique dans le pipeline, afin de différencier correctement l'état du processeur pour chaque instruction.

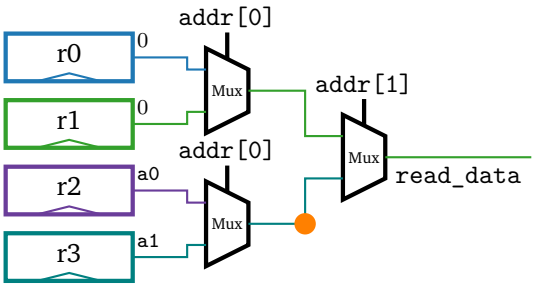
6.1.3 Fuites liées au banc de registres

Le banc de registres est un composant essentiel et central des processeurs. Il est composé d'éléments mémorisants, de largeur égale à un mot, individuellement accessibles en lecture et en écriture. Un banc de registres peut être composé d'un ou plusieurs ports de lecture et d'écriture. Chaque port de lecture est un arbre de multiplexeurs, tel qu'illustré dans la figure 6.13a, permettant la sélection, par éliminations successives, du registre à lire sur le signal du port. Les ports d'écriture - illustrés dans la figure 6.13b - sont composés, pour chaque registre, d'un multiplexeur permettant de choisir si la valeur à écrire est l'ancienne valeur du registre ou la nouvelle valeur présente sur le port d'écriture. Le signal utilisé pour piloter ces multiplexeurs est un `write_enable`, calculé à partir de l'indice du registre de destination de l'instruction.

Pour des raisons de lisibilité, les exemples de cette section sont donnés pour un banc de quatre registres, mais le comportement en présence de plus de registres est identique. Il est important de noter que l'organisation choisie pour ce banc de registres est arbitraire et que d'autres, présentant les mêmes fuites mais pour des valeurs différentes de stabilité, sont aussi possibles pour une même représentation `HDL` (comme illustré dans la figure 5.10).

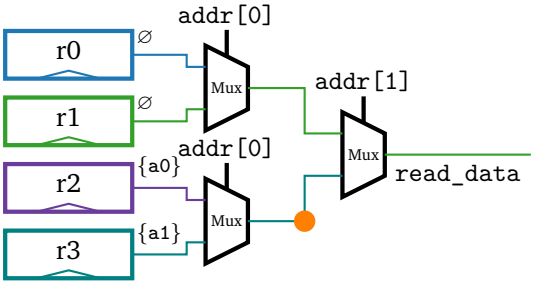


(a) Cycle `t-1`, lecture de `r0`.

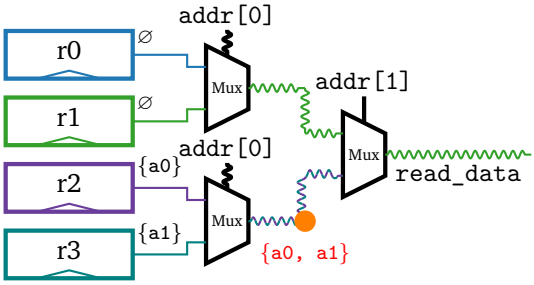


(b) Cycle `t`, lecture de `r1` : transition `a0`→`a1` sur le signal ●.

Fig. 6.14 : `(g=0,t=1) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port de lecture.



(a) Lecture de `r1`, cas avec `addr[0]` stable : aucun glitch ne peut se produire.



(b) Lecture de `r1`, cas avec `addr[0]` instable : potentiel glitch de {`a0`, `a1`} sur le signal ●.

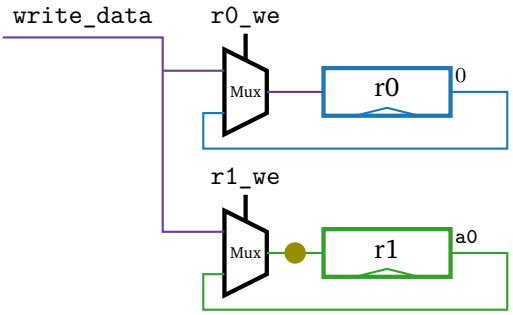
Fig. 6.15 : `(g=1,t=0) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port de lecture.

Banc de registres - Ports de lecture

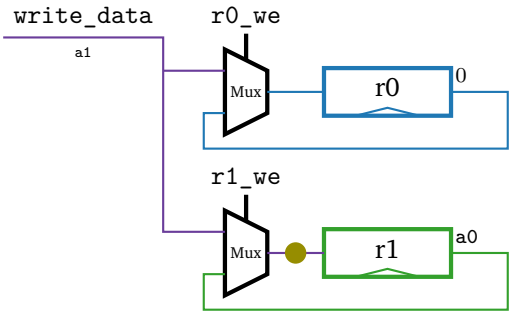
Des fuites peuvent se produire sur les ports de lecture des bancs de registres lorsque des transitions ou des glitches se produisent dans l'arbre de multiplexeurs ou sur le signal du port.

Lors de deux lectures successives, des transitions peuvent avoir lieu sur l'ensemble des multiplexeurs de l'arbre de chaque port de lecture ainsi que sur les signaux des ports de lecture. Ce cas est illustré dans la figure 6.14, dans laquelle le signal ● fuit en transition (`a0`→`a1`).

De même, lors d'une lecture au sein d'un cycle, la stabilité des sélecteurs des multiplexeurs des ports de lecture du banc de registres peut induire des fuites dans ce même arbre de multiplexeurs. La figure 6.15 illustre deux cas possibles, lorsque tous les signaux sont stables ou lorsque tous les signaux sont stables à l'exception d'un des sélecteurs de multiplexeur. Dans ce second cas (figure 6.15b), le signal ● fuit en valeur avec glitches, son ensemble de fuite associé étant {`a0`, `a1`}.

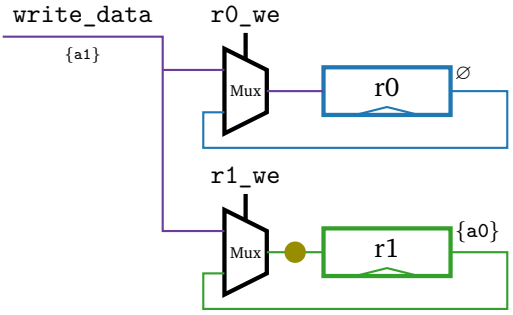


(a) Cycle `t-1`, écriture de n'importe quelle valeur dans `r0`.

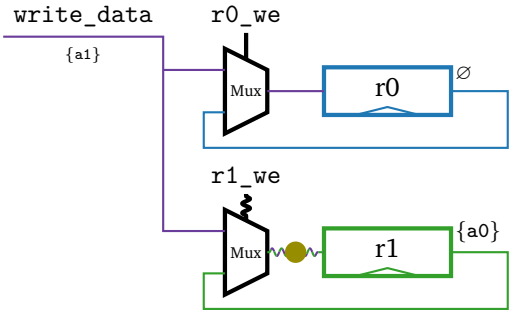


(b) Cycle `t`, écriture de `r1` : transition `a0`→`a1` sur le signal ●.

Fig. 6.16 : `(g=0,t=1) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port d'écriture.



(a) Écriture de `r0`, cas avec `r1_we` stable.



(b) Écriture de `r0`, cas avec `r1_we` instable : potentiel glitch {`a0`, `a1`} sur le signal ●.

Fig. 6.17 : `(g=1,t=0) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port d'écriture.

Banc de registres - Ports d'écriture et registres

Des fuites peuvent se produire sur les ports d'écriture des bancs de registres lorsque des transitions ou des glitches se produisent sur le signal du port ou dans les multiplexeurs réalisant la sélection entre la valeur du port et la valeur actuelle du registre (assimilable au registre).

Lors d'écritures, des transitions peuvent avoir lieu dans les registres ainsi que sur le signal du port d'écriture pour des écritures successives. Le cas d'une transition en registre (qui serait aussi visible au niveau du jeu d'instructions) est illustré dans la figure 6.16, dans laquelle le signal ● réalise la transition `a0`→`a1`.

De même, lors d'une écriture au sein d'un cycle, la stabilité des signaux *write_enable* du banc de registres peut induire des fuites sur les multiplexeurs qu'ils pilotent. La figure 6.17 illustre deux cas possibles, lorsque tous les signaux sont stables ou lorsque tous les signaux sont stables à l'exception d'un des sélecteurs de multiplexeur. Dans ce second cas (figure 6.17b), le signal ● fuit en valeur avec glitches, son ensemble de fuite associé étant {`a0`, `a1`}.

Banc de registres - Contre-mesures pour les ports de lecture et d'écriture

Les fuites sur les ports de lecture et d'écriture, qu'elles soient dues à des glitches ou à des transitions, sont extrêmement communes pendant l'exécution de programmes, compilés ou non. Elles peuvent se produire aussi bien sur les signaux des ports qu'au sein des arbres de multiplexeurs, rendant complexe leur identification sans l'utilisation d'outils tels qu'`aLEAKator`. Lors de mesures réelles, sans connaître précisément l'implantation de l'arbre de multiplexeurs du circuit, il est difficile de différencier les fuites qui relèvent d'une transition de celles qui relèvent d'un glitch.

Différentes méthodes de résolution sont envisageables pour ces deux types de fuites. Pour les fuites en transition sans glitch ayant lieu sur les signaux des ports de lecture ou d'écriture du banc de registres, lire (respectivement écrire) une valeur non sensible en registre empêchera cette transition. Cette contre-mesure est illustrée pour un port de lecture du banc de registres dans la figure 6.18.

Si une fuite en transition a lieu dans l'arbre de multiplexeurs de lecture ou dans un multiplexeur d'écriture, la réaffectation des variables vers d'autres registres est une contre-mesure possible. Il est aussi possible d'ajouter des lectures ou écritures réalisant des nettoyages partiels des arbres de multiplexeurs mais il n'existe aucune garantie qu'une solution existe pour tous les placements de variables.

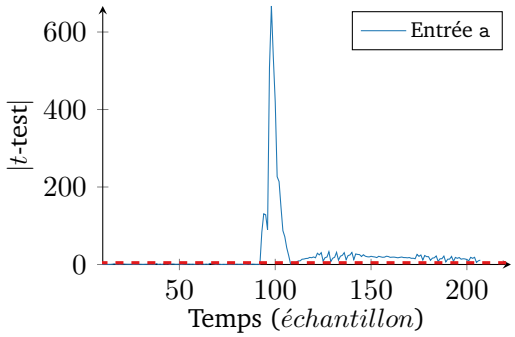
Pour les fuites dues à un glitch, une contre-mesure est de stabiliser les multiplexeurs de lecture ou d'écriture. Cela peut être réalisé en figeant certains champs des instructions ou en réaffectant différemment les variables en registre et en exploitant l'éventuelle stabilité actuelle des instructions.

Une solution alternative pour les glitches et les transitions dans les multiplexeurs de lecture et d'écriture est l'éviction du contenu d'un ou plusieurs registres vers la mémoire ainsi que le nettoyage, par l'écriture d'une valeur non-symbolique, de leurs contenus afin qu'aucun glitch ni transition ne soit possible.

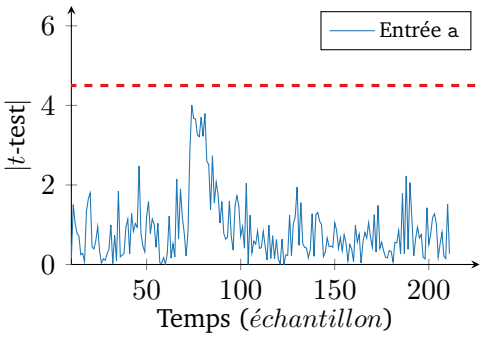
Cette seconde méthode est illustrée pour un port d'écriture dans la figure 6.19, dans laquelle le changement de l'indice du registre écrit par une instruction sans effet sur la fonctionnalité du programme avant l'écriture d'un élément symbolique empêche l'apparition d'une fuite en glitch.

Banc de registres - Bypass

Il est commun pour les processeurs pipelinés d'implémenter des bypass pour le banc de registres. Les cœurs Cortex-M3 et Cortex-M4 ont, par exemple, deux bypass entre le banc



(a) Exécution du listing 6.x, non sécurisé.



(b) Exécution du listing 6.y, sécurisé.

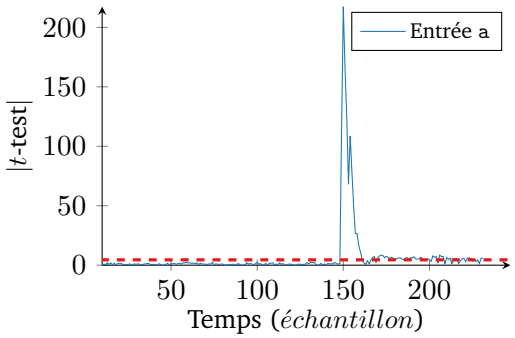
```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r14=s1, r9=blk
6  ldr r0, [r0] // r0<-a0
7
8  ldr r9, [r9] // Nett. bus lecture
9  eor r8, r9, r9 // Nop
10 eor r1, r9, r9 // Nop
11 ldr r1, [r14] // r1<-a1
12 ldr r2, [r9] // Nett. bus lecture
13 eor r2, r2, r9 // Nop (×2)
14
15 and r1, r1, r9
16 // Transition lecture registre
17 // Glitches aussi possibles
18 and r0, r0, r9
```

Listing 6.x : Lecture de registre sans nettoyage préalable: le port de lecture du banc de registres réalise a0→a1.

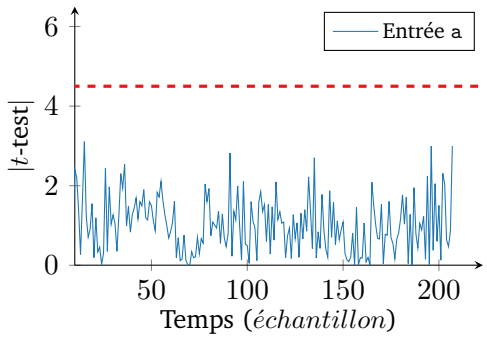
```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r14=s1, r9=blk
6  ldr r0, [r0] // r0<-a0
7
8  ldr r9, [r9] // Nett. bus lecture
9  eor r8, r9, r9 // Nop
10 eor r1, r9, r9 // Nop
11 ldr r1, [r14] // r1<-a1
12 ldr r2, [r9] // Nett. bus lecture
13 eor r2, r2, r9 // Nop (×2)
14
15 and r1, r1, r9
16 // Nett. lecture banc de registre
17 // Glitches toujours possibles
18 and r2, r2, r9
19 and r0, r0, r9
```

Listing 6.y : Lecture de registre avec nettoyage préalable: le port de lecture du banc de registres réalise a0→blk→a1.

Fig. 6.18 : Vecteur démonstratif : fuite en transition lors d'une lecture du banc de registres.



(a) Exécution du listing 6.z, non sécurisé.



(b) Exécution du listing 6.aa, sécurisé.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r14=s1, r9=blk
6  ldr r0, [r0] // r0<-a0
7  ldr r9, [r9] // Nett. bus lecture
8
9  eor r9, r14, r14 // Nop
10 // Sans stabilisation de
11 // l'écriture vers r4
12 eor r9, r14, r14
13 ldr r4, [r14]
14 eor r9, r14, r14
15 eor r9, r14, r14 // Nop
16
17 // Répétition sans lire a1
18 ldr r14, =blk
19 eor r9, r14, r14 // Nop
20 eor r9, r14, r14
21 ldr r5, [r14] // r5<-blk
22 eor r9, r14, r14
23 eor r9, r14, r14 // Nop
```

Listing 6.z : Écriture sans stabilisation préalable de l'adresse du registre à écrire: l'ensemble de fuite du signal d'écriture de `r0` est {`a0`, `a1`} à l'écriture de `r4`.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  blk:.word blk
5  .text // r0=s0, r14=s1, r9=blk
6  ldr r0, [r0] // r0<-a0
7  ldr r9, [r9] // Nett. bus lecture
8
9  eor r9, r14, r14 // Nop
10 // Stabilisation de l'écriture
11 // vers r4
12 eor r4, r14, r14
13 ldr r4, [r14] // r4<-a1
14 eor r4, r14, r14
15 eor r9, r14, r14 // Nop
16
17 // Répétition sans lire a1
18 ldr r14, =blk
19 eor r9, r14, r14 // Nop
20 eor r5, r14, r14
21 ldr r5, [r14] // r5<-blk
22 eor r5, r14, r14
23 eor r9, r14, r14 // Nop
```

Listing 6.aa : Écriture avec stabilisation préalable de l'adresse du registre à écrire: aucun glitch ne peut se produire au cours de l'écriture de `r4`.

Fig. 6.19 : Vecteur démonstratif : fuite en valeur avec glitches lors de l'écriture du banc de registres. L'écriture est réalisée une seconde fois mais en ne manipulant que des valeurs non-symboliques pour mettre en évidence que cette fuite est uniquement due au glitch lors de l'écriture.

de registres en écriture et en lecture pour l'entrée de l'ALU. Cela permet qu'une instruction décodée, dont le registre lu en opérande est écrit par l'instruction actuellement dans l'ALU, soit directement disponible lorsque l'instruction décodée sera transférée vers l'étage d'exécution. Sans ce bypass, un cycle de gel des étages en amont de l'étage dans lequel l'opérande est attendue serait nécessaire pour que la valeur soit écrite puis lue et accessible. Les bypass peuvent causer des glitches liant l'instruction actuellement en cours d'exécution à l'instruction en cours de décodage ou actuellement traitée par la LSU.

La contre-mesure d'ajout d'une instruction ne manipulant que des valeurs non-symboliques peut aussi être utilisée pour émuler un cycle de gel de pipeline.

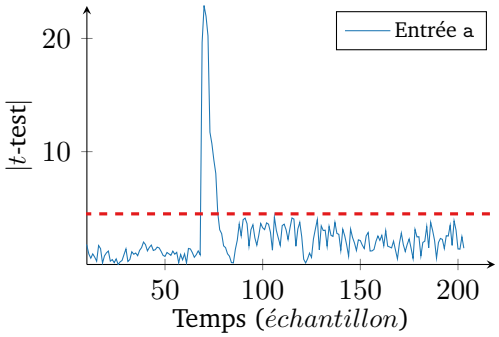
6.1.4 Fuites liées à l'unité arithmétique et logique (ALU)

ALU - Instructions exécutées en un cycle

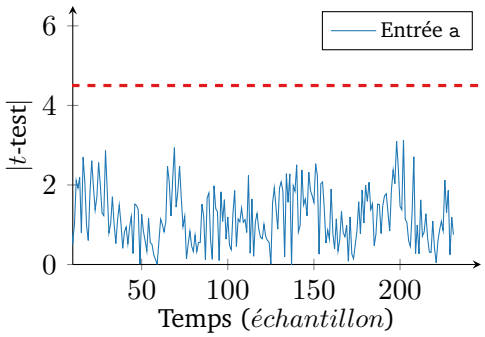
L'ALU peut être source de nombreuses fuites, par exemple lorsque les deux ports en entrée glitchent ensemble ou encore lorsqu'on fait passer `m` via un port d'entrée à un cycle et `k` $\oplus$ `m` via un second port au cycle suivant.

Le programme `Refresh` présenté dans la section 5.4 (Comparaison cycle-à-cycle des verdicts de fuite avec des mesures réelles) est un exemple de fuite des ports d'entrée de l'ALU dans lequel aucune autre fuite n'est présente. La version qui ne nettoie pas les deux ports de l'ALU avant de réaliser le remasquage est donnée dans la figure 6.20a et présente une fuite illustrée dans la figure 6.20a. Cette manière de nettoyer l'ALU n'est pas correcte car le `eor` (ou exclusif, ligne 14) propage la valeur sur les deux ports pendant le nettoyage du registre `r4`. Puis le `mov` (ligne 15) ne nettoie qu'un seul port et finalement le `ldr` (ligne 17) nettoie ce même port en plus de charger la valeur, ce qui est à l'origine de la fuite. Les différents bypass des processeurs peuvent aussi être sources de glitches et de transitions qui se manifestent dans l'ALU.

Le nettoyage des ports de l'ALU peut être réalisé finement, comme dans le listing 6.ac, ou plus grossièrement avec l'ajout d'instructions ALU ne manipulant que des opérandes non-symboliques. Le nettoyage effectué ici consiste à convertir l'instruction `ldr` qui était encodée sur 16 bits en sa variante 32 bits qui nettoie l'autre port de l'ALU. L'efficacité du nettoyage est illustrée dans la figure 6.20b. Lorsque les fuites sont dues aux bypass, l'ajout d'instructions émulant des gels de pipeline ou le réordonnancement des instructions est nécessaire.



(a) Exécution du listing 6.ab, non sécurisé.



(b) Exécution du listing 6.ac, sécurisé.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  m0:.word 0 // m0
5  blk:.word blk
6  .text// r0=s0, r1=s1, r2=m, r3=blk
7  ldr r4, [r0] // r4<-a0
8  ldr r5, [r2] // r5<-m0
9
10 eor r4, r4, r5 // r4<-m0^a0
11 str r4, [r0] // s0<-m0^a0
12
13 str r3, [r3] // Nett. bus écriture
14 eor r4, r4, r4 // Mauvais nett.
15 mov r4, 0x0 // Nett. port A
16
17 ldr r6, [r1] // r6<-a1 (16 bits)
18
19 eor r6, r6, r5 // r6<-a1^m0
20 str r6, [r2] // s1<-a1^m0
```

Listing 6.ab : Remasquage avec ports de l'ALU incorrectement nettoyés.

```
1  .data
2  s0:.word 0 // a0
3  s1:.word 0 // a1
4  m0:.word 0 // m0
5  blk:.word blk
6  .text// r0=s0, r1=s1, r2=m, r3=blk
7  ldr r4, [r0] // r4<-a0
8  ldr r5, [r2] // r5<-m0
9
10 eor r4, r4, r5 // r4<-m0^a0
11 str r4, [r0] // s0<-m0^a0
12
13 str r3, [r3] // Nett. bus écriture
14 eor r4, r4, r4 // Mauvais nett.
15 mov r4, 0x0 // Nett. port A
16
17 ldr.w r6, [r1] // r6<-a1 (32 bits)
18
19 eor r6, r6, r5 // r6<-a1^m0
20 str r6, [r2] // s1<-a1^m0
```

Listing 6.ac : Remasquage avec ports de l'ALU correctement nettoyés.

Fig. 6.20 : Vecteur démonstratif : fuite en valeur avec glitches lors d'une opération ALU.

ALU - Instruction exécutées en plusieurs cycles

Les opérations multi-cycles sont implémentées à l'aide de registres qui servent d'accumulateurs ; ceux-ci peuvent donc naturellement être sources de fuite.

Les processeurs qui utilisent le gating des entrées des opérateurs comme le multiplicateur ne présentent généralement pas ces fuites, mais les processeurs plus simples n'en sont pas protégés.

Le nettoyage des accumulateurs peut être réalisé en utilisant les opérateurs avec des valeurs non-symboliques.

6.1.5 Instructions réalisant de multiples opérations

Des instructions réalisant de multiples opérations « à la fois » existent et peuvent être sources de fuites complexes à corriger.

Un tel exemple pour les Cortex-M3 et Cortex-M4 est une utilisation problématique de `ldm r0, {r1, r2}` pour lire depuis la mémoire deux shares qui composent un secret dans les registres `r1` et `r2` aux adresses `r0` et `r0+4` respectivement. Puisque cette instruction cause une fuite en transition sans glitch (et une fuite en valeur avec glitches) sur le bus de lecture de la mémoire, aucune instruction avant ou après ne permet d'empêcher celle-ci, sauf à modifier en amont la mémoire, ce qui serait une tâche complexe. Une solution plus réaliste est la réécriture de cette instruction (et de toutes les opérations mémoire multiples `ldrd`, `strd`, `ldm`, `stm`, `pop` et `push`) avec un groupe d'instructions fonctionnement équivalent.

Les contre-mesures précédemment détaillées peuvent ensuite être appliquées pour éliminer les fuites dans ces opérations mémoire consécutives.

6.2 Système expert pour l'élimination automatique de fuites

La section précédente a détaillé différentes sources de fuite pour lesquelles des contre-mesures associées ont été proposées. Cette section est dédiée à l'élimination automatique des fuites de programmes exécutés sur des processeurs en exploitant les informations rendues disponibles par `aLEAKator`. Dans un premier temps, les hypothèses sur les programmes pouvant être sécurisés par cette méthode sont détaillées. Une stratégie de sécurisation est ensuite proposée et les contre-mesures pouvant être appliquées dans le cadre de cette stratégie sont définies. Enfin, la logique d'application de ces contre-mesures est détaillée

pour composer le système expert réalisant l'élimination des fuites dans le modèle `RR 1-sw-probing` des programmes exécutés sur le processeur ARM Cortex-M4. Cette section présente l'état des travaux menés jusqu'à la rédaction de ce manuscrit.

6.2.1 Hypothèses de sécurisation

Les programmes sur lesquels l'élimination automatique des fuites est appliquée doivent respecter les hypothèses présentées dans cette sous-section.

Sécurisation en valeur sans glitch

L'élimination automatique des fuites en valeur sans glitch est un problème complexe, hors du cadre de cette étude. Si le programme n'est pas masqué, l'utilisation de méthodes présentées dans la section 3.5 est une première étape à réaliser. Si le masquage est incorrectement implémenté, une étape de sécurisation manuelle préliminaire, éventuellement aidée par `aLEAKator`, doit être réalisée. Les contre-mesures à appliquer dans ce cas sont détaillées dans la sous-section 6.1.1. L'approche suppose donc que les programmes en entrée ne présentent aucune fuite dans le modèle `1-sw-probing`.

Flot de contrôle

Les programmes doivent être exempts de boucle, d'appel de fonction/macro et de directive préprocesseur. Une compilation *inlinant*[2] tous les appels de fonctions/les macros, exécutant le préprocesseur et déroulant toutes les boucles est nécessaire. Cela n'est pas une contrainte définitive ; un système expert plus précis et exploitant mieux la structure du programme pourrait corriger des programmes sans limitations sur le flot de contrôle. Cette hypothèse entraîne une augmentation de la taille des programmes en entrée. Il ne s'agit pas d'une limitation définitive et la suppression de cette hypothèse est discutée dans les perspectives à la fin de ce chapitre.

Instructions réalisant de multiples opérations

Les instructions identifiées dans la sous-section 6.1.5 qui réalisent plusieurs opérations (notamment pour ARM : `ldm`, `pop`, `strd`, etc.) doivent être remplacées par plusieurs

2. *Inliner* une fonction consiste à remplacer les instructions d'appel à cette fonction par l'ensemble de ses instructions.

instructions émulant leur fonctionnement. Cela pourra, à terme, être automatisé par `aLEAKator`.

6.2.2 Stratégie de sécurisation

De nombreuses sources de fuite et contre-mesures ont été présentées dans la section 6.1. Cependant, dans un objectif d'élimination automatique des fuites, certaines de ces contre-mesures sont complexes à mettre en œuvre. Cette sous-section est dédiée à la définition d'une stratégie de sécurisation et à la définition de propriétés qui devront être respectées par les contre-mesures applicables par le système expert.

Propriété de sécurisation

L'élimination automatique des fuites consiste en la modification d'un programme initial `Assembleur` à l'aide d'informations obtenues sur les fuites identifiées lors de l'exécution du binaire résultant de la compilation de ce programme. L'hypothèse sur la linéarité du flot de contrôle introduite dans la sous-section 6.2.1 implique que les instructions du programme `Assembleur` ne sont exécutées qu'une seule fois. Cette unicité d'exécution permet de garantir que lorsqu'une contre-mesure doit être appliquée à partir d'une instruction, il est suffisant de s'assurer qu'elle n'altère pas la fonctionnalité du programme en ce point. Autrement, il serait nécessaire d'assurer que pour tous les chemins d'exécution passant par les instructions ajoutées pour la contre-mesure, la fonctionnalité du programme n'est pas altérée. Par exemple, une contre-mesure insérée dans une boucle serait exécutée de multiples fois, les instructions ajoutées pourraient être adaptées à la première exécution du corps de boucle mais pas aux autres.

Soit un programme `Assembleur` P respectant les hypothèses de la sous-section 6.2.1 et dont l'exécution est constituée de n instructions ordonnées $i_0 \dots i_{n-1}$. Cette séquence d'instructions est nommée *trace d'exécution* par la suite. Pour appliquer une contre-mesure à partir d'une fuite identifiée dans le binaire, une association doit exister entre une instruction `Assembleur` initiale et une ou plusieurs (pour le cas des pseudo-instructions[3]) instructions dans le binaire exécuté. Cette association permet d'établir un lien entre une instruction implantée et la ligne du programme `Assembleur` à l'origine de cette instruction.

Les transformations théoriquement applicables sur ce programme P sont l'*ajout*, la *modification* et la *suppression* d'instructions. Cependant, nous limitons les contre-mesures à celles respectant les propriétés P_1, P_2, P_3 et P_4 présentées ci-après. Cette limitation est réalisée

3. Les pseudo-instructions sont des instructions `Assembleur` sources valides mais dont l'assemblage est réalisé avec plusieurs instructions. Pour le jeu d'instructions RISC-V, `la` est un exemple de pseudo-instruction.

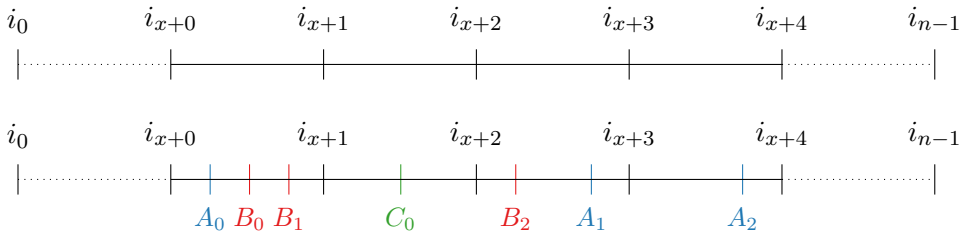


Fig. 6.21 : Représentation du programme P lors de l'ajout de trois contre-mesures d'exemple A, B et C. Les instructions des contre-mesures sont nommées respectivement, avec $i \in [0, 2]$: $A_0 \dots A_i$, $B_0 \dots B_i$ et C_0.

dans l'objectif d'obtenir une propriété de non-dégradation garantissant, lors de l'ajout de contre-mesures pour éliminer la première fuite identifiée dans une exécution à l'instruction i_j, qu'aucune nouvelle fuite ne se produira sur les instructions $i_0 \dots i_{j-1}$.

(P_1) Les instructions du programme initial sont conservées et leur ordre n'est pas modifié. Les contre-mesures ne sont donc réalisées qu'avec des *ajouts* d'instructions.

Les instructions ajoutées par les contre-mesures font partie du programme mais ne modifient pas l'ordre initial $i_0 \dots i_{n-1}$ et ne sont pas comptabilisées dans le nombre d'instructions du programme exécuté.

(P_2) Chaque contre-mesure appliquée pour éliminer la fuite d'une instruction i_j est composée d'une ou plusieurs instructions qui peuvent être placées librement entre l'instruction i_{j-1} et l'instruction i_{n-1}.

Exemple 16. La figure 6.21 illustre un programme lors de l'application de contre-mesures et clarifie que l'ordre des instructions initiales du programme est conservé. Les contre-mesures A et B sont appliquées pour éliminer des fuites liées à l'instruction i_{x+1} et la contre-mesure C pour des fuites liées à l'instruction i_{x+2}.

(P_3) Les instructions des contre-mesures ajoutées entre les instructions i_{j-1} et i_j pour corriger l'instruction i_j ne peuvent pas charger de nouvelles valeurs symboliques dans les éléments mémorisants du cœur, mais uniquement en supprimer de l'état courant du processeur ou en renvoyer vers la mémoire. Cette contrainte ne s'applique pas aux instructions ajoutées après i_j pour cette même contre-mesure.

(P_4) Les instructions ajoutées avant l'instruction responsable d'une fuite ne créent pas de nouvelles fuites mais peuvent ne pas totalement supprimer la fuite de l'instruction ciblée.

Cette dernière propriété indique qu'il n'existe pas de garantie qu'une solution soit trouvée à chaque source de fuite. En pratique, pour éviter d'ajouter indéfiniment des instructions au programme qui ne résoudront pas la fuite, des conditions de terminaison ont été établies et sont détaillées dans la sous-section 6.2.4.

Algorithme 4 : Stratégie d'élimination itérative des fuites d'un programme `Assembleur`.

Entrée : prog_source : le programme `Assembleur` à sécuriser dans le modèle `RR 1-sw-probing`
Sorties : succès : indique si la sécurisation s'est terminée avec succès
programme : le programme de la dernière itération, sécurisé si succès est vrai

```
Fonction sécuriser_programme(prog_source) :
    prog_binaire = compiler_programme (prog_source)
    // Générer la trace d'exécution complète et confirmer que le programme respecte les
       hypothèses, notamment l'absence de fuite en valeur sans glitch.
    trace, sécurisé_valeur = simuler_et_vérifier(prog_binaire, 1-sw-probing)
    si ! sécurisé_valeur alors
        // Abandon
        retourner { succès : Faux, programme : prog_source }
    // Tant que le programme n'est pas sécurisé et qu'un correctif est proposé pour
       l'itération
    tant que Vrai faire
        // La fonction système_expert réalise la simulation de prog_binaire jusqu'à ce
           qu'une règle du système expert s'applique ou que la simulation termine sans
           rencontrer de fuite dans le modèle RR 1-sw-probing
        sécurisé, correctif = système_expert(prog_binaire, trace)
        si sécurisé alors
            // Le programme est désormais sécurisé
            retourner { succès : Vrai, programme : prog_source }
        sinon
            si correctif == ∅ alors
                // Aucun correctif pertinent n'a été trouvée, abandon
                retourner { succès : Faux, programme : prog_source }
            prog_source = appliquer_correctif(prog_source, correctif)
            prog_binaire = compiler_programme(prog_source)
```

Le procédé de sécurisation suit l'algorithme 4. Ce procédé est itératif et basé sur un système expert présenté dans la sous-section 6.2.4. Le programme P est simulé par le système expert qui utilise `aLEAKator` et qui propose un ensemble d'instructions à appliquer pour corriger une instruction i_j identifiée comme la source d'une fuite. Ce correctif est appliqué et le programme est recompilé et simulé à nouveau depuis le début[4], jusqu'à ce qu'il n'y ait plus de fuites dans le modèle `RR 1-sw-probing` ou que la sécurisation échoue, faute de correctif à appliquer. Par construction avec les propriétés $P_{1...4}$, lorsqu'une fuite est

4. La simulation est réalisée depuis le début du programme mais la vérification, elle, en raison des propriétés $P_{1...4}$ peut n'être réalisée qu'à partir de l'instruction précédent la fuite de l'itération précédente.

identifiée pour une instruction i_j, l'exécution des instructions $i_0 \dots i_{j-1}$ ne présente pas de fuite. L'ajout d'un correctif respectant les propriétés énumérées pourra soit ne pas totalement corriger la fuite, soit corriger la fuite et sécuriser l'exécution de l'instruction i_j. Sous l'hypothèse que les contre-mesures permettent de résoudre les fuites rencontrées, ce qui est étudié par la suite, la sécurisation repousse donc l'apparition de la première fuite du programme (relativement aux instructions initiales) jusqu'à ce que la dernière instruction soit sécurisée.

Propriété sur le banc de registres

La sécurisation en présence de glitches du banc de lecture des registres est complexe. L'implantation réelle de l'ordre de sélection dans l'arbre de multiplexeurs étant fixée au moment de la synthèse, il n'est pas possible de déterminer dans quel ordre les différents registres seront sélectionnés avant cette étape. Il existe deux solutions pour que le banc de registres ne présente pas de fuites en présence de glitches :

1. Ne pas conserver simultanément des éléments permettant de recomposer un secret.
2. Exploiter la stabilité pour que les éléments permettant de recomposer un secret ne se rencontrent jamais dans les arbres de multiplexeurs.

La première solution a été sélectionnée pour deux raisons. Premièrement, la seconde solution nécessiterait d'établir une méthode de détection automatique de l'organisation des multiplexeurs et les contre-mesures à appliquer devraient elles-mêmes prendre en compte cette organisation. Deuxièmement, exploiter la stabilité nécessite de choisir précisément quels bits des instructions (encodées) doivent être figés selon le contenu des registres à chaque cycle. En plus d'être complexe et contraire à P_1 (non réalisable sans modifier des instructions du programme, initiales ou correctives), il n'existe pas de garantie que cette solution soit applicable dans tous les cas. Pour illustrer cette complexité, un cas d'étude pour le jeu d'instructions RISC-V et appliqué à l'Ibex est donné dans l'annexe C. Pour ces raisons, la sécurisation est réalisée en garantissant que tous les éléments sauvegardés en registre ne présentent aucune fuite lorsqu'ils sont vérifiés conjointement.

Spécificités techniques

Pour réaliser la stratégie de l'algorithme 4, des difficultés techniques doivent être résolues.

Premièrement, l'ajout d'instructions dans le programme `Assembleur` se fait relativement à une instruction du programme binaire compilé qu'`aLEAKator` identifie comme responsable d'une fuite. Pour identifier à quelle ligne du programme `Assembleur` source appliquer les contre-mesures, l'outil `labeller` a été développé. Cet outil ajoute une étiquette unique à

chaque instruction du programme. Ces étiquettes sont intégrées au binaire dans la table des symboles lors de la compilation. Ainsi, lorsqu'une fuite est identifiée, `aLEAKator` détermine quelle étiquette pointe vers l'instruction source de la fuite à l'aide du PC. Cela permet d'identifier la ligne `Assembleur` à partir de laquelle ajouter des instructions. Les contre-mesures ajoutées sont elles-mêmes étiquetées, ce qui permet ensuite d'appliquer, au besoin, d'autres contre-mesures relativement à leur position.

Deuxièmement, les contre-mesures développées doivent régulièrement faire usage d'un registre qui contient une valeur concrète arbitraire non-symbolique pour réaliser des nettoyages de sous-systèmes du processeur. Afin de limiter le coût en performance et pour ne pas avoir à modifier le programme, un registre est dédié pour les contre-mesures. Cela évite d'effectuer des sauvegardes/restaurations mémoire dès qu'un registre est nécessaire. Une zone mémoire doit aussi être allouée pour permettre 1) d'évincer le contenu des registres en les sauvegardant en mémoire et en les restaurant plus tard dans le programme ; 2) de réaliser des accès mémoire (éventuellement non-alignés) non-symboliques. Toujours dans un objectif de performance, le registre dédié est un pointeur vers le premier mot mémoire de la zone réservée et ce premier mot mémoire est, tout au long du programme, un pointeur vers lui-même. Les contre-mesures exploitent les propriétés sur la valeur du registre dédié et du premier mot de la zone mémoire réservée et les conservent. Les autres mots mémoire de la zone réservée conservent temporairement le contenu des registres évincés. L'option `-ffixed-r9` permet de compiler un programme `C` pour ARM en réservant le registre `r9`. Toujours pour l'architecture ARM, une telle zone mémoire de 16 mots de 32 bits peut être allouée en `Assembleur` dans une section `.data` avec `aleakator: .word aleakator ; .space 60` et de la manière suivante en `C` : `uint32_t aleakator[16] = {(uint32_t)&aleakator, 0};`. La taille des symboles étant accessible par le système expert qui applique les contre-mesures, celui-ci est capable de détecter un éventuel dépassement et d'indiquer que cette zone mémoire doit être agrandie pour sécuriser le programme.

Enfin, certaines contre-mesures nécessitent de savoir quels registres sont lus/écrits par les instructions du programme ainsi que si un ensemble arbitraire de signaux fuit lorsqu'ils sont vérifiés conjointement dans un modèle de fuite donné. Pour déterminer les registres lus/écrits par les instructions, une première exécution du programme est réalisée par `aLEAKator` pour générer une trace d'exécution en extrayant du pipeline les instructions binaires exécutées puis en exploitant l'outil de désassemblage `Capstone`[5] [Con13]. Une telle trace contient les informations annotées dans le listing 6.ad sous un format facilement exploitable par `aLEAKator`.

5. Disponible sur github : `https://github.com/capstone-engine/capstone`

```
1 uxtb r4, r4         // Reads: r4      // Writes r4
2 str  r0, [r2, #36] // Reads: r0,r2
3 ldrb r0, [r3, #78] // Reads: r3      // Writes r0
4 eor  r1, r1, r10   // Reads: r1,r10 // Writes r1
```

Listing 6.ad : Exemple de programme annoté avec les registres lus et écrits par les instructions. Ces informations sont associées à chaque étiquette d'instruction pour composer la trace d'exécution.

6.2.3 Conception de correctifs adaptés à une application automatique

Nous appelons *correctifs* les implémentations des contre-mesures pour un processeur donné. Ces implémentations respectent les propriétés $P_{1...4}$ établies dans la sous-section 6.2.2. Les correctifs décrits dans cette section sont créés pour le processeur ARM Cortex-M4. Des variantes RISC-V similaires ont été développées pour le processeur Ibex et sont données dans l'annexe D.

Correctif pour les registres

La contre-mesure d'éviction du contenu de registres permet la suppression des fuites liées au banc de registres. Elle nécessite à la fois des informations extraites de la trace d'exécution du programme et des informations extraites de l'état du processeur.

Une méthode pour choisir quels contenus de registre évincer est fournie dans l'algorithme 5. Les entrées de cet algorithme sont :

1. L'ensemble des registres du processeur ayant une valeur symbolique (il ne serait pas pertinent d'évincer le contenu d'un registre qui ne fait pas intervenir de valeur sensible).
2. Une liste de registres à conserver obligatoirement ; ces registres sont ceux qui sont lus par l'instruction courante.
3. Le modèle de fuite dans lequel effectuer les vérifications.
4. Un ensemble de signaux à vérifier conjointement avec le banc de registres.

Cet algorithme retourne un ensemble de registres dont le contenu doit être déplacé vers la mémoire en amont de l'instruction courante pour que le banc de registres vérifié conjointement avec l'ensemble de signaux additionnels d'entrée dans le modèle de fuite donné en entrée soit sécurisé. Cette liste de registres est déterminée en trouvant un ensemble de registres éligibles, le plus petit possible, dont l'éviction du contenu élimine la fuite lors de la vérification conjointe du banc de registres et des signaux additionnels.

En suivant l'algorithme 5, un ou plusieurs registres sont sélectionnés pour être évincés. Pour chaque éviction, les étapes suivantes sont appliquées :

1. Analyse de vivacité du registre à partir de la trace d'exécution et si le registre est vivant :
   a) Une zone mémoire dans l'espace alloué pour le symbole `aleakator` est identifiée pour pouvoir restaurer le registre lorsque cela sera nécessaire.
   b) Si la zone mémoire choisie avait déjà été utilisée pour sauvegarder un autre registre précédemment (depuis restauré) : nettoyage de la cellule mémoire par l'écriture d'une valeur non-symbolique.
   c) Sauvegarde du contenu du registre dans la zone mémoire.
   d) Ajout de l'instruction de rechargement avant la prochaine instruction qui utilise comme source le registre évincé.
2. Nettoyage du registre.
3. Si la prochaine instruction est un rechargement : nettoyage du bus d'écriture de la mémoire, pour éviter une transition éventuelle.

Cette stratégie permet, lorsqu'une instruction fuit dans la logique de lecture du banc de registres, de revenir à un état du banc de registres permettant l'exécution de cette instruction sans fuite dans la logique de lecture du banc de registres.

Ce correctif est noté éviction du contenu d'un ou plusieurs registres par la suite.

Correctifs pour la mémoire

Les fuites du bus de lecture mémoire peuvent être corrigées en ajoutant une lecture d'une valeur non-symbolique avant la lecture causant une fuite. Si la source de la fuite est liée au registre interne réalisant les lectures non-alignées qui avait été mentionné dans la section 6.1 (Fuites liées à la mémoire - Lectures non-alignées), la lecture non-symbolique doit, elle aussi, être non-alignée. Aucune information sur l'état actuel du processeur ou sur la trace d'exécution n'est nécessaire pour corriger ces fuites. Identiquement, les fuites liées aux écritures sur le bus ou dans le registre interne peuvent être corrigées par des écritures non-symboliques (éventuellement non-alignées) et ne nécessitent aucune information sur l'état du processeur. Pour les opérations mémoire alignées, les instructions `ldr r9, [r9]` et `str r9, [r9]` réalisent le nettoyage du bus de lecture ou d'écriture sans altération de `r9`. Pour les accès non-alignés, il peut être nécessaire d'altérer la zone mémoire contenant le pointeur comme illustré dans le listing 6.ae ou de réserver deux mots mémoire supplémentaires.

Algorithme 5 : Stratégie de sélection du ou des contenus de registres à évincer pour que le banc de registres et l'ensemble de signaux additionnels ne fuient plus. La méthode `sécurisé` réalise la vérification conjointe, dans un modèle de fuite donné, d'un ensemble de signaux et retourne vrai en l'absence de fuite.

Entrées : registres : l'ensemble des registres dont le contenu est symbolique
conserver : l'ensemble des registres qui ne peuvent pas être évincés
modèle_fuite le modèle de fuite dans lequel effectuer les vérifications
signaux_supp : un ensemble de signaux à ajouter à la vérification

Sorties : ok : un booléen représentant le succès de la fonction
registres_à_évincer : un ensemble de registres dont le contenu doit être évincé pour que la fuite issue de la vérification conjointe du banc de registres et des signaux additionnels ne se produise plus

```
// Il faut que la propriété de sécurité ne soit pas déjà respectée pour que le recours à l'éviction soit pertinent
```

Précondition : ! `sécurisé`(modèle_fuite, registres ∪ signaux_supp)

```
// Il faut que les registres à conserver+signaux supplémentaires soient sécurisés sans quoi aucune éviction possible ne pourra empêcher la fuite
```

Précondition : `sécurisé`(modèle_fuite, conserver ∪ signaux_supp)

Fonction `contenu_à_évincer`(registres, conserver, modèle_fuite, signaux_supp) **:**

```
// Prioriser les évictions (parmi les registres qui ne sont pas dans conserver) qui font intervenir le moins de registres possible
pour taille_uplet = 1 à |registres\conserver| faire
    // Toutes les combinaisons de taille taille_uplet de registres (hors registres à conserver)
    combinaisons_possibles = combinaisons(taille_uplet, registres\conserver)
    pour chaque essai_éviction ∈ combinaisons_possibles faire
        // Est-ce qu'évincer les registres de la combinaison proposée permet d'empêcher la fuite ?
        si sécurisé(modèle_fuite, registres\essai_éviction ∪ signaux_supp) alors
            // Proposition d'éviction trouvée
            retourner { ok : Vrai, registres_à_évincer : essai_éviction }
// Aucune solution trouvée, évincer tous les registres (hors ceux à conserver) ne suffirait pas à empêcher la fuite
retourner { ok : Faux, registres_à_évincer : ∅ }
```

```
1 // Écriture non-alignée d'un demi-mot
2 strh.w  r9, [r9, #1]
3 // Restauration du premier mot de la zone mémoire réservée
4 movw    r9, :lower16:aleakator
5 movt    r9, :upper16:aleakator
6 str.w   r9, [r9]
```

Listing 6.ae : Exemple de correctif pour nettoyer le chemin des écritures non-alignées.

Les fuites liées à la réécriture de cellules mémoire ne peuvent pas être corrigées ainsi. Il est nécessaire d'identifier quelle zone mémoire est affectée et d'y écrire une valeur non-symbolique sans altérer les autres octets qui composent le mot mémoire et qui ne sont pas écrits. Cette identification peut être réalisée à partir de l'état du processeur, en récupérant le signal de largeur d'écriture transmis à la mémoire ou à partir de la trace d'exécution, en reproduisant l'instruction d'écriture avec un registre source différent. Pour les jeux d'instructions complexes comme le `ARMv7-M`, il est nécessaire de prendre en compte les pré/post-incréments qui modifient l'adresse avant ou après l'écriture pour ne pas fausser l'exécution subséquente.

Ces correctifs sont notés respectivement nettoyage bus mémoire lecture, nettoyage registre interne mémoire et nettoyage cellule mémoire par la suite.

Correctifs pour le pipeline et l'ALU

Le nettoyage des entrées de l'ALU peut être réalisé finement (comme dans l'exemple sur la correction du programme `refresh` de la figure 6.20) mais nécessite des informations avancées sur l'état du processeur et son architecture. De manière plus générique, l'ajout d'instructions exécutées en un cycle et ne manipulant que des opérandes non-symboliques est une solution simple, qui ne présente aucun surcoût et qui réalise un nettoyage plus large. Cette solution fonctionne aussi pour l'ajout d'instructions émulant des gels du pipeline. Par exemple, l'instruction `and r9,r9, r9` est garantie de ne manipuler que des valeurs non-symboliques, de nettoyer les deux ports de l'ALU et de conserver la valeur du pointeur dans le registre `r9` dédié aux correctifs.

Bien que les implémentations de ces deux correctifs soient identiques, ces correctifs sont désignés par la suite par deux noms différents pour plus de clarté sur le besoin spécifique lors de l'élimination d'une fuite : émulation gel du pipeline et nettoyage ALU.

Exploitation des modèles de fuite pour guider le choix des correctifs ainsi que leur implémentation

Pour appliquer la stratégie proposée dans cette section, qui vise à sécuriser des logiciels dans le modèle `RR 1-sw-probing`, il est nécessaire que le système expert puisse effectuer des vérifications dans des modèles de fuite alternatifs.

Exemple 17. La figure 6.22 illustre une situation dans laquelle il est impossible de réaliser une éviction de contenu de registre sans recourir à une vérification qui ne considère pas les glitches. Le port d'écriture fuit lorsque les glitches sont considérés et par conséquent

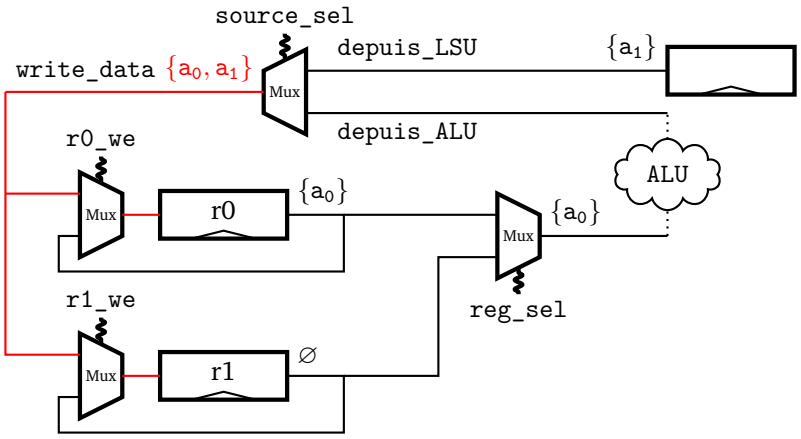


Fig. 6.22 : Vision fortement simplifiée du processeur ARM Cortex-M4 dans un état pour lequel une éviction directement menée avec des vérifications dans le modèle `RR 1-sw-probing` échouerait.

les registres `r0` et `r1` aussi. Pour réaliser correctement l'éviction dans cette situation, une vérification en valeur sans glitch du signal en provenance de la LSU avec seulement `r0` (fuite) puis avec seulement `r1` (absence de fuite) est nécessaire. En effet, si la valeur du bus doit être maintenue (par exemple parce que le cycle courant est une lecture depuis la mémoire), le contenu du registre `r1` doit être évincé en amont de l'instruction courante. Dans le modèle `RR 1-sw-probing` les vérifications individuelles de `r0` et `r1` mènent toutes deux à des verdicts non-sécurisés (en raison de `write_data`) et pourraient amener à la conclusion erronée qu'aucune éviction suffirait à empêcher cette fuite.

La détection précoce de situations menant à des fuites complexes à résoudre est une justification de la réalisation de vérifications supplémentaires à celles qui seraient normalement réalisées pour une vérification dans le modèle `RR 1-sw-probing`. Cela permet notamment de garantir la propriété définie dans la section courante (Propriété sur le banc de registres) sur le banc de registres à chaque cycle. Ces vérifications, qui peuvent être réalisées dans des modèles de fuite alternatifs au `RR 1-sw-probing`, guident aussi les choix réalisés pour l'application des correctifs, comme illustré dans cette sous-section.

6.2.4 Organisation des applications de correctif

Les correctifs définis dans la sous-section 6.2.3 sont les seules transformations applicables sur le programme pour cette sous-section. Le système expert doit appliquer les différents correctifs pour mener à bien l'élimination des fuites en fonction des informations extraites de l'état du processeur en fonctionnement et de la trace d'exécution du programme. Le système expert qui réalise l'élimination des fuites est appelé par l'algorithme 4 et est implémenté dans l'algorithme 6. Les différents correctifs présentés jusqu'à présent sont

associés à des prédicats qui définissent leurs conditions d'application et qui sont rassemblés dans une *base de règles* utilisée par le système expert. L'algorithme 6 réalise la simulation du programme binaire depuis le début en essayant d'appliquer les différentes règles à chaque cycle et en vérifiant la présence de fuite dans le modèle `RR 1-sw-probing`. La fonction prend fin dès qu'une règle s'applique. Cependant, si une fuite dans le modèle `RR 1-sw-probing` est identifiée et qu'aucune règle ne s'applique, l'élimination automatique des fuites échoue.

Algorithme 6 : Implémentation du système expert. La base de règles contient des règles qui sont composées d'une condition d'application appelée prédicat et d'une action. Les règles sont parcourues dans l'ordre et la première dont le prédicat est vrai s'applique en mettant fin à la fonction.

Entrées : prog_binaire : le programme binaire à exécuter
trace : la trace d'exécution complète du programme
Sorties : sécurisé : indique si la simulation complète a été réalisée sans identifier de fuite
correctif : une séquence d'instructions à ajouter au programme source

```
Fonction système_expert(prog_binaire, trace) :
    // Exécution de prog_binaire jusqu'à la première fuite ou jusqu'à ce qu'une règle
       s'applique
    pour cycle = 0 à dernier_cycle faire
        // Simulation et vérification du prochain cycle du programme
        état_simulation = simuler(prog_binaire, cycle)
        // Chaque règle est composée d'un prédicat et d'une action.
        pour chaque règle ∈ règles faire
            // Si cette règle s'applique
            si règle.prédicat(état_simulation, trace) == Vrai alors
                // Déterminer la séquence d'instructions qui implémente l'action de la
                   règle pour résoudre la fuite
                instructions = formater_correctif (règle.action, trace, état_simulation)
                retourner { sécurisé : Faux, correctif : instructions }
        // Si le cycle n'est pas sécurisé dans le RR 1-sw-probing et qu'aucune règle ne
           peut s'appliquer, abandonner l'élimination automatique des fuites
        cycle_est_sécurisé = est_sécurisé(état_simulation, RR 1-sw-probing)
        si !cycle_est_sécurisé alors
            retourner { sécurisé : Faux, correctif : ∅ }
        // Sinon, répéter ces opération sur le prochain cycle
    // La simulation complète s'est déroulée sans fuite
    retourner { sécurisé : Vrai, correctif : ∅ }
```

Les correctifs disponibles n'ont pas de garantie d'être exhaustifs, il est donc théoriquement possible, comme pris en compte lors de la présentation de l'algorithme 4 dans la section courante (Propriété de sécurisation), qu'aucune solution ne soit applicable par le système expert. Pour détecter ces cas et ne pas appliquer infiniment des correctifs inefficaces, deux

Signal	Nom
Registre `i`	$\texttt{r}_\texttt{i}$
Bus de lecture mémoire	`read_bus`
Bus d'adresse mémoire	`read_addr`
Registre interne LSU	`lsu_reg`
Cellule mémoire	`mem_cell`
Opération mémoire pipelinée dans le cycle	`lsu_pipe_advance`
Bypass mémoire→banc de registres	`fwd_lsu_reg`
Bypass mémoire→ALU	`fwd_lsu_alu`

Tab. 6.1 : Récapitulatif des signaux nommés dans le système expert.

mécanismes sont implémentés : les correctifs nettoyage bus mémoire lecture et émulation gel du pipeline sont appliqués au plus une fois pour corriger une unique instruction. De plus, si un nombre s (seuil déterminé par l'utilisateur) d'applications de correctifs ne permet pas de sécuriser jusqu'à l'instruction suivante, l'élimination des fuites est arrêtée.

Le système expert appliquant les correctifs a accès aux noms des signaux nommés dans le HDL. Les signaux à l'origine d'une fuite qui s'est produite dans le cycle sont nommés `racine` par la suite et sont définis comme l'ensemble des signaux qui fuient dont aucun signal ascendant ne fuit. Enfin, le système expert réalise des vérifications d'ensembles de signaux en précisant s'il faut utiliser les valeurs ou ensembles de fuite des cycles précédents ou courants ainsi que dans quels modèles de fuite réaliser ces vérifications, notamment pour les raisons identifiées dans la section courante (Exploitation des modèles de fuite pour guider le choix des correctifs ainsi que leur implémentation). Ces vérifications additionnelles sont réalisées dans le cadre des prédicats d'application des règles et dans le cadre de la fonction `formater_correctif` (par exemple pour l'application du correctif éviction du contenu d'un ou plusieurs registres).

Pour le Cortex-M4, il peut être complexe de déterminer l'instruction à l'origine d'une fuite (et donc d'identifier où appliquer le correctif) en raison du pipeline d'instructions et des accès mémoire qui peuvent être pipelinés. Dans l'éventualité où aucun correctif n'est applicable ou si la contre-mesure d'éviction n'identifie aucun registre dont le contenu doit être évincé, une unique application du correctif émulation gel du pipeline est réalisée. Cela peut, lors de la prochaine itération, clarifier l'origine de la fuite pour appliquer une autre règle de la base de règles.

Différents signaux sont nommés dans le système expert ; le tableau 6.1 récapitule la liste donnée ci-après. Les signaux disponibles sont notamment les registres (notés $\texttt{r}_\texttt{i}$, `i` étant un indice de registre du jeu d'instructions), le bus de lecture de la mémoire (noté `read_bus`), le bus d'adresse de la mémoire (noté `read_addr`), le registre interne de la

Nom	Fonction
`sécurisé`	Vérifie conjointement un ensemble de signaux dans un modèle de fuite donné. La valeur du cycle précédent (notée $^{t-1}$) ou du cycle courant (notée ^{t}) peut être choisie pour chaque signal de l'ensemble.
`cycle_ok`	Retourne vrai lorsque la vérification pour le cycle dans le modèle `RR 1-sw-probing` ne présente aucune fuite.
`racines`	Retourne l'ensemble des signaux identifiés comme l'origine d'une des fuites dans le `RR 1-sw-probing` du cycle courant.
`déjà_appliqué`	Retourne vrai si le système expert a déjà appliqué ce correctif pour cette même instruction du programme initial.

Tab. 6.2 : Récapitulatif des fonctions appelées par le système expert.

LSU pour les accès non-alignés (noté `lsu_reg`), une cellule mémoire (notée `mem_cell`), le signal indiquant si une opération mémoire a été pipelinée et les bypass mémoire→banc de registres/mémoire→ALU (respectivement notés `fwd_lsu_reg` et `fwd_lsu_alu`).

Le système expert, afin de déterminer précisément le correctif à appliquer, peut avoir recours aux fonctions définies dans le tableau 6.2.

La base de règles qui détermine un correctif adapté à l'état courant pour le Cortex-M4 est donné ci-après. Cette base de règles est celle utilisée dans la fonction `système_expert` de l'algorithme 6 qui est appelée par l'algorithme 4 pour réaliser l'élimination automatique des fuites pour ce processeur. Dès qu'une règle s'applique, le programme source est modifié et la prochaine itération d'élimination de fuites est lancée sans essayer les autres règles. D'autres bases de règles pour ce même processeur pourraient être définies et utiliser d'autres correctifs. L'établissement d'une base de règle optimale (en termes de complétude, de complexité de mise en œuvre ou de taille des programmes après correction) est abordé en perspective à la fin de ce chapitre.

1. Si la vérification conjointe des registres du banc de registres du cycle *précédent* avec le bus mémoire en lecture du cycle *courant* échoue en valeur sans glitch.

   L'état du banc de registres avant écriture est incompatible avec la valeur lue depuis la mémoire à ce cycle.

   Prédicat : `!sécurisé((g=0,t=0) 1-sw-probing, read_bus`^{t} $\cup$ $\left(\bigcup^{i} r_i{}^{t-1}\right)$)

   Action : éviction du contenu d'un ou plusieurs registres

2. Si la vérification en transition sans glitch du bus de lecture mémoire échoue ou si le cycle présente une fuite en transition avec glitches dont le signal à l'origine est le bus de lecture mémoire.

   > Une fuite en transition sur le bus de lecture de la mémoire ou une fuite en transition avec glitches est causée par le bus de lecture de la mémoire.

   Prédicat : `(!cycle_ok` et `racine == read_bus)`

   Action : nettoyage bus mémoire lecture

3. Si le cycle fuit en transition avec glitches et que le signal à l'origine est le signal d'adresse de lecture mémoire.

   > Une opération mémoire avec une adresse symbolique a été effectuée précédemment et le bus d'adresse fuit avec une autre valeur faisant partie de l'état.

   Prédicat : `(!cycle_ok` et `racine == read_addr)`

   Action : nettoyage bus mémoire lecture (Ici l'objectif est de nettoyer le bus d'*adresse* et non particulièrement celui de *données*)

4. Si la vérification conjointe des registres du banc de registres du cycle *courant* en valeur sans glitch échoue.

   > Une instruction a écrit en registre une valeur qui ne respecte pas l'hypothèse que les registres ne doivent pas fuir lorsqu'ils sont vérifiés conjointement.

   Prédicat : `!sécurisé((g=0,t=0) 1-sw-probing,` $\left(\bigcup^{\mathtt{i}} \mathtt{r_i}^{\mathtt{t}}\right)$`)`

   Action : éviction du contenu d'un ou plusieurs registres

5. Si le cycle fuit en transition avec glitches.

   Prédicat : `!cycle_ok`

   a) Si la vérification des registres individuels du banc de registres du cycle *courant* en transition avec glitches échoue.

      > Une fuite en transition avec glitches dans un registre indique qu'une valeur qui va être écrite est incompatible avec un des registres.

      Prédicat : $\exists$`i` $t.q.$ $\mathtt{r_i}^{\mathtt{t}} \in$ `racines`

      Action : éviction du contenu d'un ou plusieurs registres

   b) Si les bypass du pipeline « mémoire→banc de registres » ou « mémoire→ALU » sont les signaux à l'origine de la fuite du cycle.

Les bypass en provenance de la mémoire interfèrent avec l'état courant. Le nettoyage du bus de lecture nettoie aussi les bypass.

Prédicat : `fwd_lsu_reg ∈ racines` ou `fwd_lsu_alu ∈ racines`

Action : nettoyage bus mémoire lecture

c) Si la LSU a pipeliné deux opérations.

Lorsque deux opérations mémoire sont pipelinées, il est complexe d'identifier quelle opération est source de la fuite. Ajouter une opération entre les deux n'élimine probablement pas la fuite mais supprime l'ambiguïté, ce qui permettra à un prochain correctif d'éliminer la fuite sur la bonne instruction.

Prédicat : `lsu_pipe_advance == 1`

Action : émulation gel du pipeline

d) *Appliqué une unique fois par instruction initiale* : Si le registre interne de la LSU est à l'origine de la fuite.

Le registre interne de la LSU peut être source de glitches qui se propagent à travers de nombreux éléments de l'état courant.

Prédicat : `!déjà_appliqué` et `lsu_reg ∈ racines`

Action : nettoyage registre interne mémoire

e) Si le registre interne de la LSU est à l'origine de la fuite.

Le nettoyage du registre interne de la LSU ne peut pas être réalisé sans fuite ; nettoyer l'état pourra permettre de réaliser ce nettoyage.

Prédicat : `lsu_reg ∈ racines`

Action : éviction du contenu d'un ou plusieurs registres

f) *Appliqué une unique fois par instruction initiale* : Si une cellule mémoire est à l'origine de la fuite.

La cellule mémoire ciblée pour écriture fuit avec l'état courant du processeur qui se propage en glitches sur le bus d'écriture.

Prédicat : `!déjà_appliqué` et `mem_cell ∈ racines`

Action : nettoyage cellule mémoire

g) Si une cellule mémoire est à l'origine de la fuite.

Le nettoyage de la cellule ne peut pas être réalisé sans fuite (probablement car un glitch a lieu sur le port d'écriture de la mémoire vers la cellule mémoire).

Prédicat : `mem_cell ∈ racines`

Action : éviction du contenu d'un ou plusieurs registres

h) *Appliqué une unique fois par instruction initiale*.

Prédicat : `!déjà_appliqué`

Action : nettoyage ALU

i) *Abandon sans correctif : échec de la sécurisation.*

6. Sinon, aucune règle ne s'applique et aucune fuite dans le `RR 1-sw-probing` n'a été identifiée jusqu'à ce cycle : continuer la simulation du programme.

La conception du système expert pour des processeurs sans pipeline d'exécution (décodage et exécution réalisés dans un même étage) comme l'Ibex nécessite moins de règles. Un compromis est nécessaire entre la quantité/précision des règles et la performance des programmes sécurisés finaux. En effet, des règles précises permettent de limiter l'application de correctifs qui ne sont pas nécessaires mais imposent de connaître plus précisément le fonctionnement de la microarchitecture.

6.3 Résultats de sécurisation pour le Cortex-M4

La section précédente a défini le système expert intégré dans `aLEAKator` pour éliminer les fuites de programmes exécutés sur le processeur ARM Cortex-M4. Cette section démontre la pertinence de ce système expert et de la stratégie de sécurisation en détaillant les résultats d'élimination sur différents programmes.

Pour chaque programme sécurisé, l'évolution du nombre de cycles/d'instructions, le nombre d'itérations de sécurisation, le temps nécessaire pour sécuriser le programme puis les différents correctifs appliqués sont donnés.

Les programmes considérés sont le `dom_and`, le `secmult`, le programme de `refresh` présenté dans la section 5.4 (Comparaison cycle-à-cycle des verdicts de fuite avec des mesures réelles) et l'`AES` masqué avec le schéma de Herbst. Le tableau 6.3 détaille les résultats de sécurisation et le tableau 6.4 présente les différents correctifs appliqués. Toutes les règles du système expert ont été appliquées au moins une fois au cours de l'élimination des fuites de ces programmes. L'élimination des fuites augmente le nombre de cycles d'exécution des programmes d'un facteur de 3,4 pour l'`AES` et triple le nombre d'instructions

des programmes AES et dom_and. Chaque expérience a produit une version sécurisée dans le modèle RR 1-sw-probing. Il est cependant important de noter qu'aLEAKator n'offre aucune garantie que le système expert puisse conclure systématiquement sur une version sans fuite quel que soit le programme d'entrée.

Programme	Cycles		Instructions		Itérations	Temps
	Avant	Après	Avant	Après		
dom_and	48	135	27	91	18	2m40s
secmult	639	672	531	554	6	4m23s
refresh	40	41	29	30	1	10s
aes_herbst	8 406	29 005	7 119	21 766	6 584	≈ 1 mois

Tab. 6.3 : Élimination des fuites sur des programmes exécutés sur le processeur ARM Cortex-M4.

Correctif	dom_and	secmult	refresh	aes_herbst
Nettoyage bus mémoire lecture	2	1	0	1 518
Éviction du contenu d'un registre	16	10	0	3 325
Nettoyage cellule mémoire	0	0	0	788
Émulation gel du pipeline	2	0	0	776
Nettoyage ALU	2	0	1	862
Nettoyage registre interne mémoire	0	0	0	57

Tab. 6.4 : Correctifs appliqués pour éliminer les fuites des programmes sur le processeur ARM Cortex-M4.

Aucun travail de l'état de l'art n'a, à notre connaissance, proposé de méthode de sécurisation automatique de programmes pouvant exploiter la microarchitecture réelle sur laquelle s'exécutent les programmes comme dans ces travaux. La méthode détaillée dans ce chapitre permet l'obtention de programmes sécurisés dans le RR 1-sw-probing sur la microarchitecture finale là où les méthodes existantes, présentées dans la section 3.5, ne prennent pas en compte la microarchitecture ou prennent en compte un modèle abstrait incomplet [ZM24]. Il s'agit, à notre connaissance, de la première sécurisation (automatique ou non) d'un AES masqué avec le schéma de Herbst dans le RR 1-sw-probing. Les résultats présentés sont une preuve de concept montrant que cette piste de méthode de résolution peut fonctionner en pratique. La prochaine section détaille les différentes limitations et les améliorations possibles pour cette méthode d'élimination automatique de fuites.

6.4 Limitations et horizon

Ordonnancement des instructions : La principale limitation observée est que de très nombreuses itérations ne sont utiles que pour réaliser les opérations inverses de ce que le

compilateur a généré. Vu que les opérations mémoire sont rassemblées au maximum par le compilateur, cela donne souvent lieu à des chargements de données qui doivent être évincées directement puis rechargées pour que le chargement suivant (qui ne sera utile que plus tard) soit effectué pour lui-même être évincé car la donnée initialement chargée est la première utilisée. Cependant, un autre ordonnancement des instructions mémoire du programme initial, au prix de la limitation du gain du pipelining des opérations mémoire, pourrait permettre de limiter significativement le nombre d'itérations et de correctifs appliqués.

Pour illustrer ce propos, le listing 6.af donne un exemple simple et la figure 6.23 propose un ordonnancement alternatif au programme du ET logique masqué avec le schéma `DOM` pour lequel la performance d'élimination automatique des fuites est donnée dans le tableau 6.5. Le résumé des contre-mesures appliquées dans ce cas est présenté dans le tableau 6.6.

```
1  // Chargements
2  ldr r0, =m
3  ldr r1, [r0] // r1<-m
4  ldr r0, =a0
5  ldr r2, [r0] // r2<-a0
6  ldr r0, =a1
7  ldr r12, [r0] // r12<-a1
8
9  // Remasquages
10 eor r2, r1, r2
11 eor r12, r1, r12
```

Listing 6.af : Exemple de programme dont l'ordonnancement nécessitera de nombreuses itérations d'élimination de fuites. Une première approche, qui est garantie de réduire le nombre d'itérations, est de ne charger le share a1 qu'avant son utilisation, sans quoi le share a0 est chargé (ligne 5) puis déchargé (ligne 7) pour charger a1, lui-même déchargé (ligne 10) pour recharger a0 pour exécuter l'instruction de la ligne 10. Charger a1 après la ligne 10 permet d'éviter ces évictions/rechargements multiples.

Lorsque cela est possible, un réordonnancement des opérations mémoire devrait être réalisé ; cette opération semble automatisable à l'aide d'une analyse statique et permettrait d'appliquer une politique de chargement « au plus tard ». Une politique similaire pour réaliser des déchargements « au plus tôt » des valeurs mortes en registre pourrait aussi être imaginée, mais elle pourrait éventuellement avoir un coût supérieur en instructions en nettoyant inutilement des registres. Cela pourrait être combiné à une analyse de vivacité des registres contenant une valeur symbolique pour les nettoyer préventivement mais le gain de cette stratégie reste à mesurer.

```
ldr r0, =m
ldr r1, [r0] // r1<-m
ldr r0, =a0
ldr r2, [r0] // r2<-a0
ldr r0, =a1
ldr r12, [r0] // r12<-a1
ldr r0, =b1
ldr r3, [r0] // r3<-b1
ldr r0, =b0
ldr r6, [r0] // r6<-b0

and r4, r2, r3 // r4<-a0&b1
eor r4, r4, r1 // r4<-(a0&b1)^m
and r11, r12, r3 // r11<-a1&b1
// r11<-(a1&b1)^((a0&b1)^m)
eor r11, r11, r4
and r7, r12, r6 // r7<-a1&b0
eor r7, r7, r1 // r7<-(a1&b0)^m
and r8, r2, r6 // r8<-a0&b0
// r5<-(a0&b0)^((a1&b0)^m)
eor r5, r7, r8

ldr r0, =res0
str r11, [r0] // Sauvegarde c0
ldr r10, =res1
str r5, [r10] // Sauvegarde c1
```

Listing 6.ag : Version non réordonnancée

```
ldr r0, =m
ldr r1, [r0] // r1<-m
ldr r0, =a0
ldr r2, [r0] // r2<-a0
ldr r0, =b1
ldr r3, [r0] // r3<-b1

and r4, r2, r3 // r4<-(a0&b1)
eor r4, r4, r1 // r4<-(a0&b1)^m
ldr r0, =a1
ldr r12, [r0]   // r12<-a1
and r11, r12, r3 // r11<-a1&b1
// r11<-(a1&b1)^((a0&b1)^m)
eor r11, r11, r4
ldr r0, =res0
str r11, [r0] // Sauvegarde c0
ldr r0, =b0
ldr r6, [r0] // r6<-b0
and r7, r12, r6 // r7<-a1&b0
eor r7, r7, r1 // r7<-(a1&b0)^m
and r8, r2, r6 // r8<-a0&b0
// r5<-(a0&b0)^((a1&b0)^m)
eor r5, r7, r8

ldr r10, =res1
str r5, [r10] // Sauvegarde c1
```

Listing 6.ah : Version réordonnancée

Fig. 6.23 : Exemple de différence d'ordonnancement de programme, appliqué au ET masqué avec le schéma DOM. Les instructions sont identiques, seul l'ordre diffère entre ces deux versions.

Programme	Cycles		Instructions		Itérations	Temps
	Avant	Après	Avant	Après		
`réordo_dom_and`	48	82	27	51	7	1m03s
`dom_and`	48	135	27	91	18	2m40s

Tab. 6.5 : Différences de performance de l'élimination des fuites pour deux ordonnancements distincts des instructions du ET logique 32 bits masqué avec le schéma `DOM` sur le processeur ARM Cortex-M4.

Correctif	`réordo_dom_and`	`dom_and`
Nettoyage bus mémoire lecture	0	2
Éviction du contenu d'un registre	9	16
Nettoyage cellule mémoire	0	0
Émulation gel du pipeline	0	2
Nettoyage ALU	1	2
Nettoyage registre interne mémoire	0	0

Tab. 6.6 : Différences d'application des correctifs pour deux ordonnancements distincts des instructions du ET logique 32 bits masqué avec le schéma `DOM` sur le processeur ARM Cortex-M4.

Segmentation pour résolution parallèle : étant donné que la fonctionnalité du programme n'est pas altérée par les correctifs, il serait techniquement possible de segmenter le programme et de paralléliser l'élimination des fuites des segments. Cela est réalisable à condition de définir des barrières évinçant tous les contenus des registres en mémoire et nettoyant l'état du processeur aux points de jonction des segments. Cela pourrait apporter un gain conséquent en temps d'élimination des fuites.

Programmes avec des fonctions et boucles : la limitation imposant d'*inliner* les fonctions et de dérouler les boucles est un défaut de l'implémentation actuelle. L'exploitation des *G*raphes de *F*lot de *C*ontrôle (CFG) ou l'ajout de correctifs s'exécutant conditionnellement pourrait permettre de lever cette limitation.

Support des autres processeurs utilisables dans `aLEAKator` : le système expert présenté dans ce manuscrit a été créé pour le processeur ARM Cortex-M4. Un système expert adapté au processeur Ibex a été implémenté sur le même principe de sécurisation que pour le Cortex-M4 et permet la sécurisation des programmes `Secmult` et `DOM_AND` mais n'a pas encore été testé sur l'AES. Des détails sur les correctifs implémentés pour l'Ibex sont donnés dans l'annexe D. À terme, étendre l'élimination automatique des fuites à tous les processeurs simulables par `aLEAKator` est un objectif à poursuivre.

Extraction automatique de la base de règles : la conception des bases de règles est une tâche complexe. Cela nécessite d'identifier les sources de fuite spécifiques à un processeur, de concevoir des correctifs fonctionnels et de spécifier un ordonnancement d'application des

règles pour obtenir un système expert qui fonctionne en pratique. Une méthode d'assistance pour l'étape de conception de la base de règle serait bénéfique et permettrait d'ajouter de nouveaux processeurs à moindre effort.

Optimalité de la base de règles : ces travaux préliminaires proposent une base de règle sans discuter de son optimalité. Les critères d'évaluation pourraient être multiples, notamment : la taille des programmes après sécurisation, le nombre d'itérations, la généricité des règles ainsi que la fréquence d'échec de sécurisation. La poursuite de ces travaux en abordant la question de l'optimalité selon ces critères est nécessaire.

Troisième partie

Conclusion

7 Conclusion

Plusieurs travaux récents ont proposé des méthodes d'analyse, au niveau du circuit, de sécurité face aux attaques par canaux auxiliaires d'implémentations matérielles ou logicielles protégées. Cependant, des limitations en termes de modèles de fuites, de passage à l'échelle sur des implémentations matérielles ou logicielles complexes, ainsi que le manque de comparaison avec des circuits réels avaient été identifiés. De plus, l'utilisation de méthodes formelles de vérification prenant en compte la microarchitecture pour éliminer automatiquement les fuites se produisant lors de l'exécution de programmes n'avait, à notre connaissance, jusqu'à présent pas été étudiée. Par ailleurs, ces méthodes n'ont que peu été comparées à des mesures réelles, que ce soit en termes de réalisme des modèles de fuite ou en termes de fuites identifiées. Dans ce contexte, ces travaux de thèse avaient pour but de proposer une nouvelle méthode de vérification formelle prenant en compte la microarchitecture, des modèles de fuites flexibles, des propriétés de sécurité variées et permettant la vérification à différents niveaux de granularité. Dans un second temps, ces travaux avaient pour objectif d'étudier la pertinence de l'exploitation de telles méthodes de vérification pour une application à l'élimination automatique des fuites en contraignant les instructions manipulant des informations sensibles dans les programmes implémentant des fonctions cryptographiques.

Le chapitre 5 a introduit `aLEAKator`, une implémentation disponible sous licence libre, d'une nouvelle méthode formelle de vérification d'implémentations cryptographiques matérielles ou logicielles masquées, basée sur une simulation en domaines mixtes. Cette méthode permet d'utiliser les modèles de fuites de l'état de l'art et introduit le concept de granularité de vérification, relatif à la taille des signaux supportant l'information en pratique. Après avoir formalisé la notion de circuit, de fuite et de leur propagation, nous avons proposé des optimisations des vérifications à réaliser pour analyser un circuit dans différents modèles de fuite. Des résultats de vérification pour quatre programmes sur cinq processeurs différents ont été détaillés. De plus, des comparaisons cycle-à-cycle ont été réalisées entre les verdicts d'`aLEAKator` et des mesures sur des circuits réels du Cortex-M3 et du Cortex-M4, qui sont deux processeurs mondialement utilisés. Pour la première fois, une implémentation du schéma de Herbst de l'AES a été vérifiée complètement dans le modèle `RR 1-sw-probing` (bien que protégée pour le modèle `(0,0) 1-sw-probing` et donc non exempte de fuites) et cette vérification a été réalisée dans ces travaux sur cinq processeurs différents et utilisant les jeux d'instructions RISC-V et ARM.

Le chapitre 6 réalise un inventaire des sources communes de fuites lors de l'exécution de programmes masqués sur des processeurs et des solutions possibles pour les éliminer. Les expressions exactes présentes dans l'état du processeur simplifient considérablement l'explication des fuites observées en pratique, nécessaire pour cet inventaire. À partir de cette analyse, une méthode exploitant `aLEAKator` a été proposée afin d'éliminer automatiquement les fuites survenant lors de l'exécution de programmes cryptographiques sur le processeur ARM Cortex-M4 en modifiant itérativement les programmes. Les résultats de sécurisation de quatre programmes sont détaillés. De manière notable, la version générée pour le programme AES masqué avec le schéma de Herbst est, à notre connaissance, la première implémentation de ce programme sécurisée dans le modèle `RR 1-sw-probing` pour le Cortex-M4.

Ces travaux proposent des avancées concrètes par rapport aux limitations identifiées dans l'état de l'art pour l'analyse et le renforcement de sécurité face aux attaques par canaux auxiliaires d'implémentations cryptographiques matérielles et logicielles.

7.1 Perspectives

Différentes perspectives d'évolution de ces travaux sont données dans cette section.

Amélioration d'`aLEAKator` : des résultats issus de vérifications à l'ordre supérieur et pour des propriétés de sécurité comme SNI ont été produits par `aLEAKator`. Cependant, ces vérifications ne bénéficient pas d'optimisations sur le nombre de signaux à vérifier comme c'est le cas pour les vérifications de NI et TPS dans le modèle `RR 1-sw-probing`. Ainsi, des propositions de réduction du nombre de signaux à vérifier basées sur la structure des circuits pour des propriétés de sécurité/modèles de fuites/ordres supérieurs sont une piste à explorer. De plus, d'autres propriétés telles que PINI et o-PINI seraient intéressantes à prendre en charge pour améliorer encore la polyvalence d'`aLEAKator`. Enfin, l'implémentation de l'outil peut encore être améliorée pour en simplifier l'utilisation et en améliorer les performances.

Prise en charge de nouvelles applications et de nouveaux processeurs : la polyvalence d'`aLEAKator` a été démontrée à l'aide de vérifications sur de nombreux processeurs. Cependant, les processeurs actuellement proposés restent de complexité faible à moyenne. L'ajout de processeurs plus complexes ou implémentant des fonctionnalités de sécurité serait bénéfique pour l'étude réalisée par `aLEAKator`. De plus, l'ajout de nouvelles applications pour la vérification dans `aLEAKator` serait intéressante. Notamment, les applications de cryptographie post-quantique (PQC) sont actuellement fortement étudiées et vérifier, même

partiellement, les algorithmes PQC de l'état de l'art serait une contribution utile à leur développement.

Élimination automatique de fuites : les programmes actuellement supportés pour l'élimination automatique des fuites sont limités à ceux exempts de boucles, macros, fonctions et qui n'utilisent pas d'instructions mémoire multiples. Ces limitations sont principalement des hypothèses réalisées pour simplifier la conception de la méthode initiale d'élimination de fuites et pourraient théoriquement être évitées. De plus, bien qu'actuellement aucune méthode ne permette de réaliser l'élimination de fuites comme proposé par `aLEAKator`, les autres méthodes présentées dans l'état de l'art permettent de produire des programmes sécurisés. Comparer les résultats d'`aLEAKator` avec ces programmes en termes de performance, de taille et de sécurité des programmes produits est la suite directe de ces travaux sur l'élimination de fuites. Enfin, le support de nouveaux processeurs et l'ajout de la possibilité de réordonnancer des instructions sont aussi des points à étudier. La définition de règles optimales ainsi que d'une méthode permettant d'obtenir une base de règles sont également des éléments à considérer. Les avancées sur ces pistes pourront mener à la publication des résultats présentés dans ces travaux.

Applications alternatives de la méthode de simulation symbolique à d'autres problèmes : les simulations symboliques de processeurs peuvent être utilisées pour des problèmes autres que la vérification d'absence de fuite face aux attaques par canaux auxiliaires. Notamment, ces simulations permettent de détecter aisément lorsque le nombre de cycles d'exécution dépend des secrets, ce qui est une caractéristique nécessaire à prendre en compte lors de la conception d'implémentations cryptographiques. En effet, si le nombre de cycles d'exécution dépend des secrets, le PC du processeur deviendra nécessairement symbolique, empêchant de poursuivre la simulation. De même, l'approche symbolique peut être appliquée à l'analyse de résistance face à l'injection de certaines fautes au sein de processeurs implémentant des contre-mesures de détection de ces fautes. Par exemple, rendre un registre symbolique peut permettre d'identifier qu'après un nombre de cycles donné, une alarme a bien été levée par le processeur (détection) ou que plus aucune valeur symbolique n'est présente dans l'état et en mémoire (correction).

7.2 Publications

7.2.1 Articles

— Amiot, N., Meunier, Q., Heydemann, K., & Encrenaz, E. (2026). aLEAKator : HDL Mixed-Domain Simulation for Masked Hardware & Software Formal Verification. IACR *Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2), 132-166. `https://doi.org/10.46586/tches.v2026.i2.132-166`

7.2.2 Communications

— *Workshop on Practical Hardware Innovations in Security Implementation and Characterization* (PHISIC), Présentation, mai 2025, Gardanne, France.

— *Journées du Groupe de Travail sur les Méthodes Formelles pour la Sécurité* (GTMFS) Présentation, mars 2026, Luz-Saint-Sauveur, France.

— *Japan Society for the Promotion of Science Summer Fellowship 2026* (JSPS) Poster, juin 2026, Hayama, Japon.

7.2.3 Productions

— `aLEAKator`[1] : L'implémentation a reçu le badge « Artifact Functional » dans le cadre de la publication de l'article `aLEAKator` à TCHES.

1. Disponible sous licence libre sur github : `https://github.com/noeamiot/aLEAKator`.

Bibliographie

[ABP12] Giovanni AGOSTA, Alessandro BARENGHI et Gerardo PELOSI. « A code morphing methodology to automate power analysis countermeasures ». In : *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California : Association for Computing Machinery, 2012, p. 77-82 (cf. p. 40).

[Agr+03] Dakshi AGRAWAL, Bruce ARCHAMBEAULT, Josyula R. RAO et Pankaj ROHATGI. « The EM Side-Channel(s) ». In : 2003, p. 29-45 (cf. p. 7).

[Ama+11] Antoine AMARILLI, Sascha MÜLLER, David NACCACHE et al. « Can Code Polymorphism Limit Information Leakage ? » In : *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*. Sous la dir. de Claudio A. ARDAGNA et Jianying ZHOU. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 1-21 (cf. p. 18, 40).

[Ami+26] Noé AMIOT, Quentin MEUNIER, Karine HEYDEMANN et Emmanuelle ENCRENAZ. « aLEAKator : HDL Mixed-Domain Simulation for Masked Hardware & Software Formal Verification ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2026.2 (avr. 2026), p. 132-166 (cf. p. 23, 47).

[Ars+20] Muhammad ARSATH K F, Vinod GANESAN, Rahul BODDUNA et Chester REBEIRO. « PARAM : A Microprocessor Hardened for Power Side-Channel Attack Resistance ». In : *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, p. 23-34 (cf. p. 32).

[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 : *Smart Card Research and Advanced Applications*. Sous la dir. de Marc JOYE et Amir MORADI. Cham : Springer International Publishing, 2015, p. 64-81 (cf. p. 23).

[BP18] Alessandro BARENGHI et Gerardo PELOSI. « Side-channel security of superscalar CPUs : evaluating the impact of micro-architectural features ». In : *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California : Association for Computing Machinery, 2018 (cf. p. 23).

[Bar+19] Gilles BARTHE, Sonia BELAÏD, Gaëtan CASSIERS et al. « maskVerif : Automated Verification of Higher-Order Masking in Presence of Physical Defaults ». In : 2019, p. 300-318 (cf. p. 23, 33, 35).

[Bar+16] Gilles BARTHE, Sonia BELAÏD, François DUPRESSOIR et al. « Strong Non-Interference and Type-Directed Higher-Order Masking ». In : 2016, p. 116-129 (cf. p. 29, 88).

[Bar+15] Gilles BARTHE, Sonia BELAÏD, François DUPRESSOIR et al. « Verified Proofs of Higher-Order Masking ». In : 2015, p. 457-485 (cf. p. 29, 33, 88).

[Bay+11] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert et Paolo Ienne. « A first step towards automatic application of power analysis countermeasures ». In : *Proceedings of the 48th Design Automation Conference*. DAC '11. San Diego, California : Association for Computing Machinery, 2011, p. 230-235 (cf. p. 40, 93).

[Bay+12] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne et Wayne Burleson. « An architecture independent instruction shuffler to protect against side-channel attacks ». In : *ACM Trans. Archit. Code Optim.* 8.4 (jan. 2012) (cf. p. 17).

[Bea] BearSSL. *BearSSL - Main*. URL : `https://www.bearssl.org/` (visité le 9 avr. 2026) (cf. p. 18).

[Bec+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel et Ingrid Verbauwhede. « Provable Secure Software Masking in the Real-World ». In : 2022, p. 215-235 (cf. p. 2, 23).

[Bei+25] Christof Beierle, Jakob Feldtkeller, Anna Guinet et al. « INDIANA - Verifying (Random) Probing Security Through Indistinguishability Analysis ». In : 2025, p. 33-63 (cf. p. 34).

[Bel+20] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Quentin Meunier et Inès Ben El Ouahma. « Maskara : Compilation of a Masking Countermeasure With Optimized Polynomial Interpolation ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), p. 3774-3786 (cf. p. 40).

[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* 9 (2019), p. 231-242 (cf. p. 33, 35).

[BF] Trail of Bits et Contributeurs du Forum. *[RFC] Constant-Time Coding Support*. URL : `https://discourse.llvm.org/t/rfc-constant-time-coding-support/87781/28` (visité le 9 avr. 2026) (cf. p. 18).

[Bor+12] Julia Borghoff, Anne Canteaut, Tim Güneysu et al. « PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract ». In : 2012, p. 208-225 (cf. p. 74).

[BKN22] Dusan Bozilov, Miroslav Knezevic et Ventzislav Nikov. « Optimized threshold implementations : securing cryptographic accelerators for low-energy and low-latency applications ». In : 12.1 (avr. 2022), p. 15-51 (cf. p. 21, 74).

[BCO04] Eric Brier, Christophe Clavier et Francis Olivier. « Correlation Power Analysis with a Leakage Model ». In : 2004, p. 16-29 (cf. p. 15).

[Buh+22] Ileana Buhan, Lejla Batina, Yuval Yarom et Patrick Schaumont. « SoK : Design Tools for Side-Channel-Aware Implementations ». In : 2022, p. 756-770 (cf. p. 31).

[CAB25] Adrien Cassagne, Noé Amiot et Manuel Bouyer. *Dalek : An Unconventional and Energy-Aware Heterogeneous Cluster*. 2025. arXiv : `2508.10481 [cs.DC]` (cf. p. 72).

[CS20] Gaëtan Cassiers et François-Xavier Standaert. « Trivially and efficiently composing masked gadgets with probe isolating non-interference ». In : *IEEE Transactions on Information Forensics and Security* 15 (2020), p. 2542-2555 (cf. p. 29, 88).

[CCR26] CCRA. *Common Criteria for Information Technology Security Evaluation (CC)*. 2026. URL : https://www.commoncriteriaportal.org/cc/index.cfm (visité le 15 avr. 2026) (cf. p. 2).

[Cha+99] Suresh CHARI, Charanjit S. JUTLA, Josyula R. RAO et Pankaj ROHATGI. « Towards Sound Approaches to Counteract Power-Analysis Attacks ». In : 1999, p. 398-412 (cf. p. 19).

[CRR03] Suresh CHARI, Josyula R. RAO et Pankaj ROHATGI. « Template Attacks ». In : 2003, p. 13-28 (cf. p. 15).

[Con13] Capstone Engine CONTRIBUTORS. *The Ultimate Disassembly Framework – Capstone – The Ultimate Disassembler*. 2013. URL : https://www.capstone-engine.org/ (visité le 2 juin 2026) (cf. p. 121).

[Con18] lowRISC CONTRIBUTORS. *Security Features — Ibex Documentation 0.1.dev50+geede2fbbe documentation*. 2018. URL : https://ibex-core.readthedocs.io/en/latest/03_reference/security.html (visité le 28 avr. 2026) (cf. p. 17, 74).

[Con20] lowRISC CONTRIBUTORS. *Theory of Operation - OpenTitan Documentation*. 2020. URL : https://opentitan.org/earlgrey_1.0.0/book/hw/ip/otbn/doc/theory_of_operation.html (visité le 28 avr. 2026) (cf. p. 18).

[CK10] Jean-Sébastien CORON et Ilya KIZHVATOV. « Analysis and Improvement of the Random Delay Countermeasure of CHES 2009 ». In : 2010, p. 95-109 (cf. p. 18).

[CKN01] Jean-Sébastien CORON, Paul C. KOCHER et David NACCACHE. « Statistics and Secret Leakage ». In : 2001, p. 157-173 (cf. p. 13).

[CGD18] Yann Le CORRE, Johann GROSSSCHÄDL et Daniel DINU. « Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors ». In : 2018, p. 82-98 (cf. p. 32).

[Cou+16] Damien COUROUSSÉ, Thierno BARRY, Bruno ROBISSON et al. « Runtime Code Polymorphism as a Protection Against Side Channel Attacks ». In : *Information Security Theory and Practice*. Sous la dir. de Sara FORESTI et Javier LOPEZ. Cham : Springer International Publishing, 2016, p. 136-152 (cf. p. 18, 40).

[Dob+25] Daniel DOBKIN, Edut KATZ, David POPOVTZER et Itamar LEVI. « EMI Shielding for Use in Side-Channel Security : Analysis, Simulation, and Measurements ». In : *J. Emerg. Technol. Comput. Syst.* 22.1 (déc. 2025) (cf. p. 17).

[EW14] Hassan ELDIB et Chao WANG. « Synthesis of Masking Countermeasures against Side Channel Attacks ». In : *Computer Aided Verification*. Sous la dir. d'Armin BIERE et Roderick BLOEM. Cham : Springer International Publishing, 2014, p. 114-130 (cf. p. 40).

[End+17] Maik ENDER, Samaneh GHANDALI, Amir MORADI et Christof PAAR. « The First Thorough Side-Channel Hardware Trojan ». In : 2017, p. 755-780 (cf. p. 73).

[EM26] Mathieu ESCOUTELOUP et Vincent MIGLIORE. « A Hardware Design Methodology to Prevent Microarchitectural Transition Leakages ». In : *Constructive Approaches for Security Analysis and Design of Embedded Systems*. Sous la dir. de Matthieu RIVAIN et Pascal SASDRICH. Cham : Springer Nature Switzerland, 2026, p. 478-501 (cf. p. 18).

[Fan+23] Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié et Olivier Potin. « Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Ileana Buhan et Tobias Schneider. Cham : Springer International Publishing, 2023, p. 25-44 (cf. p. 7).

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga et François-Xavier Standaert. « Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model ». In : 2018.3 (2018), p. 89-120 (cf. p. 27).

[FH08] J. Ferrigno et M. Hlaváč. « When AES blinks : introducing optical side channel ». In : *IET Information Security* 2 (3 2008), p. 94-98 (cf. p. 7).

[Gao+19] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song et Taolue Chen. « Quantitative Verification of Masked Arithmetic Programs Against Side-Channel Attacks ». In : *Tools and Algorithms for the Construction and Analysis of Systems*. Sous la dir. de Tomáš Vojnar et Lijun Zhang. Cham : Springer International Publishing, 2019, p. 155-173 (cf. p. 33, 35).

[Gao+16] Si Gao, Hua Chen, Wenling Wu et al. « Linear Regression Attack with F-test : A New SCARE Technique for Secret Block Ciphers ». In : 2016, p. 3-18 (cf. p. 31).

[Gau+17] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber et al. « Near Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices ». In : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), p. 2700-2713 (cf. p. 75).

[GST17] Daniel Genkin, Adi Shamir et Eran Tromer. « Acoustic Cryptanalysis ». In : 30.2 (avr. 2017), p. 392-443 (cf. p. 7).

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard et Roderick Bloem. « Coco : Co-Design and Co-Verification of Masked Software Implementations on CPUs ». In : 2021, p. 1469-1468 (cf. p. 2, 18, 23, 38, 75).

[GPM21] Barbara Gigerl, Robert Primas et Stefan Mangard. « Secure and Efficient Software Masking on Superscalar Pipelined Processors ». In : 2021, p. 3-32 (cf. p. 38).

[GP99] Louis Goubin et Jacques Patarin. « DES and Differential Power Analysis (The "Duplication" Method) ». In : 1999, p. 158-172 (cf. p. 19).

[GHM22] 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* 41.11 (2022), p. 3733-3744 (cf. p. 2, 23, 37, 85).

[GMK16] Hannes Gross, Stefan Mangard et Thomas Korak. « Domain-Oriented Masking : Compact Masked Hardware Implementations with Arbitrary Protection Order ». In : *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*. Vienna Austria : ACM, 24 oct. 2016, p. 3-3 (cf. p. 20-23, 65, 73, 76, 88, 89).

[GSM17] Hannes Gross, David Schaffenrath et Stefan Mangard. « Higher-Order Side-Channel Protected Implementations of KECCAK ». In : *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, p. 205-212 (cf. p. 88).

[GM11] Tim GÜNEYSU et Amir MORADI. « Generic Side-Channel Countermeasures for Reconfigurable Devices ». In : 2011, p. 33-48 (cf. p. 18).

[HB21] Vedad HADŽIĆ et Roderick BLOEM. « CocoAlma : A Versatile Masking Verifier ». In : *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*. T. 2. Conference Series : Formal Methods in Computer-Aided Design. Wien : TU Wien Academic Press, 2021. Chap. 9, p. 14-23 (cf. p. 28, 34, 38, 87).

[HHB24] Johannes HARING, Vedad HADŽIĆ et Roderick BLOEM. « Closing the Gap : Leakage Contracts for Processors with Transitions and Glitches ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024 (2024), p. 110-132 (cf. p. 38).

[He+19] Miao HE, Jungmin PARK, Adib NAHIYAN et al. « RTL-PSC : Automated Power Side-Channel Leakage Assessment at Register-Transfer Level ». In : *2019 IEEE 37th VLSI Test Symposium (VTS)*. 2019, p. 1-6 (cf. p. 32).

[HOM06] Christoph HERBST, Elisabeth OSWALD et Stefan MANGARD. « An AES Smart Card Implementation Resistant to Power Analysis Attacks ». In : 2006, p. 239-252 (cf. p. 3, 18, 19, 44, 76, 169).

[Hom+14] Naofumi HOMMA, Yu-ichi HAYASHI, Noriyuki MIURA et al. « EM Attack Is Non-invasive ? - Design Methodology and Validity Verification of EM Attack Sensor ». In : 2014, p. 1-16 (cf. p. 17).

[HME25] Xunyue HU, Quentin L. MEUNIER et Emmanuelle ENCRENAZ. « Blind-Folded : Simple Power Analysis Attacks using Data with a Single Trace and no Training ». In : 2025.1 (2025), p. 475-496 (cf. p. 7, 13).

[Int26] RISC-V INTERNATIONAL. *RISC-V Ratified Specifications Library : : RISC-V Ratified Specifications Library*. 2026. URL : `https://docs.riscv.org/reference/home/index.html` (visité le 1er juin 2026) (cf. p. 177).

[ISW03] Yuval ISHAI, Amit SAHAI et David WAGNER. « Private Circuits : Securing Hardware against Probing Attacks ». In : 2003, p. 463-481 (cf. p. 19, 20, 26, 35, 89).

[KS22] Taehun KIM et Youngjoo SHIN. « ThermalBleed : A Practical Thermal Side-Channel Attack ». In : *IEEE Access* 10 (2022), p. 25718-25731 (cf. p. 7).

[Kit+26] Taiki KITAZAWA, Lennert WOUTERS, Benedikt GIERLICHS et al. « Active Electromagnetic Side-Channel Analysis : Crossing Physical Security Boundaries through Impedance Variations ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2026.1 (jan. 2026), p. 376-401 (cf. p. 7).

[KSM20] David KNICHEL, Pascal SASDRICH et Amir MORADI. « SILVER - Statistical Independence and Leakage Verification ». In : 2020, p. 787-816 (cf. p. 34, 35).

[Koc+19] Paul KOCHER, Jann HORN, Anders FOGH et al. « Spectre Attacks : Exploiting Speculative Execution ». In : 2019, p. 1-19 (cf. p. 7).

[Koc96] Paul C. KOCHER. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». In : 1996, p. 104-113 (cf. p. 1, 7).

[KJJ99] Paul C. KOCHER, Joshua JAFFE et Benjamin JUN. « Differential Power Analysis ». In : 1999, p. 388-397 (cf. p. 1, 7, 13, 14).

[Koc+11] Paul C. KOCHER, Joshua JAFFE, Benjamin JUN et Pankaj ROHATGI. « Introduction to differential power analysis ». In : 1.1 (avr. 2011), p. 5-27 (cf. p. 13, 14).

[LA04] Chris LATTNER et Vikram ADVE. « LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation ». In : *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California : IEEE Computer Society, 2004, p. 75 (cf. p. 40, 67).

[LSB22] Gaëtan LEPLUS, Olivier SAVRY et Lilian BOSSUET. « Insertion of random delay with context-aware dummy instructions generator in a RISC-V processor ». In : *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2022, p. 81-84 (cf. p. 17).

[Lim26a] Arm LIMITED. *Arm Academic Access – Arm®*. 2026. URL : `https://www.arm.com/resources/research/enablement/academic-access` (visité le 1er avr. 2026) (cf. p. 38, 76).

[Lim26b] Arm LIMITED. *Microprocessor Cores and Processor Technology – Arm®*. 2026. URL : `https://www.arm.com/products/silicon-ip-cpu` (visité le 1er avr. 2026) (cf. p. 3).

[Lip+18] Moritz LIPP, Michael SCHWARZ, Daniel GRUSS et al. « Meltdown : Reading Kernel Memory from User Space ». In : 2018, p. 973-990 (cf. p. 7).

[MPP16] Houssem MAGHREBI, Thibault PORTIGLIATTI et Emmanuel PROUFF. « Breaking Cryptographic Implementations Using Deep Learning Techniques ». In : *Security, Privacy, and Applied Cryptography Engineering*. Sous la dir. de Claude CARLET, M. Anwar HASAN et Vishal SARASWAT. Cham : Springer International Publishing, 2016, p. 3-26 (cf. p. 15).

[MOP07] Stefan MANGARD, Elisabeth OSWALD et Thomas POPP. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, jan. 2007 (cf. p. 26).

[MMS01a] David MAY, Henk L. MULLER et Nigel P. SMART. « Non-deterministic Processors ». In : 2001, p. 115-129 (cf. p. 17).

[MMS01b] David MAY, Henk L. MULLER et Nigel P. SMART. « Random Register Renaming to Foil DPA ». In : 2001, p. 28-38 (cf. p. 18).

[May00] Rita MAYER-SOMMER. « Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards ». In : 2000, p. 78-92 (cf. p. 13).

[MWO16] David MCCANN, Carolyn WHITNALL et Elisabeth OSWALD. *ELMO : Emulating Leaks for the ARM Cortex-M0 without Access to a Side Channel Lab*. Cryptology ePrint Archive, Report 2016/517. 2016 (cf. p. 31).

[MDS99] Thomas MESSERGES, Ezzy DABBISH et Robert SLOAN. « Investigations of Power Analysis Attacks on Smartcards ». In : (sept. 1999) (cf. p. 11).

[MT23] Quentin MEUNIER et Abdul TALEB. « VerifMSI : Practical Verification of Hardware and Software Masking Schemes Implementations ». In : *Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT*. INSTICC. SciTePress, 2023, p. 520-527 (cf. p. 33, 69, 70, 89).

[Meu19] Quentin L. MEUNIER. « FastCPA : Efficient Correlation Power Analysis Computation with a Large Number of Traces ». In : *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*. CS2 '19. Valencia, Spain : Association for Computing Machinery, 2019, p. 7-12 (cf. p. 15).

[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* 49.6 (2023), p. 3359-3375 (cf. p. 33).

[Mor+18] Amir MORADI, Bastian RICHTER, Tobias SCHNEIDER et François-Xavier STANDAERT. « Leakage Detection with the χ^2-Test ». In : 2018.1 (2018), p. 209-237 (cf. p. 31).

[Mos+12] Andrew MOSS, Elisabeth OSWALD, Dan PAGE et Michael TUNSTALL. « Compiler Assisted Masking ». In : 2012, p. 58-75 (cf. p. 39).

[MWB11] Ruben A. MUIJRERS, Jasper G. J. van WOUDENBERG et Lejla BATINA. « RAM : Rapid Alignment Method ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Emmanuel PROUFF. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 266-282 (cf. p. 11).

[MM22] Nicolai MÜLLER et Amir MORADI. « PROLEAD A Probing-Based Hardware Leakage Detection Tool ». In : 2022.4 (2022), p. 311-348 (cf. p. 32, 87).

[MM24] Nicolai MÜLLER et Amir MORADI. « Robust but Relaxed Probing Model ». In : 2024.4 (2024), p. 451-482 (cf. p. 2, 28, 31, 55).

[NRR06] Svetla NIKOVA, Christian RECHBERGER et Vincent RIJMEN. « Threshold Implementations Against Side-Channel Attacks and Glitches ». In : 2006, p. 529-545 (cf. p. 20).

[OC14] Colin O'FLYNN et Zhizhang (David) CHEN. « ChipWhisperer : An Open-Source Platform for Hardware Embedded Security Research ». In : 2014, p. 243-260 (cf. p. 11, 80).

[Ope] OPENSSL. *OpenSSL*. URL : `https://www.openssl.org/` (visité le 9 avr. 2026) (cf. p. 18).

[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 : *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2021, p. 179-186 (cf. p. 17).

[PBP23] Isabella PIACENTINI, Alessandro BARENGHI et Gerardo PELOSI. « A Non Profiled and Profiled Side Channel Attack Countermeasure through Computation Interleaving ». In : *2023 26th Euromicro Conference on Digital System Design (DSD)*. 2023, p. 718-725 (cf. p. 18).

[Pos+11] Axel POSCHMANN, Amir MORADI, Khoongming KHOO et al. « Side-Channel Resistant Crypto for Less than 2,300 GE ». In : 24.2 (avr. 2011), p. 322-345 (cf. p. 73).

[PUL26] PULP. *PULP FAQs*. 2026. URL : `https://www.pulp-platform.org/index.html` (visité le 3 mai 2026) (cf. p. 74, 75).

[RLT15] Ashay RANE, Calvin LIN et Mohit TIWARI. « Raccoon : Closing Digital Side-Channels through Obfuscated Execution ». In : 2015, p. 431-446 (cf. p. 18, 40).

[RWB04] G.B. Ratanpal, R.D. Williams et T.N. Blalock. « An on-chip signal suppression countermeasure to power analysis attacks ». In : *IEEE Transactions on Dependable and Secure Computing* 1.3 (2004), p. 179-189 (cf. p. 17).

[RD01] Vicent Rijmen et Joan Daemen. *Advanced Encryption Standard (AES)*. Nov. 2001 (cf. p. 19, 73, 169).

[RP10] Matthieu Rivain et Emmanuel Prouff. « Provably Secure Higher-Order Masking of AES ». In : 2010, p. 413-427 (cf. p. 35, 76).

[RPD09] Matthieu Rivain, Emmanuel Prouff et Julien Doget. « Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers ». In : 2009, p. 171-188 (cf. p. 18).

[Roc25] Thomas Roche. « EUCLEAK Side-Channel Attack on the YubiKey 5 Series (Revealing and Breaking Infineon ECDSA Implementation on the Way) ». In : 2025, p. 4026-4043 (cf. p. 7).

[Sas+15] Pascal Sasdrich, Oliver Mischke, Amir Moradi et Tim Güneysu. « Side-Channel Protection by Randomizing Look-Up Tables on Reconfigurable Hardware - Pitfalls of Memory Primitives ». In : 2015, p. 95-107 (cf. p. 18).

[Seh+20] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic et Milos Prvulovic. « EMSim : A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals ». In : *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, p. 71-85 (cf. p. 31).

[SM21a] Aein Rezaei Shahmirzadi et Amir Moradi. « Re-Consolidating First-Order Masking Schemes ». In : 2021.1 (2021), p. 305-342 (cf. p. 74).

[SM21b] Aein Rezaei Shahmirzadi et Amir Moradi. « Second-Order SCA Security with almost no Fresh Randomness ». In : 2021.3 (2021), p. 708-755 (cf. p. 21, 73).

[She+21a] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel et al. « Rosita++ : Automatic Higher-Order Leakage Elimination from Cryptographic Code ». In : 2021, p. 685-699 (cf. p. 41, 93).

[She+21b] Madura A. Shelton, Niels Samwel, Lejla Batina et al. « Rosita : Towards Automatic Elimination of Power-Analysis Leakage in Ciphers ». In : 2021 (cf. p. 31, 41, 93).

[SMY09] François-Xavier Standaert, Tal Malkin et Moti Yung. « A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks ». In : 2009, p. 443-461 (cf. p. 31).

[Stu08] Student. « The Probable Error of a Mean ». In : *Biometrika* 6.1 (1908), p. 1-25 (cf. p. 30).

[Tal+22] Ezinam Bertrand Talaki, Olivier Savry, Mathieu Bouvier Des Noes et David Hely. « A Memory Hierarchy Protected against Side-Channel Attacks ». In : *Cryptography* 6.2 (2022) (cf. p. 21).

[TH12] Qizhi Tian et Sorin A. Huss. « A General Approach to Power Trace Alignment for the Assessment of Side-Channel Resistance of Hardened Cryptosystems ». In : *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. 2012, p. 465-470 (cf. p. 11).

[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 : *Proceedings of the 28th European Solid-State Circuits Conference*. 2002, p. 403-406 (cf. p. 17).

[TV04] K. TIRI et I. VERBAUWHEDE. « A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation ». In : *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. T. 1. 2004, 246-251 Vol.1 (cf. p. 17).

[TS24] M. Caner TOL et Berk SUNAR. « ZeroLeak : Automated Side-Channel Patching in Source Code Using LLMs ». In : 2024, p. 290-310 (cf. p. 40).

[Tri03] Elena TRICHINA. *Combinational Logic Design for AES SubByte Transformation on Masked Data*. Cryptology ePrint Archive, Report 2003/236. 2003 (cf. p. 21).

[US24] Balazs UDVARHELYI et François-Xavier STANDAERT. « Leveraging Coprocessors as Noise Engines in Off-the-Shelf Microcontrollers ». In : *Smart Card Research and Advanced Applications*. Sous la dir. de Shivam BHASIN et Thomas ROCHE. Cham : Springer Nature Switzerland, 2024, p. 148-165 (cf. p. 18).

[Vie+24] Raphael VIERA, Jean-Max DUTERTRE, Rodrigo Silva LIMA, Matthieu POMMIES et Anthony BERTRAND. « Tampering with the flash memory of microcontrollers : permanent fault injection via laser illumination during read operations ». In : 14.2 (juin 2024), p. 207-221 (cf. p. 7).

[Wan+18] Chenguang WANG, Yici CAI, Haoyi WANG et Qiang ZHOU. « Electromagnetic equalizer : an active countermeasure against EM side-channel attack ». In : *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '18. San Diego, California : Association for Computing Machinery, 2018 (cf. p. 17).

[Wol13] Claire WOLF. *Yosys Open SYnthesis Suite*. 2013 (cf. p. 63).

[WP20] Lichao WU et Stjepan PICEK. « Remove Some Noise : On Pre-processing of Side-channel Measurements with Autoencoders ». In : 2020.4 (2020), p. 389-415 (cf. p. 11).

[ZM24] Jannik ZEITSCHNER et Amir MORADI. « PoMMES : Prevention of Micro-architectural Leakages in Masked Embedded Software ». In : 2024.3 (2024), p. 342-376 (cf. p. 40, 133).

[ZMM23] Jannik ZEITSCHNER, Nicolai MÜLLER et Amir MORADI. « PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries ». In : 2023.3 (2023), p. 391-421 (cf. p. 32).

[ZCF24] Feng ZHOU, Hua CHEN et Limin FAN. « Prover - Toward More Efficient Formal Verification of Masking in Probing Model ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.1 (déc. 2024), p. 552-585 (cf. p. 34, 35, 90).

Table des figures

2.1 Vue simplifiée des traitements appliqués aux sources pour produire des circuits et des logiciels. 8

2.2 Transistors *NMOS* et *PMOS*, utilisés pour implémenter des portes logiques dans la technologie *CMOS*. 9

2.3 Exemples de portes réalisées avec la technologie *CMOS*. 9

2.4 Évolution de la tension de sortie (en rouge) et du courant (en vert) pendant le changement d'état d'un inverseur CMOS dont l'entrée (en bleu) évolue de `Vss` à `Vdd` puis de nouveau à `Vss`. 10

2.5 Visualisation des glitches sur un circuit d'exemple lorsque le front montant de l'horloge `clk` déclenche des fronts montants sur les interconnexions `a` et `b`. Pendant le court instant au cours duquel la porte NOT qui pilote `c` n'a pas encore propagé son résultat sur sa sortie, `d` propage temporairement un résultat qui n'est pas le résultat stabilisé sur sa sortie. Lorsque la porte NOT propage finalement la négation de `b` sur `c`, un délai de propagation est de nouveau nécessaire pour que cette transition se répercute sur `d`. 11

2.6 Photographie du banc de mesure incluant la carte ChipWhisperer Pro et trois cibles interchangeables. 12

2.7 Différence de consommation lors de la vérification d'un code PIN selon le nombre de caractères corrects. Traces réalisées sur le `STM32F303` sur une unique exécution par PIN. 14

2.8 Superposition des consommations pour 3000 traces en sortie de la première lecture de la SBox d'AES (octet 0 de la clé) sur le `STM32F303`. La couleur est déterminée à partir du poids de Hamming du plus clair ($HW = 0$) au plus foncé ($HW = 8$). 14

2.9 Différence de distribution de la consommation à l'échantillon de la première de la SBox d'AES (octet 0 de la clé) sur le `STM32F303`, lorsque le bit de poids faible de l'indice vaut `1` ou `0`. 15

2.10 ET logique matériel, masqué avec le schéma `DOM` [GMK16]. La propriété suivante est respectée sur les sorties : `c0` $\oplus$ `c1 = (a0` $\oplus$ `a1) & (b0` $\oplus$ `b1)`. 21

2.11 Masquage logiciel du ET logique avec le schéma `DOM` [GMK16], en `C` et en `Assembleur` ARM. 22

2.12 Corrélation sur 10 000 traces entre les valeurs secrètes et la consommation pour un ET masqué avec le schéma DOM [GMK16], tel que présenté dans le listing 2.d. L'implémentation ne présente aucune transition pouvant fuire avec les informations disponibles au niveau ISA. 23

3.1 Glitches faisant apparaître k⊕ma⊕ma=k. Sans considérer les glitches, ce circuit ne présente pas de fuite. Pour ce chronogramme, le temps de propagation des interconnexions entre les portes est ignoré et le délai de stabilisation de toutes les portes est considéré identique et égal à d. Les entrées a, b et e sont respectivement pilotées aux valeurs k⊕ma, mb et ma, au même instant t. . . . 27

3.2 Gadget simpliste réalisant le XOR des shares de mêmes indices. Ce gadget est 1-PINI mais pas 1-SNI. 30

3.3 Énumération d'expressions symboliques, avec un secret k et deux masques m_1 et m_2. L'expression 3.3a est bien indépendante de la valeur du secret k puisque connaître la valeur de ce secret ne change pas la probabilité d'obtenir un résultat ou un autre sur e. L'expression 3.3b n'est pas indépendante de k, si k vaut 0 alors les deux valeurs possibles pour e' sont équiprobables, ce qui n'est plus vrai quand k vaut 1. 34

5.1 Vue d'ensemble d'aLEAKator. 48

5.2 Représentation graphique des machines de Mealy fonctionnelles. 50

5.3 (a) Un circuit d'exemple avec $I = \{a, b, c\}$, $O = \{y, z\}$ et les signaux internes $\{u, v\}$. L'ensemble des portes combinatoires est $C = \{c_0, c_1, c_2\}$ et celui des portes séquentielles $R = \{r\}$. Dans cet exemple, f_{c_0} est une porte de fonctionnalité NON, f_{c_1} de fonctionnalité XOR et f_{c_2} de fonctionnalité ET. (b) Les fonctions de transition et de génération de la machine de Mealy fonctionnelle $\mathcal{M}(\mathcal{C})$. 51

5.4 Valeurs des signaux du circuit, avec $X = \texttt{SYMB}(x)$, $Y = \texttt{SYMB}(y)$ et $Z = \texttt{SYMB}(z)$. 53

5.5 Exemple illustrant la nécessité de vérifier chaque signal dans les modèles (0,1) et (1,1) 1-sw-probing. L'entrée i_1 fuit k en transition au cycle t tandis que la sortie o_0 non. 60

5.6 Exemple illustrant la nécessité de vérifier les signaux qui sont séparés et partiellement utilisés par une porte, avec $\texttt{k} = \texttt{SYMB}(K)$ et $\texttt{m} = \texttt{SYMB}(M)$. Le signal de 2-bits fuite k tandis qu'aucun des signaux d'un bit, qui sont les entrées de registres, ne fuit. 61

5.7 Exemple illustrant la nécessité de vérifier, au cycle t, les entrées des portes qui étaient stables au cycle $t-1$. Dans le modèle (1,1) 1-sw-probing, l'entrée i_1 fuit k au cycle t tandis que la sortie o_0 ne fuit pas. 62

5.8 Implémentation d'aLEAKator. 63

5.9 Modèle pouvant être simulé du ET logique matériel, masqué avec le schéma DOM [GMK16] (description Verilog du listing 2.b de la partie I) et la machine de Mealy associée. La propriété suivante est respectée sur les sorties après propagation complète : c0 ⊕ c1 = (a0 ⊕ a1) & (b0 ⊕ b1). 65
5.10 Différents circuits pouvant être générés lors de la synthèse de la description donnée dans le listing 5.b, réalisant la lecture d'un registre parmi quatre. À titre d'exemple, le registre r2 est lu et l'adresse est bien 2'b10 dans les deux cas. 66
5.11 Temporalité des appels aux fonctions eval et commit de cxxrtl. 67
5.12 Identification de la source d'une fuite dans le cadre de la vérification en valeur avec glitches d'un circuit utilisant un masque m et un secret s. Les signaux présentant des fuites en valeur avec glitches sont indiqués en rouge et les autres en vert. Seul le signal ③ devrait être retourné pour simplifier l'identification des signaux importants pour corriger la fuite. 68
5.13 Vérification d'expressions via des substitutions avec VerifMSI++. Les expressions manipulent des variables symboliques de mêmes tailles : un masque m, une valeur publique p et des secrets k, k_0 et k_1 70
5.14 Valeurs de t-test : variantes nsec (a) et sec-rr (b) du DOM-AND sur le Cortex-M3. 81
5.15 Valeurs de t-test : variantes nsec (a) et sec-rr (b) du DOM-AND sur le Cortex-M4. 81
5.16 Valeurs de t-test : variantes nsec (a) et sec-rr (b) du Secmult sur le Cortex-M3. 81
5.17 Valeurs de t-test : variantes nsec (a) et sec-rr (b) du Secmult sur le Cortex-M4. 81
5.18 Valeur maximale du t-test par cycle pour chaque secret et verdict d'aLEAKator associé dans le modèle RR 1-sw-probing pour le DOM-AND-nsec sur le Cortex-M4. 82
5.19 Vue simplifiée du banc de registres du processeur ARM Cortex-M4. 84
5.20 Valeur maximale du t-test par cycle pour chaque secret et verdict d'aLEAKator associé dans le modèle RR 1-sw-probing pour le Refresh sur le Cortex-M4. 86
5.21 Valeur maximale du t-test par cycle pour chaque secret et verdict d'aLEAKator associé dans le modèle RR 1-sw-probing pour le Secmult-nsec sur le Cortex-M4. 87

6.1 Sources de fuite en valeur granularité bit (modèle 1-bit-probing). 95
6.2 Altération du compilateur : réordonnancement d'opérations menant à des fuites dans le modèle 1-bit-probing. 96
6.3 Altération du compilateur : suppression d'opérations menant à des fuites dans le modèle 1-bit-probing. 96
6.4 Sources de fuite en valeur granularité support (modèle 1-sw-probing). . . . 97
6.5 Altération du compilateur : organisation mémoire menant à des fuites dans le modèle 1-sw-probing. 98

6.6 Altération du compilateur : vectorisation au sein d'un mot menant à des fuites dans le modèle `1-sw-probing`. 98
6.7 Différence de comportement pour un million de traces entre les microcontrôleurs `STM32F303` et `STM32F405`, embarquant tous les deux un processeur ARM Cortex-M4, lors de la réécriture en mémoire d'un share par un autre. Le premier share n'est jamais manipulé dans le processeur ni sur les bus mémoire. 100
6.8 Vecteur démonstratif : fuite en transition lors d'une lecture mémoire. 101
6.9 Vecteur démonstratif : fuite en transition lors d'une écriture mémoire. Le correctif ne concerne que le bus mémoire, des fuites dans d'autres modules du cœur persistent. 102
6.10 Vecteur démonstratif : fuite en transition dans une cellule mémoire. 103
6.11 Vecteur démonstratif : fuite en transition lors d'une lecture non-alignée. . . 104
6.12 Vecteur démonstratif : fuite en transition lors d'une écriture non-alignée. . . 106
6.13 Exemples de ports de lecture et d'écriture d'un banc de registres. 107
6.14 `(g=0,t=1) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port de lecture. 108
6.15 `(g=1,t=0) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port de lecture. 108
6.16 `(g=0,t=1) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port d'écriture. 109
6.17 `(g=1,t=0) 1-sw-probing` : fuite dans l'arbre de multiplexeurs du port d'écriture. 109
6.18 Vecteur démonstratif : fuite en transition lors d'une lecture du banc de registres. 111
6.19 Vecteur démonstratif : fuite en valeur avec glitches lors de l'écriture du banc de registres. L'écriture est réalisée une seconde fois mais en ne manipulant que des valeurs non-symboliques pour mettre en évidence que cette fuite est uniquement due au glitch lors de l'écriture. 112
6.20 Vecteur démonstratif : fuite en valeur avec glitches lors d'une opération ALU. 114
6.21 Représentation du programme P lors de l'ajout de trois contre-mesures d'exemple A, B et C. Les instructions des contre-mesures sont nommées respectivement, avec $i \in [0, 2]$: $A_0 \dots A_i$, $B_0 \dots B_i$ et C_0. 118
6.22 Vision fortement simplifiée du processeur ARM Cortex-M4 dans un état pour lequel une éviction directement menée avec des vérifications dans le modèle `RR 1-sw-probing` échouerait. 126
6.23 Exemple de différence d'ordonnancement de programme, appliqué au ET masqué avec le schéma `DOM`. Les instructions sont identiques, seul l'ordre diffère entre ces deux versions. 135

A.1 Organisation du calcul de l'AES masqué par le schéma de Herbst. Reproduction de la figure d'un support d'enseignement de Quentin Meunier. 171

C.1 Exemple d'un port de lecture correspondant à l'hypothèse de chargement des shares. 178

Liste des tableaux

2.1 Table de vérité de e = k $\oplus$ m, avec un secret k et un masque m. 19

3.1 Soit l'opération de masquage e = a $\oplus$ m avec des variables larges de deux bits : a, m et e. Ces variables de deux bits sont composées de variables 1 bit concaténées ; Soient a=(a_1, a_0), m=(m_0, m_0) et e=(e_1, e_0). m ne peut prendre que deux valeurs car un seul bit de masque est utilisé pour le composer. La distribution de e en considérant une granularité de deux bits n'est pas uniforme. Cependant, les distributions de e_1 et de e_0 sont bien uniformes lorsque considérées indépendamment. 36

3.2 Comparaison des méthodes formelles de co-vérification logiciels-matériels de l'état de l'art. 37

5.1 Liste des opérateurs de construction des expressions symboliques. 52

5.2 Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans les domaines $\mathbb{B}^n$ et $\mathbb{E}^n$. 55

5.3 Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans le domaine $\mathbb{S}^n$. 56

5.4 Définition des fonctions calculant la fonctionnalité des portes AND, OR et XOR dans le domaine $\mathbb{L}^n$. 57

5.5 Ensembles d'expressions construits pour un signal w à un cycle t pour chaque modèle de fuite. 59

5.6 Signaux w à vérifier à un cycle t pour chaque modèle de fuite. 60

5.7 Signaux w à vérifier et ensembles d'expressions à construire pour une vérification au cycle t dans le modèle `(1,1) 1-sw-probing` lorsque la sur-approximation est considérée. 62

5.8 Impact de l'association à une technologie pour un circuit composé uniquement d'un multiplicateur $\mathbb{B}^{32} \times \mathbb{B}^{32} \rightarrow \mathbb{B}^{64}$. L'association réalise notamment la transformation des portes complexes en portes basiques de la technologie (AND, MUX, XOR, ...). 64

5.9 Fonctions permettant de simuler toutes les portes supportées par aLEAKator. 65

5.10 Résultats de vérification obtenus par aLEAKator, Coco, Prover et Prolead. Les résultats marqués ✗* correspondent aux faux-positifs (le circuit est sécurisé mais le verdict de l'outil est « non-sécurisé »). 73
5.11 Résultats de vérification détaillés d'aLEAKator. La colonne « F » référence les figures de comparaison avec des mesures réelles, présentées dans la section courante (Comparaison avec des mesures réelles). 78
5.12 Évolution des expressions symboliques des ports d'entrée A et B de l'ALU après décodage des instructions 10 à 16 du listing 5.4.2. 86
5.13 Résultats de vérification à l'ordre supérieur d'aLEAKator. 88
5.14 Résultats de vérification des propriétés NI et SNI sur les gadgets DOM-AND et ISW-AND, à l'ordre supérieur. 89

6.1 Récapitulatif des signaux nommés dans le système expert. 128
6.2 Récapitulatif des fonctions appelées par le système expert. 129
6.3 Élimination des fuites sur des programmes exécutés sur le processeur ARM Cortex-M4. 133
6.4 Correctifs appliqués pour éliminer les fuites des programmes sur le processeur ARM Cortex-M4. 133
6.5 Différences de performance de l'élimination des fuites pour deux ordonnancements distincts des instructions du ET logique 32 bits masqué avec le schéma DOM sur le processeur ARM Cortex-M4. 136
6.6 Différences d'application des correctifs pour deux ordonnancements distincts des instructions du ET logique 32 bits masqué avec le schéma DOM sur le processeur ARM Cortex-M4. 136

B.1 Fonctions pour les domaines concret et symbolique. 174
B.2 Fonctions pour le domaine de la stabilité. 175
B.3 Fonctions pour le domaine des ensembles de fuite. 176

C.1 Encodage du jeu d'instructions RV32I. 177

Liste des listings

2.a. Fonction `verify_pin` : vérification d'un mot de passe de 4 chiffres dans l'intervalle $[0, 9]$ par rapport à une référence. 13

2.b. Implémentation `Verilog`. 21

2.c. Implémentation `C`. 22

2.d. Implémentation `Assembleur`. 22

5.a. ET masqué avec le schéma DOM généré par `cxxrtl`. 65

5.c. Chargement des shares `a0` et `a1` dans le `DOM-AND-nsec`. 83

5.d. Programme `Refresh`. Les `ldr` et `mov` sont encodés en 16 bits et toutes les autres instructions en 32 bits. 84

6.a. Remasquage de `a` $\oplus$ `m` par un masque `mt`, pour obtenir `x = a` $\oplus$ `mt`. . . . 96

6.b. Version sécurisée, `mt` est ajouté avant d'enlever le masquage par `m`. 96

6.c. Version exposant le secret `a`, `m` est enlevé avant le masquage par `mt`. 96

6.d. Recombinaison des deux shares d'un secret au sein d'une même variable et remasquage par `mt`. 96

6.e. Version sécurisée, toutes les opérations décrites en `C` sont effectuées. . . . 96

6.f. Version non sécurisée, les opérations liées au masque `m` ne sont pas effectuées, exposant `a`. 96

6.g. Déclaration d'octets de shares et masques. 98

6.h. Version sécurisée, les deux mots mémoire de 32 bits ne présentent pas de fuite. 98

6.i. Version non-sécurisée, le premier mot mémoire de 32 bits fuit le secret `a`. . 98

6.j. Calcul avec des indices spécifiques entre deux tableaux, dont chaque manipulation d'octet est sécurisée au niveau de représentation `C`. 98

6.k. Version sécurisée, le compilateur n'a pas eu recours à la vectorisation et réalise les mêmes opérations que décrites en C. 98

6.l. Version non-sécurisée, la vectorisation place a0 et a1 au sein d'un même mot de 32 bits, exposant le secret a. 98

6.m. Réécriture d'un share en mémoire par le second share du même secret. . . 100

6.n. Lecture sans nettoyage préalable : le bus de lecture mémoire réalise a0 → a1. 101

6.o. Lecture avec nettoyage préalable : le bus de lecture mémoire réalise a0 → blk → a1. 101

6.p. Écritures mémoire sans cycle d'attente : le bus d'écriture mémoire réalise a0 → a1. 102

6.q. Écritures mémoire avec cycle d'attente : le bus d'écriture mémoire réalise a0 → blk → a1. 102

6.r. Écriture sans nettoyage préalable : une cellule mémoire réalise a1 → a0. . 103

6.s. Écriture avec nettoyage préalable : une cellule mémoire réalise a1 → @a0→ a0. 103

6.t. Lecture non-alignée sans nettoyage préalable, le registre interne réalise a0 → a1 (avec un désalignement égal sur ces variables). 104

6.u. Lecture non-alignée avec nettoyage préalable, le registre interne réalise a0 → blk→ a1 (avec un désalignement égal sur ces variables). 104

6.v. Écriture non-alignée sans nettoyage préalable, le registre interne réalise a0 → a1 (avec un désalignement égal sur ces variables). 106

6.w. Écriture non-alignée avec nettoyage préalable, le registre interne réalise a0 → blk → a1 (avec un désalignement égal sur ces variables). 106

6.x. Lecture de registre sans nettoyage préalable : le port de lecture du banc de registres réalise a0→a1. 111

6.y. Lecture de registre avec nettoyage préalable : le port de lecture du banc de registres réalise a0→blk→a1. 111

6.z. Écriture sans stabilisation préalable de l'adresse du registre à écrire : l'ensemble de fuite du signal d'écriture de r0 est {a0, a1} à l'écriture de r4. . . 112

6.aa. Écriture avec stabilisation préalable de l'adresse du registre à écrire : aucun glitch ne peut se produire au cours de l'écriture de r4. 112

6.ab. Remasquage avec ports de l'ALU incorrectement nettoyés. 114

6.ac. Remasquage avec ports de l'ALU correctement nettoyés. 114

6.ad. Exemple de programme annoté avec les registres lus et écrits par les instructions. Ces informations sont associées à chaque étiquette d'instruction pour composer la trace d'exécution. 122

6.ae. Exemple de correctif pour nettoyer le chemin des écritures non-alignées. . 124

6.af. Exemple de programme dont l'ordonnancement nécessitera de nombreuses itérations d'élimination de fuites. Une première approche, qui est garantie de réduire le nombre d'itérations, est de ne charger le share a1 qu'avant son utilisation, sans quoi le share a0 est chargé (ligne 5) puis déchargé (ligne 7) pour charger a1, lui-même déchargé (ligne 10) pour recharger a0 pour exécuter l'instruction de la ligne 10. Charger a1 après la ligne 10 permet d'éviter ces évictions/rechargements multiples. 134

6.ag. Version non réordonnancée 135

6.ah. Version réordonnancée 135

C.a. Décodage matériel des instructions par l'Ibex et lectures des données vers l'ALU. 178

Liste des algorithmes

1 Parcours inductif de la structure d'un circuit $\mathcal{C}$ à partir d'un signal w. 51

2 Simulation en domaines mixtes d'un signal w à un cycle t. 57

3 Vérification de la propriété TPS, adapté de [MT23]. 70

4 Stratégie d'élimination itérative des fuites d'un programme `Assembleur`. . . 119

5 Stratégie de sélection du ou des contenus de registres à évincer pour que le banc de registres et l'ensemble de signaux additionnels ne fuient plus. La méthode `sécurisé` réalise la vérification conjointe, dans un modèle de fuite donné, d'un ensemble de signaux et retourne vrai en l'absence de fuite. . . 124

6 Implémentation du système expert. La base de règles contient des règles qui sont composées d'une condition d'application appelée prédicat et d'une action. Les règles sont parcourues dans l'ordre et la première dont le prédicat est vrai s'applique en mettant fin à la fonction. 127

A

Brève explication de l'AES masqué avec le schéma de Herbst

L'*A*dvanced *E*ncryption *S*tandard (*AES*) [RD01] est un algorithme de chiffrement symétrique opérant sur des blocs de 128 bits et utilisant des clés de 128, 192 ou 256 bits. La fonction traite des blocs de valeur à chiffrer de 128 bits en entrée et produit 128 bits de message chiffré en sortie. Les messages plus longs que 128 bits sont supportés via des modes d'utilisation tels que CBC (chaînage des blocs) ou CTR (exploitant un compteur).

Le chiffrement d'un bloc de message est réalisé au travers d'une succession de rondes, dont le nombre dépend de la taille de la clé (10, 12 ou 14 rondes). Chaque opération qui compose ces rondes opère sur le *state* qui est un tableau de 16 octets (représenté informellement comme un tableau de 4×4 octets), initialisé avec la valeur du bloc à chiffrer. Chaque ronde, à l'exception de la dernière qui ne réalise pas l'opération **MixColumns** et de la première qui réalise aussi l'opération **AddRoundKey** au début, est composée des opérations successives suivantes :

1. **SubBytes**, une substitution appliquée indépendamment à chaque octet du *state* à l'aide d'une SBox (table de substitution).
2. **ShiftRows**, qui réalise une permutation des octets du *state* par ligne.
3. **MixColumns**, qui transforme les octets du *state* en utilisant les autres octets de même colonne.
4. **AddRoundKey**, consistant en une opération $\oplus$ entre le *state* et une clé de ronde.

Les clés de ronde utilisées par l'opération **AddRoundKey** sont calculées en amont par une méthode nommée **KeySchedule** qui dérive la clé principale en clés de ronde de manière déterministe.

Le schéma de masquage proposé par Herbst *et al.* [HOM06] permet de masquer, en ayant recours au masquage booléen, les octets du *state*, à chaque instant du calcul de l'AES.

Cela est réalisé en modifiant le **KeySchedule** et le **SubBytes** et en introduisant une étape nommée **ChangeMasks**. L'`AES-Herbst` a été créé pour limiter l'utilisation de masques et réutilise les mêmes masques à travers les rondes. Ainsi, six masques d'un octet chacun `m`, `m'`, `m1`, `m2`, `m3` et `m4` sont nécessaires. Les masques `m` et `m'` sont utilisés pour masquer tous les octets du *state* respectivement avant et après l'opération **SubBytes**. Les masques `m1`,

`...`, `m4` sont utilisés pour masquer chaque ligne pendant l'opération **MixColumns** car cette opération réalise le $\oplus$ de plusieurs octets du state entre eux, ce qui aurait pour effet d'en annuler le masquage par *m'*.

La suite de cette annexe est dédiée à l'explication des modifications apportées aux différentes opérations. La figure A.1 résume l'évolution des masques utilisés pour masquer l'état à chaque opération.

Les valeurs `m1'`, `...`, `m4'` sont calculées comme les images de `m1`, `...`, `m4` par l'étape **MixColumns**, qui est linéaire par rapport au $\oplus$. Initialement, l'état est masqué avec les masques `m1'`, `...`, `m4'`. L'opération **KeySchedule** est modifiée pour que l'opération **AddRoundKey** change ces masques en l'unique masque `m`. L'opération **SubBytes** est modifiée pour exploiter une SBox alternative appelée `SBox'` qui respecte la propriété suivante : `SBox'[x⊕m] = SBox[x]⊕m'`. Cette opération réalise donc désormais le calcul de la SBox ainsi que la conversion vers le masque `m'`. L'opération **ShiftRows** ne modifie pas les masques et reste inchangée. La nouvelle opération **ChangeMasks** réalise la conversion du masque `m'` vers les masques pour chaque ligne `m1`, `...`, `m4`. Finalement, l'opération **MixColumns** n'est pas changée mais transforme les lignes vers les valeurs masquées par `m1'`, `...`, `m4'`.

Le **KeySchedule** peut aussi être masqué afin de ne pas générer les clés de ronde en utilisant les clés sans protection.

Ce schéma de masquage propose de n'utiliser qu'un nombre très faible de masques, le rendant particulièrement adapté aux applications embarquées.

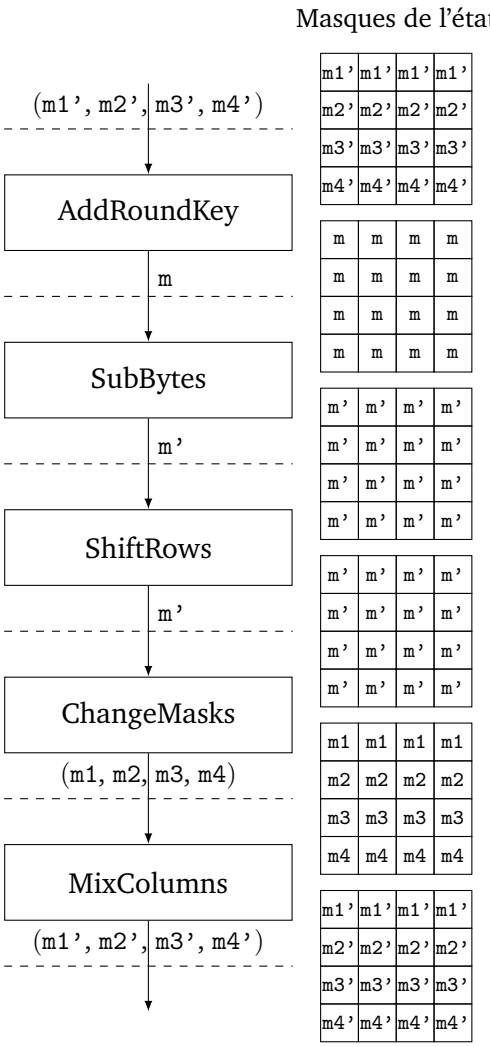


Fig. A.1 : Organisation du calcul de l'AES masqué par le schéma de Herbst. Reproduction de la figure d'un support d'enseignement de Quentin Meunier.

Fonctions de propagation dans les différents domaines

B

Cette annexe est dédiée aux différentes fonctions permettant de calculer les valeurs dans chaque domaine des signaux de sortie en fonction de leurs entrées.

Des implémentations des opérateurs de comparaison signées et non signées (EQ, NE, GT, LT, GE, LE) ont également été créées mais ne sont pas données dans cette annexe. Ces implémentations sont cependant disponibles sur le dépôt github d'`aLEAKator`.

La porte MUX qui réalise la sélection entre deux signaux de n bits `w = sel ? a : b` est implémentée comme `w = OR(AND(repeat(sel,`n`), a), AND(NOT(repeat(sel,`n`)), b)` et n'est donc pas détaillée dans cette annexe.

B.1 Domaine concret et domaine symbolique

f_g	Domaines	Entrées	Définitions de ${}^c f_g[i]$ et de ${}^e f_g[i]$
AND OR XOR	$\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$	$({}^c\alpha_{w1}, {}^c\alpha_{w2})$	${}^c\alpha_{w1}[i]\ and/or/xor\ {}^c\alpha_{w2}[i]$
	$\mathbb{E}^n \times \mathbb{E}^n \to \mathbb{E}^n$	$({}^e\alpha_{w1}, {}^e\alpha_{w2})$	OP_{AND/OR/XOR} $({}^e\alpha_{w1}[i],\ {}^e\alpha_{w2}[i])$
NOT	$\mathbb{B}^n \to \mathbb{B}^n$	$({}^c\alpha_w)$	$not\ {}^c\alpha_w[i]$
	$\mathbb{E}^n \to \mathbb{E}^n$	$({}^e\alpha_w)$	OP_NOT $({}^e\alpha_w[i])$
ADD SUB MUL	$\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$	$({}^c\alpha_{w1}, {}^c\alpha_{w2})$	${}^c\alpha_{w1}\ add/sub/mul\ {}^c\alpha_{w2}[i]$
	$\mathbb{E}^n \times \mathbb{E}^n \to \mathbb{E}^n$	$({}^e\alpha_{w1}, {}^e\alpha_{w2})$	OP_{ADD/SUB/MUL} $({}^e\alpha_{w1}[i], {}^e\alpha_{w2}[i])$
neg	$\mathbb{B}^n \to \mathbb{B}^n$	$({}^c\alpha_w)$	$(neg\ {}^c\alpha_w)[i]$
	$\mathbb{E}^n \to \mathbb{E}^n$	$({}^e\alpha_w)$	OP_NEG $({}^e\alpha_w)[i]$
trunc	$\mathbb{B}^n \to \mathbb{B}^p$	$({}^c\alpha_w)$	${}^c\alpha_w[i]$
	$\mathbb{E}^n \to \mathbb{E}^p$	$({}^e\alpha_w)$	OP_EXTRACT$(p-1, 0, {}^e\alpha_w)[i]$
rtrunc	$\mathbb{B}^n \to \mathbb{B}^p$	$({}^c\alpha_w)$	${}^c\alpha_w[i+n-p]$
	$\mathbb{E}^n \to \mathbb{E}^p$	$({}^e\alpha_w)$	OP_EXTRACT$(n-1, n-p, {}^e\alpha_w)[i]$
zext	$\mathbb{B}^n \to \mathbb{B}^p$	$({}^c\alpha_w)$	$\begin{cases} 0 & \text{si } i \geq n \\ {}^c\alpha_w[i] & \text{sinon} \end{cases}$
	$\mathbb{E}^n \to \mathbb{E}^p$	$({}^e\alpha_w)$	OP_ZEXT$((p-n), {}^e\alpha_w)[i]$
rzext	$\mathbb{B}^n \to \mathbb{B}^p$	$({}^c\alpha_w)$	$\begin{cases} 0 & \text{si } i < p-n \\ {}^c\alpha_w[i-p+n] & \text{sinon} \end{cases}$
	$\mathbb{E}^n \to \mathbb{E}^p$	$({}^e\alpha_w)$	OP_RZEXT$((p-n), {}^e\alpha_w)[i]$
sext	$\mathbb{B}^n \to \mathbb{B}^p$	$({}^c\alpha_w)$	$\begin{cases} {}^c\alpha_w[n-1] & \text{si } i \geq p \\ {}^c w[b_i] & \text{sinon} \end{cases}$
	$\mathbb{E}^n \to \mathbb{E}^p$	$({}^e\alpha_w)$	OP_SEXT$((p-n), {}^e\alpha_w)[i]$
repeat	$\mathbb{B}^1 \to \mathbb{B}^p$	$({}^c\alpha_w)$	${}^c\alpha_w[0]$
	$\mathbb{E}^1 \to \mathbb{E}^p$	$({}^e\alpha_w)$	${}^e\alpha_w[0]$
shr	$\mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^n$	$({}^c\alpha_{w1}, {}^c\alpha_{w2})$	$({}^c\alpha_{w1} >> /sshr\ {}^c\alpha_{w2})[i]$
	$\mathbb{E}^n \times \mathbb{E}^p \to \mathbb{E}^n$	$({}^e\alpha_{w1}, {}^e\alpha_{w2})$	OP_LSR/OP_ASR $({}^e\alpha_{w1}, {}^e\alpha_{w2})[i]$
shl	$\mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^n$	$({}^c\alpha_{w1}, {}^c\alpha_{w2})$	$({}^c\alpha_{w1} << {}^c\alpha_{w2})[i]$
	$\mathbb{E}^n \times \mathbb{E}^p \to \mathbb{E}^n$	$({}^e\alpha_{w1}, {}^e\alpha_{w2})$	OP_LSL $({}^e\alpha_{w1}, {}^e\alpha_{w2})[i]$

Tab. B.1 : Fonctions pour les domaines concret et symbolique.

B.2 Domaine de la stabilité

f_g	Domaines	Entrées	Définition de ${}^s f_g[i]$
AND	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i]) \vee ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(0))) \vee ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(0)))$
OR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i]) \vee ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(1))) \vee ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(1)))$
XOR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	${}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i]$
NOT	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s\alpha_w[i]$
ADD SUB MUL	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$\bigwedge_{j=0}^{n-1} ({}^s\alpha_{w1}[j] \wedge {}^s\alpha_{w2}[j])$
NEG	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^n$	$({}^s\alpha_w, {}^e\alpha_w)$	$\bigwedge_{j=0}^{n-1} ({}^s\alpha_w[j])$
TRUNC	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s\alpha_w[i]$
RTRUNC	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s\alpha_w[i+n-p]$
ZEXT	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \top & \text{si } i \geq n \\ {}^s\alpha_w[i] & \text{sinon} \end{cases}$
RZEXT	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \top & \text{si } i < p-n \\ {}^s\alpha_w[i-p+n] & \text{sinon} \end{cases}$
SEXT	$\mathbb{S}^n \times \mathbb{E}^n \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} {}^s\alpha_w[n-1] & \text{si } i \geq p \\ {}^s\alpha_w[i] & \text{sinon} \end{cases}$
REPEAT	$\mathbb{S}^1 \times \mathbb{E}^1 \rightarrow \mathbb{S}^p$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s w[0]$
SHR	$\mathbb{S}^n \times \mathbb{S}^p \times \mathbb{E}^n \times \mathbb{E}^p \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2})$	$\bigwedge_{j=0}^{n-1} ({}^s\alpha_{w1}[j] \wedge {}^s\alpha_{w2}[j])$
SHL	$\mathbb{S}^n \times \mathbb{S}^p \times \mathbb{E}^n \times \mathbb{E}^p \rightarrow \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2})$	$\bigwedge_{j=0}^{n-1} ({}^s\alpha_{w1}[j] \wedge {}^s\alpha_{w2}[j])$

Tab. B.2 : Fonctions pour le domaine de la stabilité.

B.3 Domaine des ensembles de fuite

f_g	Domaines	Entrées	Définition de ${}^l f_g[i]$
AND OR XOR	$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{w1}, {}^l\alpha_{w2}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} {}^l\alpha_{w1}[i] \cup {}^l\alpha_{w2}[i] & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$
NOT	$\mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} {}^l\alpha_{wi}[i] & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$
ADD SUB MUL	$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{w1}, {}^l\alpha_{w2}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \bigcup_{j=0}^{n-1} \left({}^l\alpha_{w1}[j] \cup {}^l\alpha_{w2}[j]\right) & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$
NEG	$\mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \bigcup_{j=0}^{n-1} \left({}^l\alpha_{wi}[j]\right) & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$
TRUNC	$\mathbb{L}^n \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	${}^l\alpha_{wi}[i]$
RTRUNC	$\mathbb{L}^n \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	${}^l\alpha_{wi}[i+n-p]$
ZEXT	$\mathbb{L}^n \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \varnothing & \text{si } i \geq n \\ {}^l\alpha_{wi}[i] & \text{sinon} \end{cases}$
RZEXT	$\mathbb{L}^n \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \varnothing & \text{si } i < p-n \\ {}^l\alpha_{wi}[i-p+n] & \text{sinon} \end{cases}$
SEXT	$\mathbb{L}^n \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} {}^l\alpha_{wi}[n-1] & \text{si } i \geq p \\ {}^l\alpha_{wi}[i] & \text{sinon} \end{cases}$
REPEAT	$\mathbb{L}^1 \times \mathbb{S}^p \times \mathbb{E}^p \to \mathbb{L}^p$	$({}^l\alpha_{wi}, {}^s\alpha_w, {}^e\alpha_w)$	${}^l\alpha_{wi}[0]$
SHR	$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{w1}, {}^l\alpha_{w2}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \bigcup_{j=0}^{i} \left({}^l\alpha_{w1}[j]\right) \cup \bigcup_{j=0}^{p-1} \left({}^l\alpha_{w2}[j]\right) & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$
SHL	$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n \times \mathbb{E}^n \to \mathbb{L}^n$	$({}^l\alpha_{w1}, {}^l\alpha_{w2}, {}^s\alpha_w, {}^e\alpha_w)$	$\begin{cases} \bigcup_{j=i}^{n-1} \left({}^l\alpha_{w1}[j]\right) \cup \bigcup_{j=0}^{p-1} \left({}^l\alpha_{w2}[j]\right) & \text{si } \neg {}^s\alpha_w[i] \\ \{{}^e\alpha_w[i]\} & \text{sinon} \end{cases}$

Tab. B.3 : Fonctions pour le domaine des ensembles de fuite.

C

Complexité de l'exploitation de la stabilité pour le banc de registres, cas du jeu d'instructions RISC-V appliqué au processeur ibex

Cette annexe discute de la complexité de garantir des propriétés de stabilité sur les instructions pour éviter les fuites sur le bus de lecture du banc de registres dans le cadre de programmes RISC-V en prenant l'exemple de l'Ibex.

Le tableau C.1 [Int26] [1] présente l'encodage des instructions au sein du jeu d'instructions 32 bits RISC-V.

Le décodage étant réalisé matériellement, l'identification de l'*opcode*, l'extraction des opérandes et les lectures du banc de registres sont réalisées systématiquement. Les valeurs lues en registre sont masquées à postériori si elles ne sont pas nécessaires pour l'instruction. Par exemple, pour le registre `rs2`, une implémentation matérielle du décodage peut systématiquement interpréter les bits 24 à 20 comme pilotant le banc de lecture de registre puis ne propager la valeur lue vers l'ALU que si l'instruction est de type `R`, `S` ou `B`. C'est le cas de l'Ibex, qui réalise en un cycle le décodage de l'instruction, la lecture du banc de registres et l'exécution de l'instruction (cas général hors mécanisme de write-back, et instructions

1. RISC-V Instruction Set Manual Volume I version 20260120, ratified, Chapter 35. RV32/64G Instruction Set Listings | Page 609 : `https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf`.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-Type
$imm[11:0]$		rs1	funct3	rd	opcode	I-Type
$imm[11:5]$	rs2	rs1	funct3	$imm[4:0]$	opcode	S-Type
$imm[12\|10:5]$	rs2	rs1	funct3	$imm[4:1\|11]$	opcode	B-Type
$imm[31:12]$				rd	opcode	U-Type
$imm[20\|10:1\|11\|19:12]$				rd	opcode	J-Type

Tab. C.1 : Encodage du jeu d'instructions RV32I.

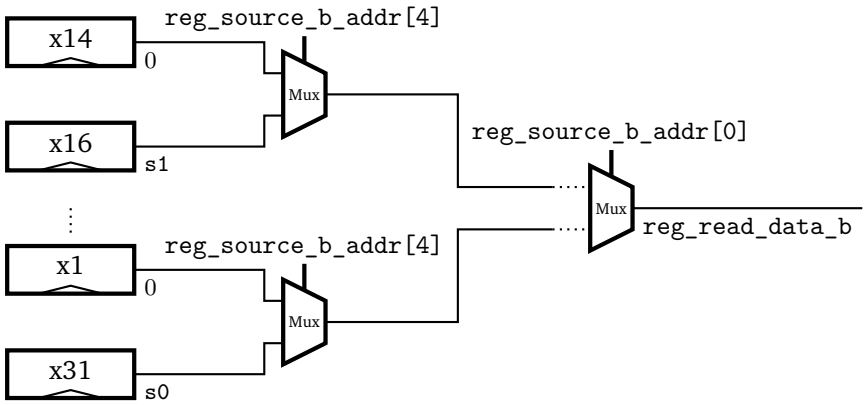


Fig. C.1 : Exemple d'un port de lecture correspondant à l'hypothèse de chargement des shares.

multi-cycle). Le listing C.a schématise le décodage et la lecture des valeurs depuis le banc de registre pour leur utilisation dans l'ALU.

```
1  logic [31:0] instr;
2  logic [4:0] reg_source_b_addr;
3  logic [31:0] imm_b;
4  logic [31:0] reg_read_data_b;
5  logic [6:0] opcode;
6  logic is_bsr;
7
8  assign reg_source_b_addr = instr[24:20];
9  assign opcode = instr[6:0];
10 assign imm_b = format_imm(instr[31:12], opcode);
11 assign reg_read_data_b = regfile[reg_source_b_addr];
12 assign is_bsr = opcode[5];
13 assign alu_b_port = (is_bsr) ? reg_read_data_b : imm_b;
```

Listing C.a : Décodage matériel des instructions par l'Ibex et lectures des données vers l'ALU.

La suite de cette section est un exemple montrant que changer l'adresse d'implantation de la section `.data` peut mener à des suites d'instructions identiques au niveau assembleur mais résultant en des verdicts de fuites différents.

Soit une valeur secrète a séparée en deux shares `s1` et `s0`, positionnés respectivement dans les registres `x16` et `x31`, avec un arbre de multiplexeurs organisé de telle manière que seule une instabilité sur tous les bits de `reg_source_b_addr` provoque une fuite en valeur avec glitches entre ces deux registres. La figure C.1 illustre un banc de registre correspondant à cette hypothèse. Soit l'instruction courante `sw a1, 0xc(t0)`, réalisant une écriture du registre `a1` à l'adresse pointée par le contenu du registre `t0`, décalée de `0xC = 12` octets. Cette instruction est encodée `0x00 0xB2 0xA6 0x23` et ses bits 24 à 20 sont donc `0b01011`.

En supposant deux placements différents de la section `.data` et l'instruction précédente qui charge le registre `t0` à la valeur souhaitée pour la lecture.

L'instruction `la t0, symbol` est une pseudo-instruction permettant de charger l'adresse d'un symbole en registre. Elle se décompose en deux instructions, soit l'adresse du symbole est trouvée dans une *GOT* (*G*lobal *O*ffset *T*able) si l'option *PIC* (pour *P*osition *I*ndependent *C*ode) est fournie, soit l'adresse directe du symbole est calculée. Ce dernier cas est le comportement par défaut et est le comportement considéré dans cet exemple. La première instruction est donc `auipc t0, symbol[31:12]` chargeant les 20 bits de poids forts de l'adresse. La deuxième est `addi t0, t0, symbol[11:0]`.

Nous supposons, par exemple en raison d'un ajout de symbole plus tôt dans la section `.data`, l'adresse de `symbol` à charger dans `t0` passe de `0x?? 0x?? 0x?1 0x74` à `0x?? 0x?? 0x?2 0x38`. Les deux `addi` sont donc encodés respectivement `0x17 0x42 0x82 0x93` et `0x23 0x82 0x82 0x93`, avec les bits 24 à 20 valant respectivement `0b10100` et `0b11000`.

Le registre qui contient l'instruction à être décodée puis exécutée pour le cycle passe donc d'une de ces deux valeurs à celle de l'écriture `sw a1, 0xc(t0)` (`0b01011`). La stabilité de signaux non symboliques, issus de registres est le `XNOR` entre ces deux valeurs, résultant en des stabilités de : `0b00000` et `0b10001` et donc en une fuite en valeur avec glitches pour le premier cas mais pas le second.

Bien qu'`aLEAKator` détecte de telles fuites, il apparaît qu'il est complexe de se reposer sur la stabilité de l'arbre de multiplexeurs pour sécuriser un programme. Tout changement d'adresse d'implantation ainsi que toute instruction exécutée (même ne manipulant aucune valeur sensible) pendant que des éléments sensibles sont chargés dans le banc de registres peuvent causer une fuite. Ces raisons justifient de vérifier à chaque cycle qu'aucune fuite ne pourrait se produire même dans l'éventualité où tous les registres du banc de registre feraient apparaître, lorsque les glitches sont pris en compte, leur valeur sur le port de lecture.

Correctifs pour le processeur Ibex

D

Le registre `x5` est réservé pour pointer vers la première adresse de la zone mémoire d'`aLEAKator` dans le programme. Contrairement au processeur ARM Cortex-M4 qui possède deux étages de pipeline différents pour le décodage et l'exécution des instructions, l'Ibex n'en possède qu'un. Ainsi, la contre-mesure émulation gel du pipeline n'est pas nécessaire.

La contre-mesure de nettoyage en lecture du bus, nettoyage bus mémoire lecture, est implémentée à l'aide de l'instruction : `lw x0, 0(x5)`.

Afin d'effectuer un nettoyage des entrées dans l'étage de décodage/exécution, la contre-mesure nettoyage ALU est implémentée avec l'instruction : `xor x0, x0, x0`.

Le concept pour réaliser l'éviction de registres est identique à celui pour le processeur ARM Cortex-M4 ; la contre-mesure éviction du contenu d'un ou plusieurs registres est donc transposée en RISC-V.

Le système expert n'a pas été totalement implémenté pour l'Ibex mais a permis avec ces trois correctifs de résoudre les fuites rencontrées sur les programmes `DOM_AND` et `Secmult`. À terme, il sera nécessaire de définir et tester des correctifs pour les contre-mesures nettoyage cellule mémoire et nettoyage registre interne mémoire.