Protection du contenu des mémoires externes dans les systèmes embarqués, aspect matériel

Salaheddine Ouaarab

▶ To cite this version:

Salaheddine Ouaarab. Protection du contenu des mémoires externes dans les systèmes embarqués, aspect matériel. Cryptographie et sécurité [cs.CR]. Télécom ParisTech, 2016. Français. NNT : 2016ENST0046 . tel-02120616

HAL Id: tel-02120616
https://pastel.hal.science/tel-02120616

Submitted on 6 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2016-ENST-0046

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Electronique et Communication »

présentée et soutenue publiquement par

Salaheddine OUAARAB

le 9 Septembre 2016

Protection du contenu des mémoires externes dans les systèmes embarqués, aspect matériel

Directeur de thèse : **Renaud PACALET**
Co-encadrement de la thèse : **Guillaume DUC**

Jury

M. Bruno ROUZEYERE, Professeur, Université de Montpellier, Montpellier	Rapporteur
M. Bruno ROBISSON, Ingénieur de recherche, CEA, Gardanne	Rapporteur
M. Jean-Luc DANGER, Directeur d'Etudes, Télécom ParisTech, Paris	Examinateur
M. Bernard KASSER, Directeur R&D sécurité, STMicroelectronics, Rousset	Examinateur
M. Gilles SASSATELLI, Directeur de Recherche CNRS, LIRMM, Montpellier	Examinateur

T H È S E

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Table des matières

Remerciements **11**

Introduction **15**

1 Éléments de cryptographie **17**
1.1 Définitions 17
1.2 Techniques de chiffrement 18
1.2.1 Chiffrement symétrique 19
1.2.1.1 Chiffrement de flux 20
1.2.1.1.1 Masque jetable (*One Time Pad*, OTP) 20
1.2.1.1.2 Chiffrement de flux synchrone 21
1.2.1.1.3 Chiffrement de flux auto-synchrone 21
1.2.1.2 Chiffrement par bloc 23
1.2.1.2.1 Structure du chiffrement par bloc 23
1.2.1.2.2 Exemple : AES 24
1.2.1.2.3 Modes d'opération 24
Electronic CodeBook (ECB) 25
Cipher-Block Chaining (CBC) 26
Cipher FeedBack (CFB) 27
Output FeedBack (OFB) 28
Mode compteur 29
1.2.1.3 Conclusion 31
1.2.2 Chiffrement asymétrique 31
1.2.3 Sécurité des techniques de chiffrement 32
1.3 Techniques de vérification d'intégrité 33
1.3.1 Fonctions de hachage 33
1.3.1.1 Code de détection de modification 34
1.3.1.1.1 Exemple : SHA 34
1.3.1.2 Code d'authentification de message 35
1.3.1.2.1 Exemple : CBC-MAC 35
1.3.2 Paradoxe des anniversaires 36
1.3.3 Conclusion 36

2 Panorama des techniques de protection **37**
2.1 Modèle de menace 38
2.2 Méthodes de protection 40

2.2.1 Confidentialité . . . 40
2.2.2 Intégrité . . . 42
2.2.2.1 Méthodes de protection contre l'attaque par injection . . . 42
2.2.2.2 Méthodes de protection contre l'attaque par permutation spatiale . . . 43
2.2.2.3 Méthodes de protection contre l'attaque par rejeu . . . 44
2.3 Les arbres de Merkle réguliers . . . 46
2.3.1 Opérations . . . 47
2.3.1.1 Initialisation . . . 47
2.3.1.2 Vérification . . . 48
2.3.1.3 Mise à jour . . . 48
2.3.2 Évaluation des performances . . . 49
2.3.2.1 Notations . . . 49
2.3.2.2 Initialisation . . . 51
2.3.2.3 Lecture vérifiée . . . 55
2.3.2.4 Écriture vérifiée . . . 56
2.3.3 Variantes des arbres de Merkle . . . 57
2.3.3.1 Parallelizable Authentication Tree . . . 57
2.3.3.2 TEC-Tree . . . 57
ReadAndCheck . . . 60
WriteAndUpdate . . . 60
2.3.4 Comparaison des différents arbres . . . 60
2.4 Plates-formes de calcul sécurisées . . . 61
2.4.1 Best . . . 61
2.4.2 Dallas DS5002FP . . . 62
2.4.3 XOM . . . 63
2.4.4 AEGIS . . . 65
2.4.5 CryptoPage . . . 66
2.4.6 MESA . . . 67
2.4.7 PE-ICE . . . 67
2.5 Conclusion . . . 68

3 Les arbres de Merkle creux et la gestion de la mémoire **71**
3.1 Les arbres creux initialisés . . . 73
3.1.1 Opérations . . . 73
3.1.1.1 Initialisation . . . 73
3.1.1.2 Vérification . . . 74
3.1.1.3 Mise à jour . . . 74
3.1.2 Évaluation des performances . . . 75
3.1.2.1 Initialisation . . . 75
3.1.2.2 Lecture vérifiée . . . 76
3.1.2.3 Écriture vérifiée . . . 76
3.2 Les arbres creux non initialisés . . . 76
3.2.1 Opérations . . . 77
3.2.1.1 Initialisation . . . 77
3.2.1.2 Vérification . . . 77

3.2.1.3 Mise à jour . . . 78
3.2.2 Évaluation des performances . . . 79
3.2.2.1 Initialisation . . . 79
3.2.2.2 Lecture vérifiée . . . 80
3.2.2.3 Écriture vérifiée . . . 80
3.3 Utilisation d'un cache d'arbres de Merkle . . . 81
3.3.1 Stratégie ASAP . . . 82
3.3.2 Stratégie ALAP . . . 82
3.3.3 Cache d'arbre de Merkle ALAP et *Write-Through* . . . 82
3.3.4 Cache d'arbre de Merkle ALAP et *Write-Back-Allocate* . . . 83
3.3.4.1 Les opérations élémentaires du cache . . . 85
3.3.4.2 Cohérence interne partielle . . . 86
3.3.4.3 Gestion du cache . . . 87
3.3.4.4 Correction fonctionnelle de la gestion du cache proposée . . 98
3.3.5 Combinaison des arbres creux et du cache d'arbre . . . 99
3.4 Conclusion . . . 100

4 Cas d'étude : projet SecBus **101**
4.1 Introduction . . . 101
4.1.1 Modèle de menace . . . 102
4.1.2 Objectifs de sécurité . . . 103
4.1.2.1 Confidentialité . . . 104
4.1.2.1.1 Pages RO . . . 104
4.1.2.1.2 Pages RW . . . 104
4.1.2.2 Intégrité . . . 105
4.1.2.2.1 Pages RO . . . 105
4.1.2.2.2 Pages RW . . . 105
4.1.3 Architecture et structures de données . . . 108
4.1.3.1 Politiques de sécurité . . . 109
4.1.3.2 Paramètres de sécurité de page . . . 110
4.1.3.3 Arbre de MAC Maître (MMT) . . . 111
4.2 Architecture logicielle . . . 111
4.2.1 Bootloader . . . 111
4.2.2 SSM . . . 112
4.2.2.1 Interface d'applications . . . 112
4.2.2.2 Chargeur d'applications . . . 113
4.2.2.3 Mémoire dynamique . . . 114
4.3 Architecture matérielle . . . 114
4.3.1 Modules d'interface . . . 115
4.3.1.1 Split Module . . . 115
4.3.1.2 Merge Module . . . 115
4.3.1.3 Input Controller . . . 116
4.3.1.4 IO Registers Module . . . 116
4.3.2 Modules de protection . . . 116
4.3.2.1 Security Context Controller . . . 116
4.3.2.2 Security Controller . . . 116

4.3.2.3 Crypto Engine . . . 117
4.3.2.4 MAC Set Controller . . . 117
4.3.2.5 MAC Tree Controller . . . 117
4.4 Conclusion . . . 118

5 Simulations et validations 119
5.1 Tests de simulation des Arbres de Merkle . . . 120
5.1.1 Environnement de simulation . . . 120
5.1.1.1 Introduction de SoCLib . . . 120
5.1.1.2 Plate-forme et application . . . 121
5.1.1.2.1 Initialisation des arbres de Merkle . . . 121
5.1.1.2.2 Configuration des paramètres de sécurité . . . 121
5.1.1.2.3 Application . . . 123
5.1.1.2.4 Paramètres de la simulation . . . 123
5.1.2 Résultats . . . 123
5.1.3 Conclusion . . . 126
5.2 Validation fonctionnelle . . . 126
5.2.1 Introduction . . . 126
5.2.2 Ce que l'on prouve . . . 126
5.2.3 Quelques précisions sur la preuve de raffinement avec la méthode $\mathcal{B}$ . . . 129
5.2.3.1 Des arbres abstraits aux arbres concrets . . . 129
5.2.3.2 De l'algorithme abstrait (fonctionnel récursif) au concret (impératif itératif) . . . 133
5.2.3.3 Preuve formelle . . . 136
5.2.4 Quelques précisions sur la preuve d'équivalence avec *EasyCrypt* . . . 136
5.2.4.1 Modèle . . . 137
5.2.4.1.1 Formalisation des constructions . . . 137
5.2.4.1.2 Modèle calculatoire . . . 138
5.2.4.2 Preuve . . . 139
5.2.5 Conclusion . . . 142
5.3 Évaluation des performance du module matériel HSM . . . 143
5.3.1 Introduction . . . 143
5.3.2 Coûts matériels . . . 143
5.3.3 Augmentation de l'empreinte mémoire . . . 144
5.3.4 Configurations et résultats . . . 145
5.3.4.1 Configurations . . . 145
5.3.4.2 Résultats . . . 147
5.3.5 Conclusion . . . 153

Conclusions et perspectives 155

Appendices 157

Table des figures

1.1 Chiffrement symétrique 19
1.2 Chiffrement asymétrique 20
1.3 Chiffrement de flux synchrone 22
1.4 Chiffrement de flux auto-synchrone 23
1.5 Structure du chiffrement par bloc 24
1.6 Chiffrement par bloc en mode ECB 25
1.7 Chiffrement par bloc en mode CBC 27
1.8 Chiffrement par bloc en mode CFB 28
1.9 Chiffrement par bloc en mode OFB 29
1.10 Chiffrement par bloc en mode compteur 30
1.11 Fonction de hachage 34
1.12 Fonction de HMAC 35
1.13 Calcul de CBC-MAC 35

2.1 Zones d'attaques considérées. 39
2.2 Attaque par *injection* 40
2.3 Attaque par *permutation spatiale* 40
2.4 Attaque par *rejeu* 40
2.5 Ajout du module cryptographique à l'intérieur du SoC. 41
2.6 Protection d'une zone mémoire contre les attaques par injection 43
2.7 Protection d'une zone mémoire contre les attaques par injection 43
2.8 Calcul du CBC-MAC avec adresse 44
2.9 Protection de la mémoire externe avec des MAC et `nonce` 45
2.10 Protection de la mémoire externe avec la technique AREA 45
2.11 Arbre de Merkle 46
2.12 Arbre de Merkle protégeant l'intégrité d'un ensemble de données 47
2.13 Arbre de Merkle déséquilibré protégeant l'intégrité d'un ensemble de données . 51
2.14 Comparaison entre le temps pris pour une lecture d'un groupe de nœuds, un calcul de MAC et un écriture d'un nœud 52
2.15 Latences des différentes opérations de l'initialisation d'arbre 52
2.16 Latences des différentes opérations d'une lecture vérifiée 55
2.17 Latences des différentes opérations d'une écriture vérifiée 56
2.18 *Parallelizable Authentication Tree* (PAT) 58
2.19 Configuration d'un *DC* avant le chiffrement 58
2.20 Configuration d'un CC avant le chiffrement 59
2.21 *Tamper-Evident Counter Tree* (TEC-Tree) 59

2.22 Architecture de DS5002FP 62
2.23 Architecture de XOM 64
2.24 Principe de fonctionnement de PE-ICE 68

3.1 AMC protégeant l'intégrité d'un ensemble de données 72
3.2 Différents états d'un AMC-I 73
3.3 Différents états d'un AMC-NI 77
3.4 Politique d'écriture Write-Through 83
3.5 Politique d'écriture Write-Back 84
3.6 Système étudié avec ou sans cache d'arbre de Merkle 84
3.7 AMCached CIP 87

4.1 Zones d'attaques considérées par SecBus. 103
4.2 Protection des pages RO en intégrité par des MAC. 106
4.3 Protection des pages RW en intégrité par des arbres de MAC. 107
4.4 Les différents types de pages mémoires 108
4.5 Le HSM à l'intérieur du SoC 109
4.6 Présentation de la mémoire externe avec les pages d'intégrité et le MB 111
4.7 Architecture logicielle de SecBus 113
4.8 Architecture interne du HSM 115

5.1 Principe d'un protocole VCI 121
5.2 Plate-Forme matérielle de SoCLib avec le HSM 122
5.3 Protection d'intégrité par des arbres de Merkle 127
5.4 Les deux étapes d'une preuve 128
5.5 Modèle d'arbres 129
5.6 Exemple d'un prototype HSM sur une carte ZedBoard à base de FPGA Zynq. . . 144
5.7 Linux init start 147
5.8 Linux init duration 148
5.9 Dhrystone 148
5.10 Whetstone 149
5.11 RAMSMP 1 Gio, 2 processus, écriture, blocs de 1 kio 149
5.12 RAMSMP 1 Gio, 2 processus, écriture, blocs de 32 Mio 150
5.13 RAMSMP 1 Gio, 2 processus, lecture, blocs de 1 kio 150
5.14 RAMSMP 1 Gio, 2 processus, lecture, blocs de 32 Mio 151
5.15 RAMSpeed 1 Gio, 1 process, écriture, blocs de 1 kio 151
5.16 RAMSpeed 1 Gio, 1 processus, écriture, blocs de 32 Mio 152
5.17 RAMSpeed 1 Gio, 1 processus, lecture, blocs de 1 kio 152
5.18 RAMSpeed 1 Gio, 1 processus, lecture, blocs de 32 Mio 153

Liste des tableaux

2.1 Résumé des propriétés des arbres existants [24] 61

5.1 Résultats de la simulation sans cache 124
5.2 Impactes du seuil sur les performances du cache 124
5.3 Résultats de la simulation avec cache 124
5.4 Résultats de la simulation sans cache 125
5.5 Résumé de l'utilisation des ressources matérielles. 144

Remerciements

Je commence la présentation de ce travail par le remerciement de tous ceux qui ont aidé et facilité à sa réalisation. Je m'adresse à mes deux directeurs de thèse Monsieur Renaud PACALET et Monsieur Guillaume DUC pour leur signaler combien je suis sensible à leurs qualités. Monsieur Renaud PACALET, sa disponibilité, ses qualités humaines et ses compétences professionnelles resteront pour moi un exemple à suivre. Il m'a dirigé efficacement tout au long de ce travail, ainsi que dans bien d'autres circonstances. Monsieur Guillaume DUC, sa contribution, sa disponibilité, son sérieux et ses connaissances ont permis à mon travail d'avoir plus de valeur.

Je remercie Bruno ROUZEYERE, Bruno ROBISSON, Jean-Luc DANGER, Bernard KASSER, et Gilles SASSATELLI de me faire l'honneur de constituer mon jury de thèse.

Mes remerciements vont tout particulièrement à Jérémie BRUNEL, qui m'a aidé au début de la thèse à comprendre l'architecture de SecBus et à utiliser certains outils de travail, et aussi à Abdelmalek SI MERABET, pour son aide et ses conseils pertinents dans la partie d'expérimentations matérielles. Je remercie également, très chaleureusement, Rabea AMEUR-BOULIFA et Sophia COUDERT pour m'avoir aidé à réaliser la vérification fonctionnelle des algorithmes.

Je tiens à remercier tous mes collègues du département COMELEC à Paris et du laboratoire LabSoC à Sophia Antipolis, Sylvain GUILLEY, Yves MATHIEU, Laurent SAUVAGE, Tarik GRABA, Ludovic APVRILLE, Tullio TANZI, Daniel CAMARA, Andrea ENRICI, Hocine MOKRANI, Bassem OUNI, Xuan Thuy NGO, Zakaria NAJM, Youssef SUISSI, Taoufik CHOUTA, Annelie HEUSER, Shivam BHASIN, Houssem MAGHREBI, Sebastien THOMAS, Zouha CHERIF, Pablo RAUZy, Molka BEN ROMDHANE, et tous les membres de Secure-IC pour les différents échanges concernant mon sujet de thèse et pour l'intérêt qu'ils ont porté à mon travail. J'ai été très touché par leur disponibilité et leurs conseils précieux, pour leur dire que je vous serais sincèrement reconnaissant.

Durant ces années passées dans notre chère école, j'ai bénéficié d'un corps administratif toujours présent pour m'aider et me soutenir très chaleureusement. Ceci est l'occasion de leur présenter ma sincère et amicale considération. Je cite Mesdames Chantal Cadiat, Yvonne BANSIMBA, Florence Besnard, Chantal GUIZOL (Eurecom), Zouina Sahnoun, Nazha Essakkaki et Pascale CASTAING (Eurecom) ; et Messieurs Hamidou YAYA KONÉ, Bruno Thedrez, Philippe FOUBERT (Eurecom), Franck HEURTEMATTE (Eurecom) et Dominique ROUX.

Enfin, ces remerciements ne seraient complets sans mentionner les personnes que j'aime le plus au monde, ma mère Aicha et mon père Allali, qui ont toujours su me témoigner leur amour inconditionnel et leur soutien continu, et bien évidemment à mes frères et sœurs, Adil,

Btissam, Aziz et Wiam qui malgré la distance ont toujours marqué leur présence et leur soutien. Mes remerciements vont également à toute la famille OUAARAB et KHOUYA.

Résumé

Ces dernières années, les systèmes informatiques (*Cloud Computing*, systèmes embarqués, etc.) sont devenus omniprésents. La plupart de ces systèmes utilisent des espaces de stockage (flash, RAM, etc.) non fiables ou non dignes de confiance pour stocker du code ou des données. La confidentialité et l'intégrité de ces données peuvent être menacées par des attaques matérielles (espionnage de bus de communication entre le composant de calcul et le composant de stockage) ou logicielles. Ces attaques peuvent ainsi révéler des informations sensibles à l'adversaire ou perturber le bon fonctionnement du système. Dans cette thèse, nous nous sommes focalisés, dans le contexte des systèmes embarqués, sur les attaques menaçant la confidentialité et l'intégrité des données qui transitent sur le bus de communication avec la mémoire ou qui sont stockées dans celle-ci.

Plusieurs primitives de protection de confidentialité et d'intégrité ont déjà été proposées dans la littérature, et notamment les arbres de Merkle, une structure de données protégeant efficacement l'intégrité des données notamment contre les attaques par rejeu. Malheureusement, ces arbres ont un impact important sur les performances et sur l'empreinte mémoire du système.

Dans cette thèse, nous proposons une solution basée sur des variantes d'arbres de Merkle (arbres creux) et un mécanisme de gestion adapté du cache afin de réduire grandement l'impact de la vérification d'intégrité d'un espace de stockage non fiable. Les performances de cette solution ont été évaluées théoriquement et à l'aide de simulations. De plus, une preuve est donnée de l'équivalence, du point de vue de la sécurité, avec les arbres de Merkle classiques.

Enfin, cette solution a été implémentée dans le projet SecBus, une architecture matérielle et logicielle ayant pour objectif de garantir la confidentialité et l'intégrité du contenu des mémoires externes d'un système à base de microprocesseurs. Un prototype de cette architecture a été réalisé et les résultats de l'évaluation de ce dernier sont donnés.

Abstract

During the past few years, computer systems (*Cloud Computing*, embedded systems...) have become ubiquitous. Most of these systems use unreliable or untrusted storage (flash, RAM...) to store code or data. The confidentiality and integrity of these data can be threaten by hardware (spying on the communication bus between the processing component and the storage component) or software attacks. These attacks can disclose sensitive information to the adversary or disturb the behavior of the system. In this thesis, in the context of embedded systems, we focused on the attacks that threaten the confidentiality and integrity of data that are transmitted over the memory bus or that are stored inside the memory.

Several primitives used to protect the confidentiality and integrity of data have been proposed in the literature, including Merkle trees, a data structure that can protect the integrity of data including against replay attacks. However, these trees have a large impact on the performances and the memory footprint of the system.

In this thesis, we propose a solution based on variants of Merkle trees (hollow trees) and a modified cache management mechanism to greatly reduce the impact of the verification of the integrity. The performances of this solution have been evaluated both theoretically and in practice using simulations. In addition, a proof a security equivalence with regular Merkle trees is given.

Finally, this solution has been implemented in the SecBus architecture which aims at protecting the integrity and confidentiality of the content of external memories in an embedded system. A prototype of this architecture has been developed and the results of its evaluation are given.

Introduction

À notre époque, les systèmes numériques sont devenus omniprésents dans notre vie quotidienne. Ils sont utilisés dans plusieurs domaines, comme le *Cloud Computing*, les systèmes embarqués ou les bases de données. Ces systèmes manipulent des informations sensibles qui sont stockées dans une zone mémoire non nécessairement fiable. Par conséquent, la protection de données résidant dans un espace de stockage non fiable ou non digne de confiance est un problème critique du point de vue de la sécurité.

Malheureusement, la sécurité de ces systèmes est vulnérable à certaines attaques telles que les exploitations logicielles et les attaques matérielles. Parmi les zones critiques de ces systèmes on peut citer le bus de communication entre le ou les composants de calcul (processeurs, SoC) et les composants de stockage (la mémoire vive, le disque dur, etc.). Un intrus accédant physiquement à ces systèmes, s'il parvient à espionner le bus de communication, peut récupérer le code exécuté et les données manipulées et obtenir ainsi de l'information potentiellement sensible (attaques passives). Il peut aussi modifier les informations qui transitent sur le bus ou le contenu des mémoires (attaques actives), ce qui peut perturber le bon fonctionnement du système.

Les utilisateurs de services en ligne, par exemple, s'attendent à ce que leurs données sensibles soient stockées et protégées dans un système informatique de confiance appartenant au fournisseur de services. Cependant, si cette hypothèse de confiance n'est pas fondée, et que des données sensibles sont divulguées, la réputation du fournisseur de services peut être sévèrement compromise. Le scandale provoqué par la vente sur eBay [1] d'un ordinateur ayant précédemment appartenu à une banque et contenant des données concernant des clients de la banque est une bonne illustration des conséquences possibles d'un défaut de protection de données sensibles.

À première vue, ceci semble être uniquement un problème de confidentialité et de contrôle d'accès aux données, que les techniques classiques de chiffrement, convenablement appliquées, devraient pouvoir résoudre. Cependant, lorsqu'un attaquant obtient un accès à un système informatique manipulant des données chiffrées, il peut faire plus qu'observer passivement le système. Il peut également altérer les données manipulées, même chiffrées et, dans certains cas, modifier ainsi le comportement du système afin de le forcer à révéler des informations sensibles. Dans [2], par exemple, un attaquant altère l'exécution du logiciel en modifiant à l'aveugle des instructions et des données chiffrées jusqu'à obtenir l'accès en clair à l'ensemble du système. Le chiffrement seul ne suffit pas toujours car il ne peut garantir que la confidentialité, pas l'intégrité.

Dans cette thèse, nous nous sommes focalisés sur les attaques menaçant la confidentialité et l'intégrité des données qui transitent sur le bus de communication avec la mémoire ou qui sont stockées dans celle-ci. Dans ce contexte, plusieurs plates-formes de calcul sécurisées (Best, XOM, AEGIS, SecBus etc.) ont été proposées afin de contrer ces menaces et d'assurer la confidentialité et / ou l'intégrité du contenu des mémoires dans les systèmes embarqués.

Comme nous le verrons, la protection de l'intégrité constitue le problème le plus difficile, essentiellement à cause de la dégradation des performances qu'elle entraîne pour le système protégé. Les arbres de Merkle sont une solution bien connue pour tenter de réduire cet impact négatif en transformant un surcout linéaire (en la taille des données à protéger) en surcout logarithmique. Ils permettent de détecter les altérations de données causées par des attaques actives, y compris les attaques par rejeu, et sont utilisés, par exemple, pour protéger des bases de données ou des informations stockées sur disques durs. Cependant, appliqués dans leur version standard à la protection des mémoires externes d'un système informatique, ils dégradent tout de même significativement les performances. Nous avons étudié cette technique de contrôle d'intégrité et proposé des variantes afin de réduire l'impact sur les performances.

Cette thèse s'inscrit dans le cadre du projet SecBus, qui se différencie des autres plates-formes de calcul sécurisées de la littérature par un certain nombre de principes fondateurs particuliers que nous présenterons. C'est donc l'architecture matérielle et logicielle SecBus qui nous servira de fil conducteur et d'exemple tout au long de ce manuscrit. Les contributions présentées, cependant, sont assez générales pour être adaptées aux autres architectures.

Dans un premier temps, nous présenterons les différents algorithmes de base de la cryptographie dont nous aurons besoin tout au long de ce manuscrit (chapitre 1). Nous introduirons ensuite le modèle de menace et les mécanismes de protection proposés dans la littérature, puis, nous étudierons en profondeur la méthode des arbres de Merkle et présenterons un panorama des plates-formes de calcul sécurisées intégrant des techniques de protection des mémoires externes (chapitre 2).

Dans un deuxième temps, nous proposerons une nouvelle méthode de protection basée sur les arbres de Merkle et étudierons l'impact d'utilisation de la mémoire cache sur la gestion de cohérence des arbres de Merkle (chapitre 3). Ensuite, nous présenterons l'architecture de SecBus qui intègre les contributions proposées (chapitre 4).

Enfin, dans le chapitre 5 nous présenterons les simulations effectuées pour vérifier que les méthodes de protection proposées fonctionnent correctement et pour les comparer, ainsi que les tests réalisés sur l'architecture de SecBus. Dans la deuxième partie du chapitre, nous appliquerons des techniques de modélisation formelle et de preuve sur les algorithmes utilisés dans la gestion des arbres de Merkle.

Chapitre 1

Éléments de cryptographie

Sommaire

1.1 Définitions **17**
1.2 Techniques de chiffrement **18**
1.2.1 Chiffrement symétrique 19
1.2.2 Chiffrement asymétrique 31
1.2.3 Sécurité des techniques de chiffrement 32
1.3 Techniques de vérification d'intégrité **33**
1.3.1 Fonctions de hachage 33
1.3.2 Paradoxe des anniversaires 36
1.3.3 Conclusion 36

La *cryptographie* est l'ensemble des techniques permettant de protéger des messages, c'est-à-dire de garantir certaines propriétés de sécurité tels que la confidentialité, l'intégrité, etc. À l'opposé, la *cryptanalyse* a pour objectif de casser ces propriété de sécurité. Ce sont deux disciplines de la *cryptologie*, la science du secret [3].

Dans ce chapitre, nous présenterons les différents algorithmes de base de la cryptographie dont nous aurons besoin par la suite tout au long de ce manuscrit.

1.1 Définitions

La cryptographie conçoit des primitives de sécurité afin de garantir quatre objectifs fondamentaux [4] : la confidentialité, l'intégrité, l'authentification et la non répudiation.

Confidentialité : garantir qu'une information n'est accessible qu'à ceux dont l'accès est autorisé.

Intégrité : empêcher une altération non autorisée (volontaire ou non) d'une information, ou être capable de détecter cette altération.

Authentification : vérifier l'identité des entités participant à un protocole afin d'autoriser l'accès à des ressources (réseaux, systèmes, applications, etc.).

Non répudiation : empêcher une entité de répudier une action déjà effectuée.

Auguste Kerckhoffs en 1883 [5,6] a énoncé plusieurs principes (connus maintenant sous le nom *principes de Kerckhoffs*) que doit respecter un bon système de chiffrement pour pouvoir être utilisé sur le terrain militaire :

1. Le système doit être matériellement, sinon mathématiquement, indéchiffrable ;
2. Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi ;
3. La clé doit pouvoir en être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants ;
4. Il faut qu'il soit applicable à la correspondance télégraphique ;
5. Il faut qu'il soit portatif, et que son maniement ou son fonctionnement n'exige pas le concours de plusieurs personnes ;
6. Enfin, il est nécessaire, vu les circonstances qui en commandent l'application, que le système soit d'un usage facile, ne demandant ni tension d'esprit, ni la connaissance d'une longue série de règles à observer.

Certains de ces principes, pourtant énoncés il y a plus d'un siècle, sont toujours d'actualité. Le principe numéro 2 est ainsi devenu un axiome de base dans la conception d'un bon algorithme de cryptographie : l'adversaire doit pouvoir connaître tous les détails de l'algorithme sans que la sécurité du système l'utilisant (reposant sur un secret nommé clé) ne soit remise en cause.

L'adversaire (aussi nommé attaquant ou cryptanalyste) essaie quant à lui de casser les propriétés de sécurité en menant des attaques. Ces attaques peuvent être passives (il écoute simplement les échanges et essaie de récupérer de l'information normalement secrète) ou actives (l'adversaire va altérer les échanges).

1.2 Techniques de chiffrement

Le *chiffrement* (noté E) est un processus qui transforme un message en clair (noté P) en un message chiffré (noté C) en utilisant un secret (nommé clé et noté K_e) :

$$E_{K_e}(P) = C \tag{1.1}$$

Le *déchiffrement* (noté D) est l'opération qui permet de récupérer le message en clair à partir d'un message chiffré en utilisant une clé (notée K_d) :

$$D_{K_d}(C) = P \tag{1.2}$$

Le message chiffré n'apporte aucune information sur le message en clair pour quelqu'un qui ne connaît pas le secret nécessaire pour le déchiffrer (K_d). Le chiffrement est donc utilisé principalement pour garantir la confidentialité d'une information.

Il existe deux catégories d'algorithmes de chiffrement :

— Les algorithmes symétriques pour lesquels $K_e = K_d = K$. La clé unique K, souvent nommée clé secrète, est utilisée pour le chiffrement et le déchiffrement. Dans la figure 1.1, Alice et Bob veulent communiquer entre eux à travers une canal non sécurisée. Ils choisissent tout d'abord une clé secrète K. Ensuite, Alice chiffre le message en clair à l'aide d'un algorithme de chiffrement symétrique et la clé K, et envoie le message chiffré à Bob. Bob reçoit le message chiffré et utilise la clé K afin de retrouver le message en clair.

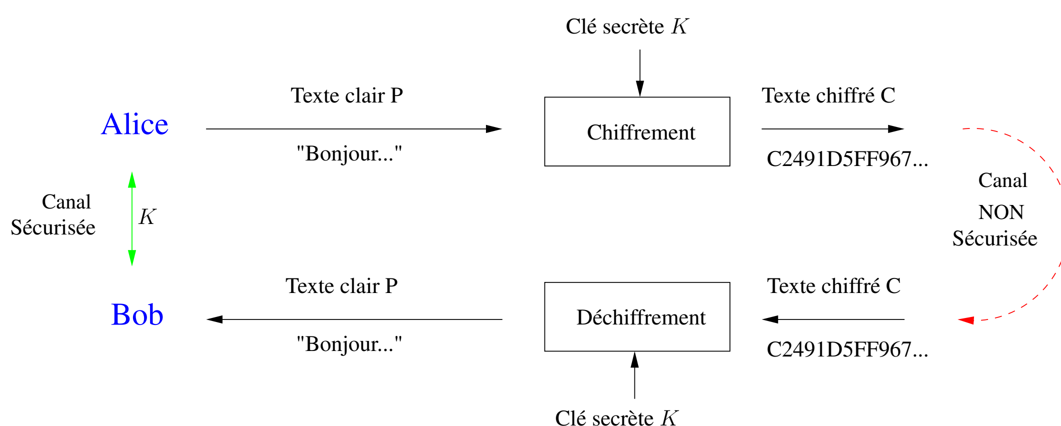


FIGURE 1.1 – Chiffrement symétrique

— Les algorithmes asymétriques pour lesquels $K_e \neq K_d$. Il y a donc deux clés différentes : une dite publique (qui peut être connue de tous) utilisée par l'algorithme de chiffrement (K_e), et une dite privée (mathématiquement associée à la clé publique) utilisée par l'algorithme de déchiffrement (K_d). Reprenons l'exemple d'Alice et Bob, présenté par la figure 1.2, où Alice veut envoyer un message confidentiel à Bob. Tout d'abord, Bob doit générer une paire de clés publique et privée (K_e,K_d) et transmettre la clé publique K_e à Alice à travers le canal non sécurisée. Ensuite, Alice chiffre son message avec K_e et l'envoie à Bob. Enfin, Bob utilise K_d pour déchiffrer le message.

Chacune de ces catégories a des avantages et des inconvénients que nous présenterons par la suite.

1.2.1 Chiffrement symétrique

Les algorithmes de chiffrement symétriques se décomposent en deux catégories : les algorithmes de chiffrement de flux et les algorithmes de chiffrement par bloc.

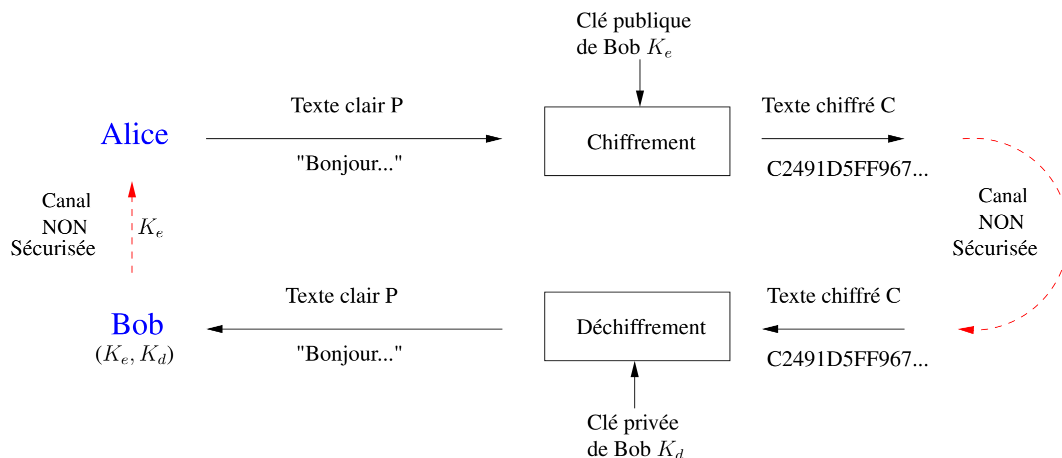


FIGURE 1.2 – Chiffrement asymétrique

1.2.1.1 Chiffrement de flux

Le concept de chiffrements de flux a été introduit par Gilbert Vernam [7] en 1917. Ces algorithmes permettent de chiffrer un flot de symboles en clair (par exemple un flux d'octets ou même un flux de bits) en utilisant un flot de clés (préparé au préalable) de la même taille.

Le déchiffrement s'effectue de la même façon, symbole par symbole, pour récupérer le message en clair à partir du flux chiffré et du même flot de clés.

Un algorithme de chiffrement de flux est composé de deux parties : un générateur de flux de clés et un algorithme réalisant le chiffrement d'un symbole en clair à l'aide d'un symbole de clé (par exemple un simple opérateur *ou exclusif*). Le générateur de flux de clés fournit un flux de bits $k_1, k_2, ..., k_n$, et l'opérateur xor réalise l'opération de chiffrement en combinant les bits de la clé (appelées souvent masque) avec les bits du message en clair $p_1, p_2, ..., p_n$, pour donner les bits du message chiffré $c_1, c_2, ..., c_n$, c'est-à-dire :

$$c_i = p_i \oplus k_i \quad \forall i \in [1, n] \tag{1.3}$$

Dans ce cas, l'opération de déchiffrement s'effectue de la même manière :

$$p_i = c_i \oplus k_i \quad \forall i \in [1, n] \tag{1.4}$$

1.2.1.1.1 Masque jetable (*One Time Pad*, OTP) Le masque jetable est une technique du chiffrement de flux qui combine le message en clair avec une clé présentant les caractéristiques suivantes : sa taille doit être au moins aussi longue que le message en clair, elle doit être totalement aléatoire et elle ne doit être utilisée qu'une seule fois (d'où le nom de masque jetable).

Claude Shannon [8] a prouvé, en 1949, qu'en respectant ces trois règles concernant la clé, le système offre une sécurité théorique absolue. Par contre, la mise en œuvre du masque jetable est difficile en pratique. En effet, la génération de flux de clés réellement aléatoires nécessite

des moyens complexes. De même, la longueur et l'utilisation unique de la clé sont deux points difficiles qui rendent l'implémentation du masque jetable inabordable.

Dans la pratique, les algorithmes de chiffrement de flux s'inspirent du masque jetable en générant le flux de clés à l'aide d'un algorithme de génération de nombres pseudo-aléatoires utilisant une graine (clé) de petite taille. Cependant, ces algorithmes ne sont plus inconditionnellement sûrs vu que la clé n'est plus réellement aléatoire. Par la suite, nous distinguerons deux catégories d'algorithmes de chiffrement de flux : les algorithmes de chiffrement de flux synchrones et les algorithmes de chiffrement de flux auto-synchrones.

1.2.1.1.2 Chiffrement de flux synchrone Un algorithme de chiffrement de flux synchrone est un algorithme où la génération du masque est indépendante du message en clair ou du message chiffré. Le masque est généré à partir d'une clé secrète k_i et d'un état interne σ_i, comme le montre les équations suivantes :

$$\sigma_{i+1} = f(\sigma_i, k) \tag{1.5}$$

$$z_i = g(\sigma_i, k) \tag{1.6}$$

où σ_0 est l'état initial de l'algorithme qui peut dépendre de la clé k, f la fonction qui permet de calculer l'état suivant, et g la fonction qui calcule le masque z_i. L'opération de chiffrement est présentée avec l'équation suivante :

$$c_i = h(z_i, p_i) \tag{1.7}$$

où h est la fonction de sortie (en général l'opérateur *ou exclusif*) qui combine le masque et le message en clair p_i pour produire le message chiffré c_i. La figure 1.3 présente le schéma de chiffrement et de déchiffrement.

L'émetteur et le récepteur doivent être synchronisés. En effet, une perte de synchronisation (insertion ou suppression d'un bit par exemple dans le flux chiffré) empêche tout déchiffrement futur. En revanche, une modification d'un bit dans le texte chiffré n'affecte pas le déchiffrement des autres bits du message chiffré.

Un avantage de cette catégorie du chiffrement de flux concerne la propagation d'erreur. Si une erreur s'est produite dans un ou plusieurs bits durant la transmission, seul le résultat du déchiffrement des bits modifiés sera impacté. Malheureusement, d'un point de vue de la sécurité, cet avantage devient un inconvénient. En effet, un adversaire qui modifie des bits dans le message chiffré est capable de connaître l'impact sur le message en clair.

1.2.1.1.3 Chiffrement de flux auto-synchrone Un algorithme de chiffrement de flux auto-synchrone est un algorithme où le masque est généré à partir d'une clé secrète k_i et d'un nombre

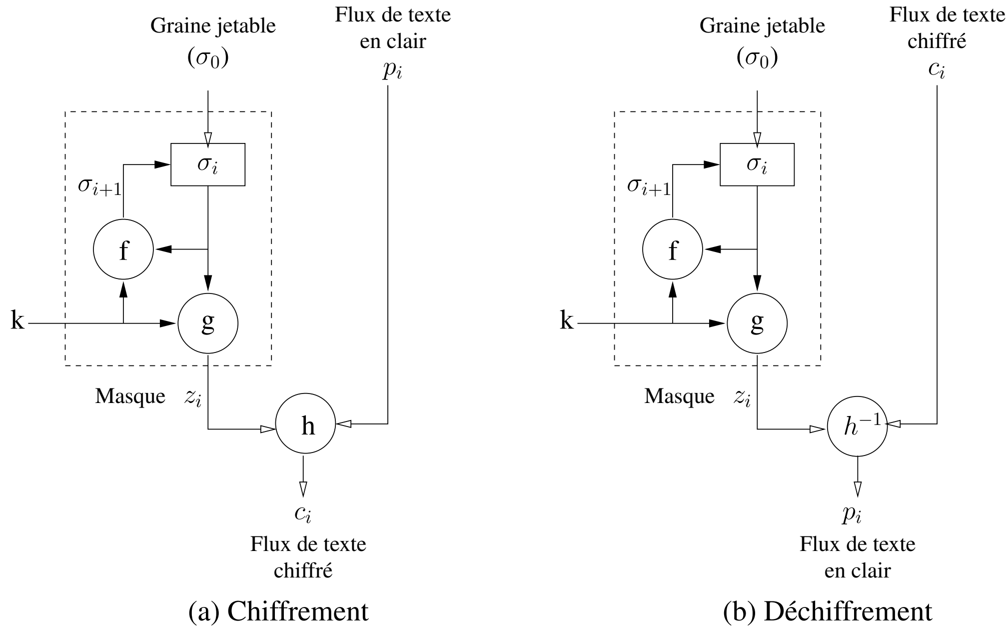


FIGURE 1.3 – Chiffrement de flux synchrone

fixe de bits des messages chiffrés précédemment. Les équations suivantes décrivent les opérations de chiffrement :

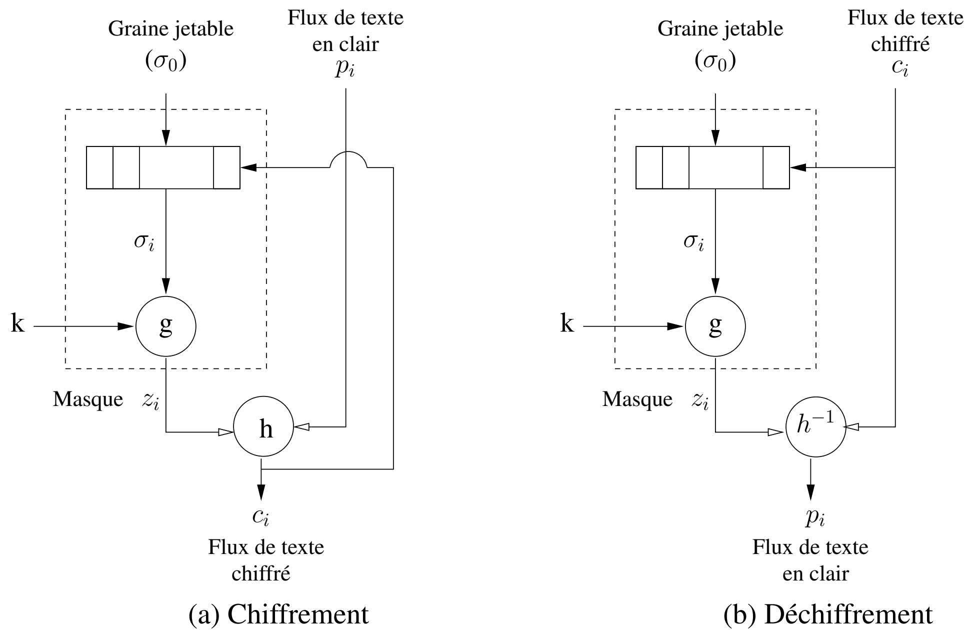
$$\sigma_i = (c_{i-t}, c_{i-t+1}, ..., c_{i-1}) \tag{1.8}$$

$$z_i = g(\sigma_i, k) \tag{1.9}$$

$$c_i = h(z_i, p_i) \tag{1.10}$$

où $\sigma_0 = (c_{-t}, c_{-t+1}, ..., c_{-1})$ est l'état initial de l'algorithme, k est la clé, g la fonction qui calcule le masque z_i et h est la fonction de sortie (en général un opérateur *ou exclusif*) qui combine le masque et le message en clair p_i pour produire le message chiffré c_i. La figure 1.4 présente le schéma de chiffrement et de déchiffrement.

La synchronisation entre l'émetteur et le récepteur est automatique (d'où le nom d'algorithme auto-synchrone). En effet, durant le déchiffrement et après la réception de n bits du chiffré, le générateur de flux de clés sera automatiquement synchrone avec le générateur de flux de clés utilisé durant le chiffrement. L'évolution de l'état interne du générateur de flux de clés ne dépend que des n bits du texte chiffré précédemment et donc le déchiffrement est automatiquement correct au bout d'un laps de temps en cas de suppression ou d'insertion d'un bit dans le texte chiffré. De même, si une erreur s'est produite dans un ou plusieurs bits durant la transmission, seul le déchiffrement des bits modifiés sera impacté et donnera des résultats erronés (le nombre de bits impactés est égal au nombre de bits utilisés comme état interne).



(a) Chiffrement (b) Déchiffrement

FIGURE 1.4 – Chiffrement de flux auto-synchrone

1.2.1.2 Chiffrement par bloc

Un algorithme de chiffrement par bloc prend en entrées un message en clair de taille fixe, une clé et produit un message chiffré de taille fixe (en général de la même taille que celle du message en clair). La taille du message en clair et du message chiffré est un paramètre de l'algorithme, par exemple 64 bits pour DES, 128 bits pour AES.

1.2.1.2.1 Structure du chiffrement par bloc Un algorithme de chiffrement par bloc est divisé en deux parties : le générateur de clés (*key schedule*) et le chemin de données (figure 1.5). Le chemin de données inclut deux opérations de base introduites par Shannon [8] qui sont la *confusion* (c'est-à-dire rendre la relation entre la clé et le texte chiffré le plus complexe possible) et la *diffusion* (c'est-à-dire permettre à chaque bit de texte clair d'avoir une influence sur une grande partie du texte chiffré). Ce chemin de données est le plus souvent composé d'une fonction (ou un ensemble de fonctions) appelée un tour (*round*) qui se répète un nombre fixe de fois. Il prend le texte en clair en entrée, et délivre en sortie le texte chiffré après le nombre fixe de tours.

Le générateur de clés traite une clé secrète et en déduit les sous-clés de tours utilisées à chaque tour du chemin de données. La dérivation de la clé est nécessaire pour ajouter de la confusion : elle augmente la dépendance de chaque bit du texte chiffré sur tous les bits de la clé secrète.

Le premier tour du chemin de données prend le texte en clair et la clé du premier tour comme entrées, puis les tours suivants prennent la sortie du tour précédent et la clé du tour correspondant comme entrées.

Le paragraphe suivant décrit l'algorithme de chiffrement par bloc le plus utilisé actuelle-

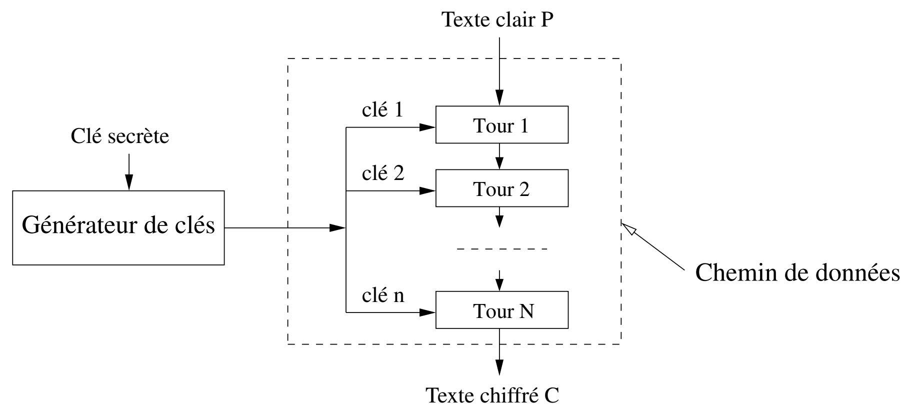


FIGURE 1.5 – Structure du chiffrement par bloc

ment et standardisé par le NIST (*National Institute of Standard and Technology*) : AES (*Advanced Encryption Standard*).

1.2.1.2.2 Exemple : AES L'algorithme Rijndael [9] a été développé par Joan Daemen et Vincent Rijmen suite à un appel de NIST afin de normaliser le chiffrement par bloc. Il a été adopté comme nouvelle norme après un processus de normalisation de cinq ans. Il a été ensuite appelé AES (*Advanced Encryption Standard*) et a remplacé le célèbre algorithme DES (*Data Encryption Standard*) [10] pour lequel la longueur de clé (56 bits) est devenue trop petite pour résister à une attaque par force brute [11] (cette attaque consiste à essayer successivement toutes les clés possibles).

L'algorithme AES traite des blocs de données de 128 bits en utilisant des clés de chiffrement de longueur de 128, 192 ou 256 bits. Un tour dans AES est défini par une série d'opérations (*AddRoundKey*, *SubBytes*, *ShiftRows* et *MixColumns*). Le nombre de tours utilisés dépend de la taille de la clé : 10 dans le cas d'une clé de 128 bits, 12 pour une clé de 192 bits et 14 pour une clé de 256 bits.

Un des objectifs d'AES était d'être implémentable efficacement aussi bien en logiciel qu'en matériel. De nombreuses implémentations matérielles ont été proposées aussi bien sur cible FPGA qu'ASIC. Par exemple, l'implémentation d'AES sur ASIC (technologie $0.18\mu m$ CMOS), présentée dans [12], montre que le chiffrement d'un bloc de 128 bits est réalisé en 11 cycles d'horloges à une fréquence de 330 MHz, ce qui a donné un débit de 3.84 Gbits/s.

1.2.1.2.3 Modes d'opération Un mode d'opération décrit la construction permettant de chiffrer un message dont la taille est différente de celle de l'algorithme de chiffrement par bloc choisi. De nombreux modes d'opération existent [13]. Nous allons présenter les principaux : ECB, CBC, CFB, OFB et compteur.

Nous utiliserons les notations suivantes :
— n est la taille de bloc de l'algorithme choisi
— K est la clé secrète

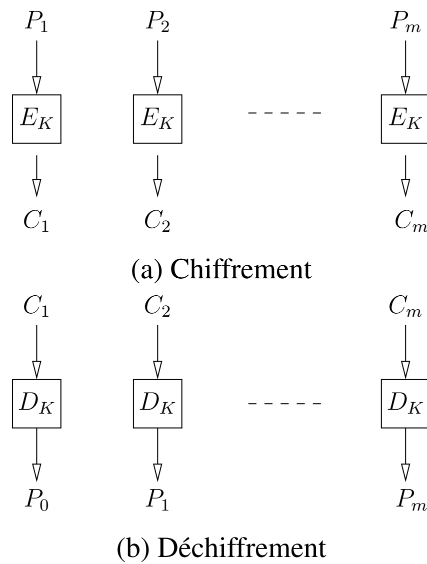


FIGURE 1.6 – Chiffrement par bloc en mode ECB

— E_K est l'opération de chiffrement d'un bloc avec la clé K
— D_K est l'opération de déchiffrement d'un bloc avec la clé K

On suppose également que la longueur des messages à chiffrer est multiple de n. Si ce n'est pas le cas, un algorithme de remplissage (*padding*) est utilisé pour étendre le message.

Electronic CodeBook (ECB) Le fonctionnement du mode ECB est simple. Le message en clair P est découpé en blocs $P_1, P_2, ..., P_m$ de taille n (celle de l'algorithme de chiffrement). Chacun de ces blocs est chiffré indépendamment des autres. Le résultat du chiffrement donne les blocs chiffrés $C_1, C_2, ..., C_m$ qui constituent le message chiffré C :

$$P = P_1 \| P_2 \| ... \| P_m \tag{1.11}$$

$$C = C_1 \| C_2 \| ... \| C_m \tag{1.12}$$

pour l'opération de chiffrement :

$$C_i = E_K(P_i) \quad \forall i \in [1, m] \tag{1.13}$$

et pour l'opération de déchiffrement :

$$P_i = D_K(C_i) \quad \forall i \in [1, m] \tag{1.14}$$

La figure 1.6 présente les deux opérations de chiffrement et de déchiffrement du mode ECB.

Le principal inconvénient du mode ECB, d'un point de vue sécurité, est que le chiffrement de deux blocs identiques du message en clair, avec la même clé, produit des blocs chiffrés identiques. Un message contenant des motifs répétitifs ne doit pas être chiffré avec un tel mode

parce que l'adversaire peut en déduire lorsque la même information se produit deux fois. Une solution pour améliorer la sécurité du mode ECB est d'inclure des bits aléatoires dans chaque bloc du texte clair avant le chiffrement [4].

Dans ce mode, les opérations de chiffrement et de déchiffrement sont entièrement parallélisables, ce qui permet de rajouter autant de fonctions de chiffrement/déchiffrement que nécessaire, et donc d'augmenter les performances. Un autre avantage du mode ECB est que la propagation des erreurs est limitée. Si un bit du message chiffré est erroné, seul le déchiffrement du bloc chiffré correspondant est impacté. Par contre l'insertion ou la suppression d'un bit dans le message chiffré affectera toute la suite du message en clair.

Cipher-Block Chaining (CBC) Dans le mode CBC, chaque bloc P_i du message en clair P est combiné avec le bloc chiffré précédent C_{i-1} (ou avec un Vecteur d'Initialisation (noté IV) choisi aléatoirement dans le cas du premier bloc (P_1)) en utilisant l'opérateur xor, afin d'obtenir le bloc chiffré C_i, comme l'illustre l'équation suivante :

$$C_0 = IV \quad (1.15)$$

$$C_i = E_K(C_{i-1} \oplus P_i) \quad \forall i \in [1, m] \quad (1.16)$$

et pour le déchiffrement :

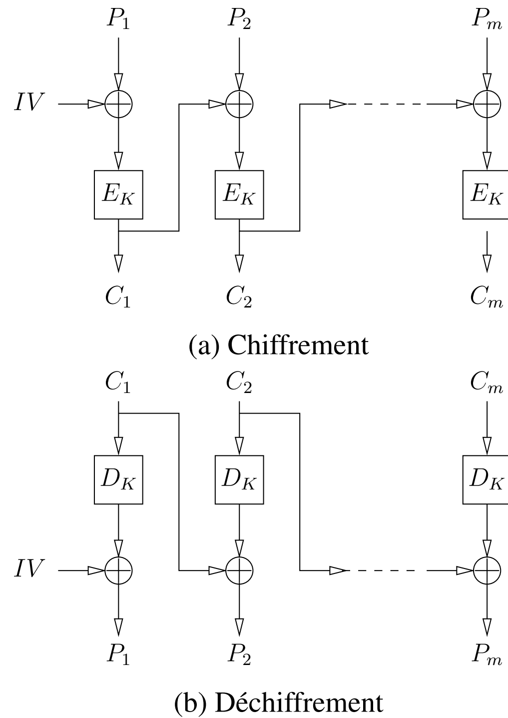
$$C_0 = IV \quad (1.17)$$

$$P_i = C_{i-1} \oplus D_K(C_i) \quad \forall i \in [1, m] \quad (1.18)$$

La figure 1.7 présente les deux schémas de chiffrement et de déchiffrement du mode CBC.

L'avantage principal du mode CBC par rapport au mode ECB est de rendre le chiffrement d'un bloc dépendant du chiffrement précédent. Cela permet de masquer les motifs présents dans le message en clair. Cependant, un message complet chiffré deux fois en mode CBC (en utilisant la même clé) produira le même message chiffré si le même IV est utilisé. Une façon de surmonter un tel inconvénient est de changer l'IV pour chaque message chiffré. Le vecteur d'initialisation doit donc être généré de manière aléatoire pour chaque message à chiffrer.

Deux blocs en clair identiques donneront un chiffré identique que si les chiffrés précédents sont identiques. Cette situation (collision, c'est-à-dire deux blocs chiffrés identiques permettant de savoir si les deux blocs en clair suivant immédiatement sont identiques ou non) est facilement détectable par l'adversaire. Les blocs chiffrés étant normalement indistinguables de blocs aléatoires, la probabilité de survenue d'une collision est donnée par le paradoxe des anniversaires et dépend directement de la taille d'un bloc (n). Cette probabilité est de $0,5$ au bout de $2^{n/2}$ blocs chiffrés. Il faut donc, lors de l'utilisation du mode CBC, changer la clé lorsque l'on s'approche de cette limite afin d'éviter la survenue d'une collision qui pourrait laisser fuir de l'information sur les messages en clair.



(a) Chiffrement

(b) Déchiffrement

FIGURE 1.7 – Chiffrement par bloc en mode CBC

L'opération de chiffrement du mode CBC ne peut pas être parallélisé entre les différents blocs constituant un même message car le chiffrement d'un bloc dépend du résultat du chiffrement du bloc précédent. Par contre, l'opération de déchiffrement peut être parallélisée car toutes les informations nécessaires sont disponibles : les deux blocs chiffrés (courant et précédent) et la clé.

Le mode CBC a un effet limité dans la propagation des erreurs. Si un bit du message chiffré est erroné, il affecte le déchiffrement du bloc en question ainsi que du bloc suivant.

Cipher FeedBack (CFB) Le mode CFB permet d'utiliser un algorithme de chiffrement par bloc comme un algorithme de chiffrement de flux auto-synchrone. La sortie de l'algorithme est le bloc chiffré C_i et l'état interne (σ_i) est défini par le bloc chiffré précédent C_{i-1}. Un tel mode nécessite un IV (la graine jetable σ_0) pour initialiser σ_i. Les équations suivantes définissent les opérations de chiffrement du mode CFB :

$$C_0 = IV \tag{1.19}$$

$$\sigma_i = E_K(C_{i-1}) \quad \forall i \in [1, m] \tag{1.20}$$

$$C_i = \sigma_i \oplus P_i \quad \forall i \in [1, m] \tag{1.21}$$

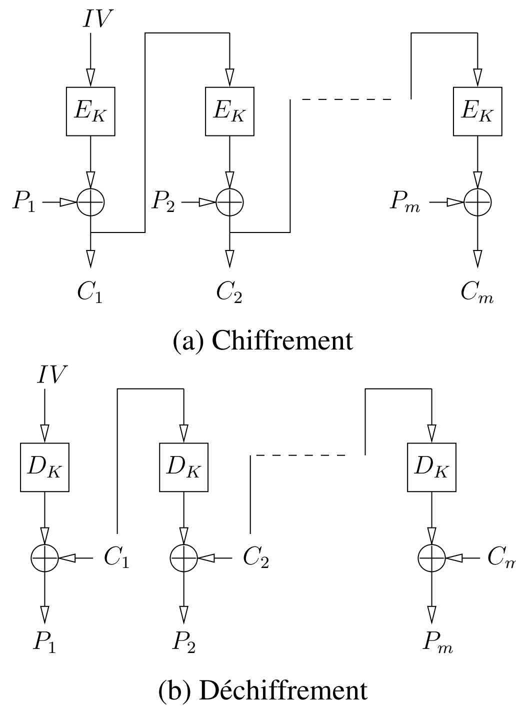


FIGURE 1.8 – Chiffrement par bloc en mode CFB

et pour le déchiffrement :

$$C_0 = IV \tag{1.22}$$

$$\sigma_i = D_K(C_{i-1}) \quad \forall i \in [1, m] \tag{1.23}$$

$$P_i = \sigma_i \oplus C_i \quad \forall i \in [1, m] \tag{1.24}$$

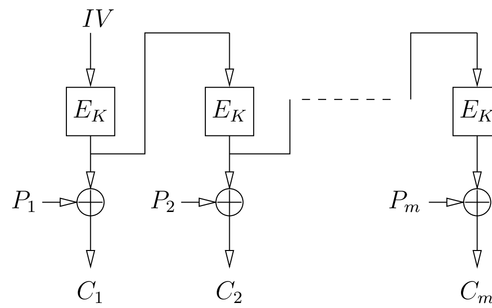
La figure 1.8 présente les deux schémas de chiffrement et de déchiffrement du mode CFB.

Le mode CFB bénéficie de la synchronisation automatique comme dans le chiffrement de flux auto-synchrone. Cette propriété permet de limiter la propagation des erreurs lors de la suppression ou de l'insertion d'un bit dans le message chiffré. De plus, la taille du message à chiffrer n'a pas besoin d'être un multiple de n contrairement aux modes ECB ou CBC.

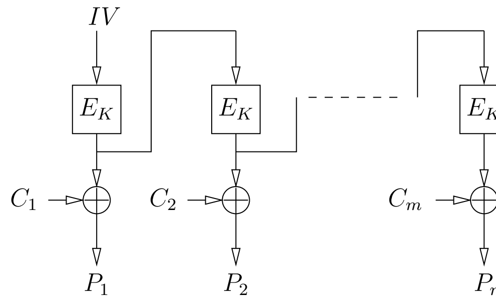
Output FeedBack (OFB) Ce mode est similaire au chiffrement de flux synchrone et donc bénéficie des mêmes propriétés comme la non propagation des erreurs. Aussi, comme dans le mode CFB, la taille du message à chiffrer n'a pas besoin d'être un multiple de n.

Les équations suivantes définissent les opérations de chiffrement du mode OFB :

$$\sigma_0 = IV \tag{1.25}$$



(a) Chiffrement



(b) Déchiffrement

FIGURE 1.9 – Chiffrement par bloc en mode OFB

$$\sigma_i = E_K(\sigma_{i-1}) \quad \forall i \in [1, m] \tag{1.26}$$

$$C_i = \sigma_i \oplus P_i \quad \forall i \in [1, m] \tag{1.27}$$

et pour le déchiffrement :

$$\sigma_0 = IV \tag{1.28}$$

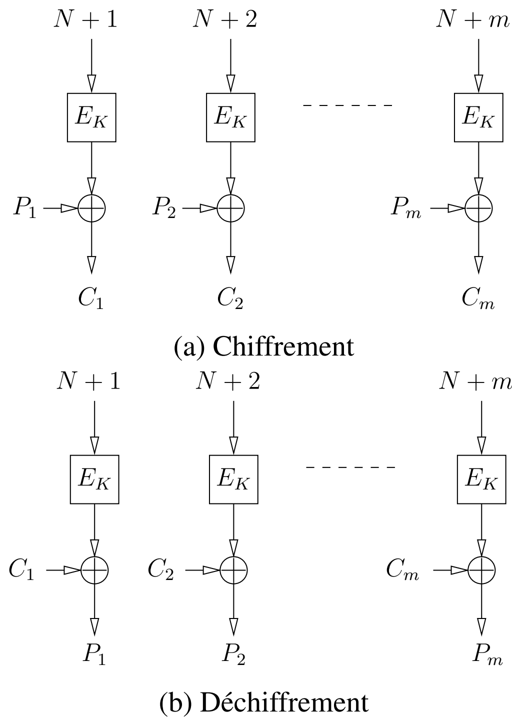
$$\sigma_i = D_K(\sigma_{i-1}) \quad \forall i \in [1, m] \tag{1.29}$$

$$P_i = \sigma_i \oplus C_i \quad \forall i \in [1, m] \tag{1.30}$$

La figure 1.9 présente les deux schémas de chiffrement et de déchiffrement du mode OFB.

Dans le mode OFB, le calcul des masques peut être effectué en avance parce qu'il dépend uniquement du vecteur d'initialisation, et donc, une fois les masques calculés, le chiffrement ou le déchiffrement peuvent effectués très rapidement (l'opérateur xor).

Mode compteur Le mode compteur permet aussi, comme les modes CFB et OFB, d'utiliser un algorithme de chiffrement par bloc comme un algorithme de chiffrement de flux. Aussi, la taille du message à chiffrer n'a pas besoin d'être un multiple de n.



(a) Chiffrement

(b) Déchiffrement

FIGURE 1.10 – Chiffrement par bloc en mode compteur

Le mode compteur est initialisé par un *Nounce* (nombre aléatoire, noté N) et incrémenté pour chaque bloc en clair. Les équations suivantes définissent les opérations de chiffrement du mode compteur :

$$\sigma_i = E_K(N + i) \quad \forall i \in [1, m] \tag{1.31}$$

$$C_i = \sigma_i \oplus P_i \quad \forall i \in [1, m] \tag{1.32}$$

et pour le déchiffrement :

$$\sigma_i = E_K(N + i) \quad \forall i \in [1, m] \tag{1.33}$$

$$P_i = \sigma_i \oplus C_i \quad \forall i \in [1, m] \tag{1.34}$$

La figure 1.10 présente les deux schémas de chiffrement et de déchiffrement du mode compteur.

Le mode compteur possède deux atouts principaux d'un point de vue des performances :
- le chiffrement ou le déchiffrement de deux blocs peuvent être parallélisés,
- la partie la plus longue de l'opération de chiffrement ou de déchiffrement (calcul de $E_k(N = i)$) ne dépend pas du bloc en clair ou du bloc chiffré et peut donc être réalisé en amont.

1.2.1.3 Conclusion

Comme il sera présenté dans la section 1.3, les algorithmes de chiffrement par bloc peuvent être utilisés comme bases pour construire d'autres algorithmes afin remplir d'autres objectifs de sécurité que la confidentialité, comme par exemple assurer l'intégrité et d'authentification de données.

Les algorithmes de chiffrement symétriques posent le problème du partage de secret (ici la clé). En effet, si deux entités veulent s'échanger des messages chiffrés, ces deux entités doivent disposer du même secret. Elles doivent donc s'être mises d'accord sur ce secret et se l'être échangé sur un canal sécurisé en amont de la communication. De plus le nombre de clés à gérer augmente de façon quadratique avec le nombre d'entités avec lesquelles on souhaite communiquer. Les algorithmes de chiffrement asymétriques permettent de palier ces deux problèmes.

1.2.2 Chiffrement asymétrique

Contrairement aux algorithmes de chiffrement symétriques, les algorithmes de chiffrement asymétrique utilisent deux clés différentes : la clé de chiffrement K_e, appelée clé publique car elle peut être transmise sur un canal non sécurisé et ne nécessite pas d'être gardée secrète, et la clé de déchiffrement K_d, appelée clé privée puisque qu'elle permet le déchiffrement et donc ne doit être connue que du destinataire du message chiffré (figure 1.2).

La notion du chiffrement asymétrique (ou cryptosystème à clé publique) a été introduite par Wilfred Diffie et Martin Hellman en 1976 [14]. Ces algorithmes sont basés sur la difficulté à résoudre certains problèmes mathématiques complexes et en particulier sur des fonctions à sens unique à trappe. Une fonction $f(x) = y$ est considérée comme une fonction à sens unique si pour tout x appartenant à l'ensemble de définition de la fonction, il est simple de calculer $f(x)$, alors qu'en pratique, connaissant y, il est difficile de trouver x tel que $y = f(x)$ sans connaître une information supplémentaire : la trappe (secret).

Un exemple de problème mathématique complexe utilisé est la factorisation de grands nombres : il est facile de calculer la multiplication de deux grands nombres, mais la fonction inverse, la factorisation est très difficile. L'algorithme à clé publique le plus célèbre est RSA [15] (pour Rivest Shamir Adleman, les noms de ses inventeurs). Il est basée sur ce problème de factorisation.

Le principal avantage du chiffrement asymétrique par rapport au chiffrement symétrique est le fait qu'il n'a pas besoin de partager un secret avant d'établir la communication, la clé de chiffrement pouvant être publique. Néanmoins, il est nécessaire d'assurer l'authenticité de cette clé de chiffrement sinon une attaque de l'homme du milieu (*man-in-the-middle*) est réalisable. Dans une telle attaque, un adversaire qui surveille un canal non sécurisé, intercepte un message (M) et éventuellement certaines données (X) utiles à des fins de sécurité, les analyse et les échange par des données choisies (M', X').

Par exemple, Alice et Bob veulent échanger des données chiffrées sur un canal non sécurisé qui est écouté par Ève, l'adversaire. Bob envoie tout d'abord sa clé publique à Alice via ce

canal. Ève peut alors intercepter cette clé et la remplacer à la volée par sa propre clé publique. Alice chiffre alors ses messages avec ce qu'elle pense être la clé publique de Bob mais qui est en réalité la clé publique d'Ève. Cette dernière peut donc déchiffrer tous les messages (à l'aide de sa clé privée correspondante) et éventuellement les rechiffrer avec la vraie clé publique de Bob pour que ce dernier ne s'aperçoive de rien.

Ce problème peut être atténué à l'aide d'une infrastructure de gestion de clés (*Public Key Infrastructure*, PKI) qui émet des certificats pour valider l'authenticité de clés. Néanmoins, elle ne fait que déplacer le problème au niveau de l'authenticité de la clé ayant signé les certificats.

En outre, les algorithmes de chiffrement asymétrique sont très lents, car ils sont basés sur des problèmes mathématiques complexes. Par exemple, dans le déchiffrement matériel de RSA prend jusqu'à 260.000 cycles avec une fréquence de 100 MHz et une taille de clé de 1024 bits [16].

1.2.3 Sécurité des techniques de chiffrement

Les attaques menées contre un système de chiffrement peuvent avoir deux objectifs : la récupération de clé ou du message en clair.

L'objectif de la première attaque est de récupérer la clé secrète ou la clé privée utilisée respectivement dans un algorithme de chiffrement symétrique ou dans un chiffrement asymétrique. L'attaque la plus pratique est l'attaque par force brute. Elle consiste à essayer toutes les clés possibles jusqu'à trouver la bonne clé permettant de récupérer le texte en clair à partir de n'importe quel texte chiffré. Par conséquent, plus la taille de la clé est longue, plus la complexité de cette attaque est grande.

Le but des attaques de récupération de message est d'obtenir des informations sur les textes clairs en observant les textes chiffrés. Un cryptosystème est à l'abri contre la récupération du message s'il est sémantiquement sécurisé, ce qui signifie qu'un adversaire n'est pas en mesure de savoir si un texte chiffré donné est le chiffrement d'un texte clair ou d'un autre.

Les objectifs de l'adversaire, définis plus haut, sont pratiquement basés sur les attaques passives qui pourraient être classées en considérant les informations dont l'adversaire dispose. Les définitions suivantes sont issues de [4] :

1. Attaque par texte chiffré seul (*ciphertext-only attack*) : l'attaquant ne dispose que de messages chiffrés.
2. Attaque par texte clair connu (*known-plaintext attack*) : l'attaquant dispose d'une série de messages clairs et des chiffrés correspondants.
3. Attaque par texte clair choisi (*chosen-plaintext attack*) : l'attaquant peut obtenir les chiffrés des messages en clair qu'il veut.
4. Attaque par texte clair choisi adaptative (*adaptative chosen-plaintext attack*) : identique à une attaque par texte clair choisi sauf que le choix des textes clairs par l'attaquant peut dépendre des textes chiffrés reçus précédemment.

5. Attaque par texte chiffré choisi (*chosen-ciphertext attack*) : l'attaquant a accès à une machine de déchiffrement. Il peut obtenir les messages en clair correspondants aux messages chiffrés qu'il veut (excepté le message qu'il essaie de déchiffrer).
6. Attaque par texte chiffré choisi adaptative (*adaptative chosen-ciphertext attack*) : identique à une attaque par texte chiffré choisi sauf que le choix des messages chiffrés soumis à l'oracle de déchiffrement par l'attaquant peut dépendre des messages en clair qu'il a reçus précédemment.

La robustesse d'un système de chiffrement est évaluée en respectant le premier et le deuxième principe de Kerckhoffs qui indiquent qu'un algorithme doit être incassable en pratique (sinon en théorie est incassable) et qu'il doit être connu du public.

Bien utilisé, le chiffrement permet de garantir la confidentialité d'informations contre des attaques passives. Néanmoins, le chiffrement seul ne peut pas garantir l'intégrité et l'authentification des données. Il est donc insuffisant dès que l'adversaire est en mesure d'altérer les messages, notamment via des attaques actives.

1.3 Techniques de vérification d'intégrité

Les techniques de vérification d'intégrité permettent de détecter des altérations, volontaires ou non, sur des données. Pour assurer l'intégrité des données, il faut avoir la capacité de détecter toute manipulation (insertion, suppression ou modification) des données par des tiers non autorisés.

Les mécanismes de vérification d'intégrité utilisent des algorithmes principalement basés sur des fonctions de hachage. Par la suite nous présenterons ces fonctions et leurs caractéristiques.

1.3.1 Fonctions de hachage

Une fonction de hachage h, également appelée une fonction de hachage à sens unique (*one-way hash function*), est une fonction qui possède au moins de deux propriétés :

— **Compression** (*compression*) : h calcule un condensé cryptographique (appelé *hash*) de longueur fixe à partir d'un message d'entrée de longueur arbitraire x.
— **Facilité de calcul** (*ease of computation*) : étant données h et une entrée x, il est facile de calculer $h(x)$.

Les fonctions de hachage sont caractérisées par leur résistance à trois propriétés de sécurité [4] :

— **Résistance à la préimage** : étant donné un condensé cryptographique y, il est difficile de forger un message x tel que $y = h(x)$.
— **Résistance à la seconde préimage** : étant donné un message x et son condensé cryptographique $y = h(x)$, il est difficile de trouver un autre message $x' \neq x$ tel que

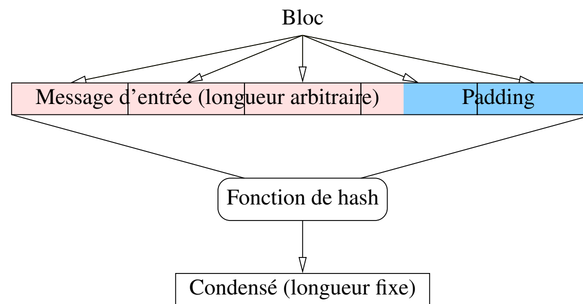


FIGURE 1.11 – Fonction de hachage

$h(x) = h(x') = y$.

— **Résistance aux collisions** : il est difficile de trouver deux messages différents x et x' ayant le même condensé cryptographique $h(x) = h(x')$.

Les fonctions de hachage sont divisées en deux catégories :

— **Codes de détection de modification** (*modification detection codes, MDC*) basés sur des fonctions de hachage sans clé et dont l'objectif est de garantir l'intégrité d'un message.
— **Codes d'authentification de message** (*message authentication codes, MAC*) basés sur des fonctions de hachage avec clé et dont l'objectif est de garantir l'intégrité et l'authentification de la source d'un message.

1.3.1.1 Code de détection de modification

Les fonctions de hachage sans clé possèdent une seule entrée de longueur quelconque, le message à hacher, et produisent un condensé de longueur fixe 1.11.

Ce type de fonctions est divisé en deux catégories :

— Fonctions de hachage à sens unique (*one-way hash function, OWHF*) qui sont caractérisées par les deux propriétés de compression et facilité de calcul, et deux propriétés de sécurité : résistance à la préimage et résistance à la seconde préimage.
— Fonctions de hachage résistant aux collisions (*collision resistant hash function, CRHF*) qui sont caractérisées par les deux propriétés de compression et facilité de calcul, et deux propriétés de sécurité : résistance à la seconde préimage et résistance aux collisions.

1.3.1.1.1 Exemple : SHA L'Agence de Sécurité Nationale américaine (NSA) a créé, en 2002, les fonctions SHA (*Secure Hash Algorithm*) [17] et qui ont été publiées par le NIST. Ces fonctions sont regroupées en une série de cinq fonctions de hachage (SHA-1, SHA-224, SHA-256, SHA-384 et SHA-512) résistantes aux collisions.

La fonction de hachage SHA-1 est l'une des fonctions de hachage les plus couramment utilisées. Par exemple, elle est utilisée dans des protocoles tels que SSL (*Secure Sockets Layer*), TLS (*Transport Layer Security*) et IPSec (*Internet Protocol Security*). SHA-1 prend en entrée un message M inférieur à 2^{64} bits et traite des blocs de 512 bits et, produit en sortie un condensé de 160 bits.

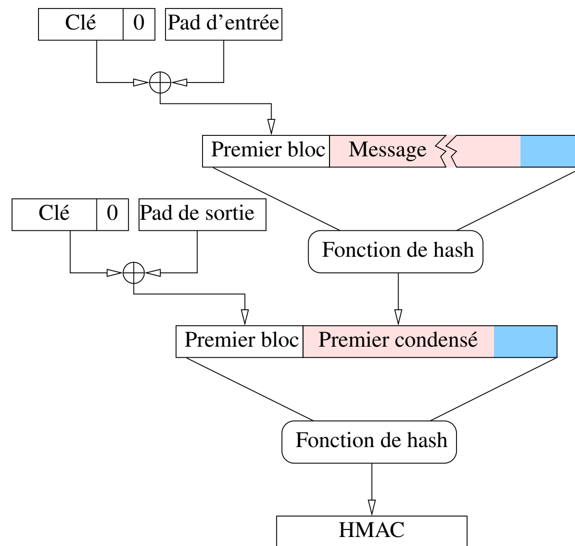


FIGURE 1.12 – Fonction de HMAC

1.3.1.2 Code d'authentification de message

Un algorithme de MAC (code d'authentification de message), ou fonction de hachage à clé, est une famille de fonctions h_k paramétrés par une clé secrète k avec les propriétés suivantes :

— h_k est une fonction de hachage (propriétés de *compression* et de *facilité de calcul*)

— étant donnés zéro ou plus couples messages-MAC $(x_i, h_k(x_i))$, il doit être impossible de calculer le MAC d'un message $x \neq x_i$ sans connaître la clé k (propriété de *résistance à la falsification*)

Une fonction de hachage peut être transformée en MAC en combinant les messages d'entrée avec une clé secrète (comme par exemple avec la construction HMAC, figure 1.12).

1.3.1.2.1 Exemple : CBC-MAC Les fonctions de MAC peuvent aussi être basées sur un algorithme de chiffrement par bloc utilisé en mode CBC : CBC-MAC (Cipher Block Chaining Message Authentication Code). Pour obtenir le MAC d'un message, on chiffre ce message avec un algorithme de chiffrement par bloc dans le mode CBC (E) avec la clé de chiffrement k et un vecteur d'initialisation ($IV = 0$). Le dernier bloc chiffré du groupe représente le MAC du message (figure 1.13).

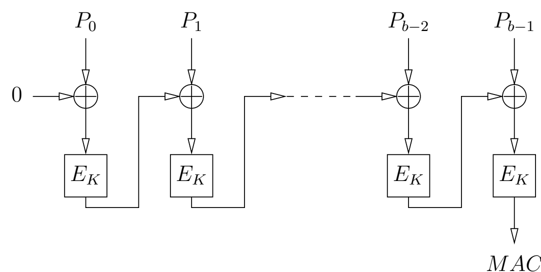


FIGURE 1.13 – Calcul de CBC-MAC

Cette construction impose néanmoins de connaître la taille du message. En effet, si un adversaire connaît deux messages (P et P') et leurs MAC associés (M et M'), il peut créer un message dont le MAC sera M' en effectuant une opération de xor entre M et le premier bloc du message M' et en concaténant M et M' ainsi modifié. Par conséquent, l'algorithme CBC-MAC ne doit pas être utilisé dans le cas où l'on ne connaît pas la taille du message.

1.3.2 Paradoxe des anniversaires

La taille du condensé est importante du point de vue de la sécurité à cause du problème de la résistance aux collisions, menacée par le paradoxe des anniversaires. En pratique, supposons une fonction h qui produit des condensés dans un ensemble de N éléments. Au bout de $\sqrt{N}$ appels à la fonction h, la probabilité que deux condensés identiques aient été générés est de $1/2$. Cela signifie que pour une fonction de hachage produisant un condensé de n bits, cette probabilité est obtenue après le calcul de $2^{n/2}$ condensés. Par exemple, une attaque basée sur le paradoxe des anniversaires permettrait de trouver une collision sur le SHA-1 avec un nombre d'opérations de l'ordre de 2^{80}.

1.3.3 Conclusion

Dans ce chapitre nous avons présenté les différents algorithmes de base de la cryptographie dont nous aurons besoin par la suite tout au long de ce manuscrit, et particulièrement les techniques de chiffrement et de vérification d'intégrité.

Chapitre 2

Panorama des techniques de protection

Sommaire

2.1 Modèle de menace **38**
2.2 Méthodes de protection **40**
2.2.1 Confidentialité 40
2.2.2 Intégrité 42
2.3 Les arbres de Merkle réguliers **46**
2.3.1 Opérations 47
2.3.2 Évaluation des performances 49
2.3.3 Variantes des arbres de Merkle 57
2.3.4 Comparaison des différents arbres 60
2.4 Plates-formes de calcul sécurisées **61**
2.4.1 Best 61
2.4.2 Dallas DS5002FP 62
2.4.3 XOM 63
2.4.4 AEGIS 65
2.4.5 CryptoPage 66
2.4.6 MESA 67
2.4.7 PE-ICE 67
2.5 Conclusion **68**

La protection de données résidant dans un espace de stockage non fiable ou non digne de confiance est un problème critique du point de vue de la sécurité. Deux propriétés non fonctionnelles sont à assurer : la confidentialité et l'intégrité.

De nombreux systèmes (par exemple dans le domaine du *cloud computing*, les systèmes embarqués ou les bases de données) manipulent des informations sensibles qui sont stockées dans des espaces de stockage non nécessairement fiables, dignes de confiance ou inaltérables. Ces données peuvent alors y être altérées par un adversaire.

Par exemple, si un attaquant ayant physiquement accès à un système embarqué parvient à espionner le bus de communication entre le ou les composants de calcul (processeurs, SoC)

et les composants de stockage (la mémoire vive, le disque dur, etc.), il peut alors récupérer le code exécuté et les données manipulées, et obtenir ainsi de l'information potentiellement sensible (attaques passives). Il peut aussi modifier les informations qui transitent sur le bus ou le contenu des mémoires (attaques actives), ce qui peut perturber le bon fonctionnement du système.

Dans la section 2.1, nous introduirons les différentes attaques possibles sur ces types de systèmes, puis nous présenterons dans la section 2.2 les mécanismes de protection proposés dans la littérature. Ensuite, dans la section 2.3, nous étudierons en profondeur une méthode particulière de protection de l'intégrité : les arbres de Merkle. Enfin, un panorama des plates-formes de calcul sécurisées intégrant ces techniques de protection est présenté dans la section 2.4.

2.1 Modèle de menace

La conception des solutions de sécurité repose principalement sur la définition des types d'attaques contre lesquelles le système doit être protégé. En effet, les mécanismes implémentés pour garantir les objectifs de sécurité peuvent être différents suivant les ressources (équipement, moyens financiers) et les compétences utilisables par un attaquant.

De nombreuses applications informatiques nécessitent un certain niveau de confidentialité (si les données manipulées ou les algorithmes utilisés sont sensibles) et d'intégrité (si le code exécuté, les données d'entrée et les résultats produits doivent être conformes à l'intention des concepteurs). Ces propriétés sont plus facilement obtenues lorsque les calculs sont effectués localement, mais sont plus difficiles à atteindre lorsque les applications sont exécutées à distance, sur des machines appartenant à des tiers. Malheureusement, ce dernier cas est de plus en plus fréquent avec le développement du *cloud computing*.

Par exemple, une entreprise ayant besoin d'effectuer des calculs intensifs, peut utiliser la puissance de calcul d'un fournisseur de *cloud computing*. Cependant, comme ces calculs ne sont plus effectués localement, l'entreprise n'a plus le contrôle sur la confidentialité et l'intégrité de ces calculs et doit faire confiance au fournisseur de *cloud computing*.

Différentes attaques peuvent compromettre ces propriétés, que se soit des attaques logicielles ou matérielles. Les attaques logicielles peuvent être le fait d'une application malveillante qui s'exécute en même temps que l'application victime dans le serveur de calcul, et à laquelle une faille dans le système d'exploitation ou l'environnement de virtualisation, permet d'espionner ou même de perturber l'application victime. Elles peuvent également être le fait du système d'exploitation ou de l'environnement de virtualisation eux-mêmes, s'ils sont aux mains d'administrateurs malveillants. Les attaques matérielles peuvent être menées par une personne physique (administrateur, technicien, intrus) disposant d'un accès physique aux serveurs sur lesquels l'application est exécutée, et qui peut récupérer physiquement le code et les données manipulées par l'application.

Cette situation ne concerne pas seulement les plates-formes de *cloud computing*, mais aussi les systèmes embarqués qui manipulent des données sensibles qui n'appartiennent pas à l'utilisateur final. Ceci est le cas, par exemple, des consoles de jeux vidéos manipulant des flux

vidéos et des informations concernant la gestion des droits numériques qui ne devraient pas être accessibles par l'utilisateur.

De nombreuses menaces visent les systèmes embarqués à plusieurs niveaux. Premièrement, les composants de calcul eux-mêmes peuvent être visés. On peut citer, par exemple, les attaques par canaux auxiliaires (SCA [18, 19]), les injections de faute [20], la rétro-conception, etc. Les attaques peuvent aussi concerner d'autres composants matériels du système complet : espionnage ou injections sur les bus de communication ou les contenus de mémoires [2], etc. Enfin, les composants logiciels peuvent également être la cible des attaques : exploitation de failles dans le système d'exploitation ou les applications [21], etc. L'objectif commun de ces attaques est généralement d'extraire des informations sensibles ou de perturber le bon fonctionnement du système embarqué.

Nous considérerons par la suite le modèle de menace suivant. Le système considéré est schématiquement composé d'un composant de calcul principal (*System-on-Chip*, SoC) et d'une mémoire externe, dans laquelle sont stockés le code et les données des applications. Nous supposons que le composant de calcul est inattaquable (ce qui exclut donc les attaques par canaux auxiliaires, la rétro-conception, etc.). Par contre, nous supposons que la mémoire et le bus mémoire sont sous le contrôle de l'attaquant (figure 2.1).

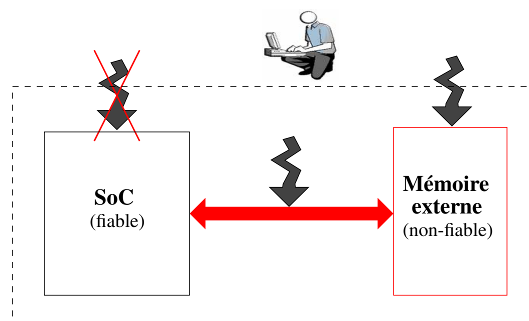


FIGURE 2.1 – Zones d'attaques considérées.

L'attaquant peut donc avoir accès aux données contenues dans la mémoire externe (problème de confidentialité) et peut également les modifier (problème d'intégrité).

L'intégrité consiste à s'assurer de la propriété suivante : *lors d'une lecture à une adresse donnée, la donnée reçue par le processeur depuis la mémoire doit être la dernière donnée stockée par le processeur à cette même adresse*. De manière générale, les attaques actives qui menacent l'intégrité de la mémoire externe non fiable peuvent être réparties en trois catégories :

— *Injection* : l'attaquant remplace une donnée par une autre donnée de son choix. Dans la figure 2.2, l'attaquant remplace la donnée `0x152AAB32` de l'adresse 3 par la donnée *forgée* `0xFFFFFFFF`.
— *Permutation spatiale* : l'attaquant remplace la donnée d'une adresse par la donnée d'une adresse différente. Dans la figure 2.3, l'attaquant remplace la donnée `0x152AAB32` de l'adresse 3 par la donnée `0x0001AE22` de l'adresse 0.
— *Permutation temporelle* ou *Rejeu* : l'attaquant remplace la donnée la plus à jour d'une adresse par une ancienne donnée de la même adresse. Dans la figure 2.4, l'attaquant remplace la donnée `0x152AAB32` de l'adresse 3, écrite au temps T_2 par l'ancienne

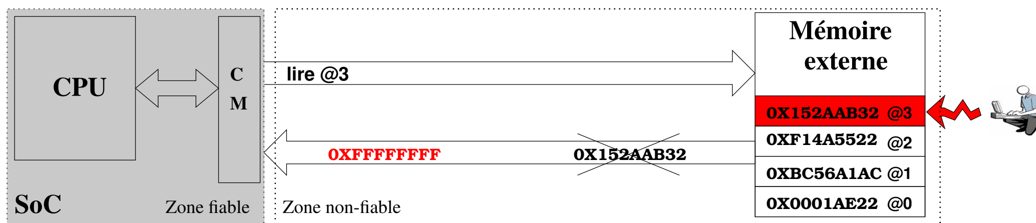


FIGURE 2.2 – Attaque par *injection*

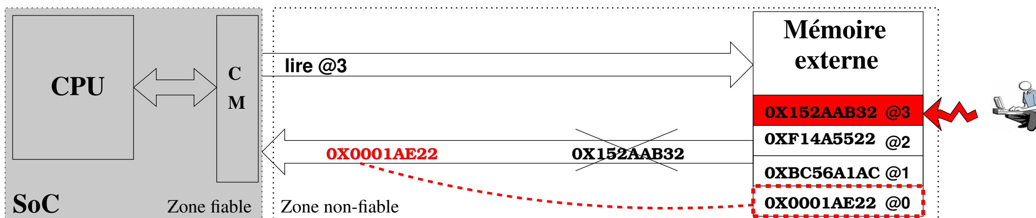


FIGURE 2.3 – Attaque par *permutation spatiale*

donnée `0xA67A11FF` de la même adresse mais écrite au temps T_1, où $T_1 < T_2$. Au moment de la réponse, la donnée retournée n'est donc pas la dernière donnée stockée à l'adresse demandée.

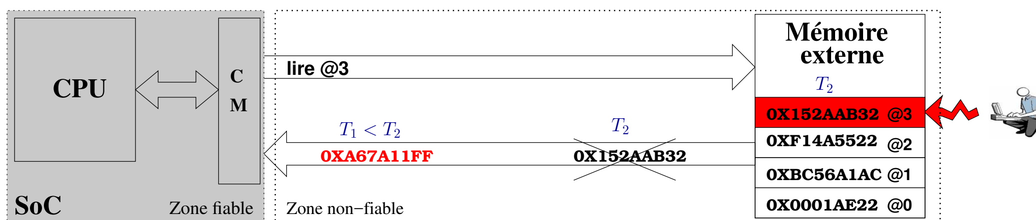


FIGURE 2.4 – Attaque par *rejeu*

2.2 Méthodes de protection

Nous présenterons dans cette section les différentes solutions qui ont été développées pour protéger la confidentialité et l'intégrité de la zone mémoire non fiable.

Dans le cas que nous étudions (système embarqué), les mécanismes de protection du contenu de la mémoire externe doivent être réalisés de l'intérieur du SoC que nous avons considéré comme zone fiable. Nous considérons donc qu'un module cryptographique est responsable de cette protection, et est localisé à l'intérieur du SoC, comme le montre la figure 2.5.

2.2.1 Confidentialité

La protection de la confidentialité est basée principalement sur des techniques de chiffrement. Parmi ces techniques, les algorithmes de chiffrement symétrique ont un avantage par

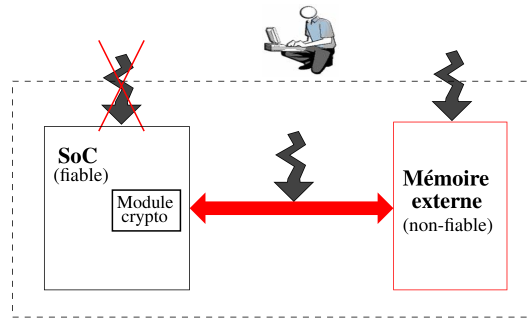


FIGURE 2.5 – Ajout du module cryptographique à l'intérieur du SoC.

rapport aux algorithmes de chiffrement asymétrique. En terme de puissance de calcul, le chiffrement asymétrique est plus coûteux que le chiffrement symétrique car il utilise des primitives mathématiques qui manipulent des grands nombres et/ou enchaînent un grand nombre d'itérations. De plus, dans notre cas, le module cryptographique joue le rôle des deux intervenants du modèle classique de communication sécurisée. Il est à la fois Alice et Bob, il chiffre et déchiffre. C'est là une différence importante, du point de vue de la sécurité, entre le stockage et le transport d'information. Si les clés cryptographiques sont produites par le SoC lui-même et ne le quittent jamais, il n'est donc pas nécessaire d'avoir recours à la cryptographie asymétrique, pas même pour établir un secret commun par le biais d'un canal non protégé. Pour ces deux raisons c'est la cryptographie symétrique qui nous fournira les primitives de chiffrement dont nous avons besoin.

Le choix entre les types de chiffrement, c'est-à-dire entre le chiffrement par bloc et le chiffrement de flux ou encore entre les différents modes d'opération (ECB, CBC, CFB, OFB et le mode compteur), dépend du type des données à protéger (en lecture seule ou en lecture et écriture), le niveau de sécurité, l'impact sur les performances du système, etc. Par exemple le mode ECB (*Electronic CodeBook*) est caractérisé par son parallélisme (le chiffrement et le déchiffrement d'un bloc est indépendant de celui d'un autre bloc) et simplicité d'implémentation. Mais du point de vue de la sécurité, un des inconvénients majeurs de ce mode d'opération est que le chiffrement de deux blocs identiques donne deux blocs chiffrés identiques. Un autre exemple, le mode compteur possède des propriétés intéressantes, en plus du parallélisme et de la simplicité d'implémentation, du point de vue des performances : une partie des opérations de chiffrement et de déchiffrement peut être réalisée en avance, sans avoir besoin des données. La latence d'accès à une mémoire distante peut donc être mise à profit pour anticiper une partie des calculs cryptographiques.

Ces modes peuvent être utilisés pour chiffrer des données non modifiables (c'est-à-dire des données en lecture seule, écrites une seule fois au début du programme, par exemple le code d'une application). Par contre, ils ne peuvent pas être utilisés pour les données modifiables (par exemple les données de pile), parce que ce sont des approximations du masque jetable (*One-Time Pad*, OTP) qui, comme son nom l'indique, exige une utilisation unique du masque. Chaque modification des données protégées doit donc être accompagnée de la génération d'un nouveau masque. Dans le cas de données modifiables nous pouvons utiliser, par exemple, le chiffrement de flux auto-synchrone ou le mode CBC du chiffrement par bloc. Le mode CBC, s'il est utilisé avec des vecteurs d'initialisation aléatoires re-générés à chaque chiffrement, pos-

sède une propriété intéressante du point de vue de la sécurité : si un même bloc de message en clair est chiffré deux fois, il donne deux blocs chiffrés différents, puisque le chiffrement dépend du bloc chiffré précédent ou, pour le premier bloc du message, du vecteur d'initialisation.

2.2.2 Intégrité

L'objectif de la vérification d'intégrité est de détecter toute injection ou corruption du code ou des données, et d'interdire l'exécution du contenu de la mémoire intentionnellement modifié. La propriété d'intégrité est plus difficile à assurer que la propriété de confidentialité car la vérification de l'intégrité d'une donnée nécessite toujours une donnée supplémentaire de référence qu'il faut produire et stocker. Dans cette sous-section, nous présenterons les différents mécanismes de protection [1] d'intégrité contre les trois attaques actives que nous avons présenté dans le modèle de menace.

2.2.2.1 Méthodes de protection contre l'attaque par injection

Pour protéger l'intégrité d'une donnée, il est nécessaire de stocker une information supplémentaire permettant de s'assurer que la donnée n'a pas été modifiée. Le condensé cryptographique est un choix naturel : il est facile à calculer et il prend moins de place que la donnée initiale.

Néanmoins, les fonctions de hachage sont publiques. Il ne servirait donc à rien de stocker les condensés dans la mémoire non fiable : si un adversaire désire remplacer dans la mémoire une donnée par une autre, il peut librement calculer le nouveau condensé et le substituer également à celui de la donnée manipulée. L'attaque ce sera pas détectée.

Par conséquent, la protection d'intégrité avec les fonctions de hachage nécessite le stockage des condensés à l'intérieur de la zone fiable (SoC) dans une mémoire locale (figure 2.6). Ainsi, si un attaquant veut modifier une donnée D_0, comme il ne peut plus modifier $\mathit{hash}_0 = H(D_0)$, il doit trouver $D'_0 \neq D_0$ tel que $H(D'_0) = \mathit{hash}_0 = H(D_0)$, c'est-à-dire casser la propriété de résistance à la seconde préimage de la fonction de hachage. Pour chaque opération d'écriture, le `hash` calculé sur un groupe de données cible est mis à jour dans la mémoire locale du SoC. La vérification d'intégrité (pendant les opérations de lecture) est effectuée en calculant le `hash` du groupe de données cible et le comparant avec le `hash` stocké dans la mémoire locale.

Cette solution de stockage interne n'est malheureusement pas extensible du fait de la taille nécessairement limitée de la mémoire locale du SoC utilisée pour stocker les condensés. Elle n'est applicable que dans les situations où la taille de la zone mémoire à protéger est limitée et connue lors de la conception du SoC.

Les fonctions les fonctions de MAC (ou les fonctions de hachage à clé) sont plus intéressantes du point de vue du problème du stockage des condensés : l'attaquant ne peut rien calculer

1. Nous utilisons ici, et dans toute la suite du manuscrit, le terme de protection d'intégrité dans le sens de détection des violations d'intégrité. Puisque nous ne pouvons pas empêcher les attaques actives, nous pouvons juste les détecter.

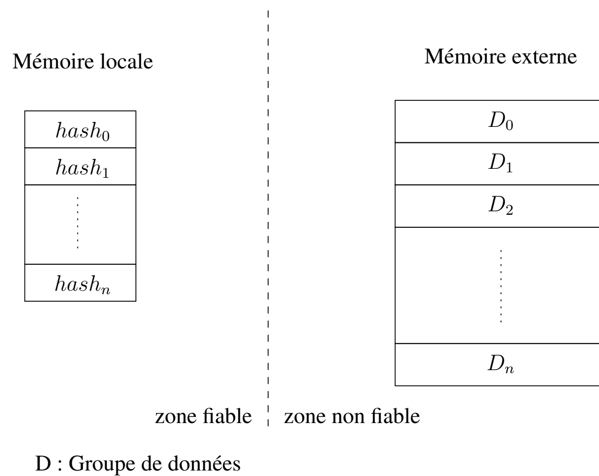


FIGURE 2.6 – Protection d'une zone mémoire contre les attaques par injection

sans clé (stockée à l'intérieur du SoC), ce qui permet de stocker les condensés dans la mémoire externe avec les groupes de données qu'ils protègent (figure 2.7). En effet, l'utilisation de la clé (inaccessible par l'attaquant) par le MAC permet, en plus de la résistance aux attaques par recherche de seconde préimage, la résistance aux attaques par recherche de collisions (trouver deux données différentes avec le même MAC). Par conséquent, l'attaque par injection peut être détectée parce que l'attaquant ne peut pas modifier le groupe de données et son MAC. La comparaison entre les deux MAC, celui stocké dans la mémoire externe et celui généré à partir du groupe de données modifié, va permettre la détection de l'erreur.

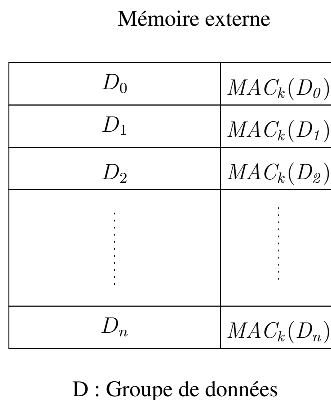


FIGURE 2.7 – Protection d'une zone mémoire contre les attaques par injection

2.2.2.2 Méthodes de protection contre l'attaque par permutation spatiale

La solution proposée contre les attaques par injection en utilisant des MAC, n'est pas suffisant pour protéger l'intégrité contre les attaques par permutation spatiale : un attaquant est capable de permuter deux groupes de données ainsi que leurs MAC sans que le système détecte une erreur dans la comparaison entre les deux MAC comparés (généré et de référence). Une

solution simple pour palier ce problème consiste à utiliser, par exemple, l'adresse du groupe de données lors du calcul du MAC afin d'empêcher un attaquant de copier le groupe de données et son MAC vers une autre adresse de la mémoire. Par exemple, on peut, dans le mode CBC-MAC, utiliser l'adresse comme premier bloc à chiffrer (figure 2.8). Donc, chaque MAC est considéré comme unique pour une adresse donnée. Un chiffrement de bloc supplémentaire est le prix de protection contre l'attaque de permutation spatiale. L'utilisation de l'adresse dans le calcul du MAC ne peut toutefois pas détecter les attaques de type rejeu parce qu'un attaquant peut rejouer un groupe de données et leur MAC sans détection (permutation temporelle entre deux MAC, le dernier mis à jour avec un autre ancien). Le MAC utilisant les adresses ne pourra donc être utilisé que pour les données non modifiables [2] dans la mémoire, qui ne sont pas menacées par le rejeu [3]. Cette possibilité est intéressante car les MAC sont des primitives cryptographiques peu complexes dont les impacts sur les performances et sur l'empreinte mémoire sont moindres que ceux des arbres de Merkle que nous étudions dans la section suivante.

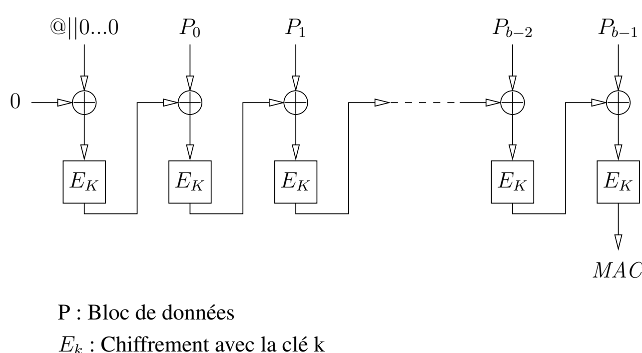


FIGURE 2.8 – Calcul du CBC-MAC avec adresse

2.2.2.3 Méthodes de protection contre l'attaque par rejeu

Les attaques par rejeu sont les plus difficiles à contrer. Une solution naïve est d'ajouter dans le calcul du MAC, en plus de l'adresse, une valeur de circonstance (`nonce`, *number used once*) incrémentée à chaque nouvelle écriture dans la mémoire externe. Les `nonces` ajoutés permettent la détection des attaques par rejeu en distinguant entre deux MAC de la même adresse mais écrits dans la mémoire dans deux temps différents. Cette solution n'est valable que si les `nonces` sont stockés dans la zone sécurisée, afin d'éviter une altération de leurs valeurs. Par contre, le stockage interne des `nonces` est, comme pour les condensés de fonctions de hachage, un point critique car la taille de la mémoire interne est généralement limitée.

Dans la même stratégie de protection, [22] propose une nouvelle méthode de protection en utilisant la technique AREA (*Added Redundancy Explicit Authentication* [23]) au niveau du bloc. Chaque groupe de données (D) est concaténé avec un `nonce` (N) et les deux ($D||N$) sont chiffrés en utilisant un algorithme de chiffrement par bloc pour générer un bloc chiffré. Les blocs chiffrés sont stockés dans la mémoire externe et la clé du chiffrement est stockée

2. On appelle *données non modifiables* les données qui sont écrites une fois, lors de leur initialisation, et qui ne sont ensuite plus jamais modifiées.

3. Rejouer une donnée constante ne peut procurer aucun avantage à un attaquant.

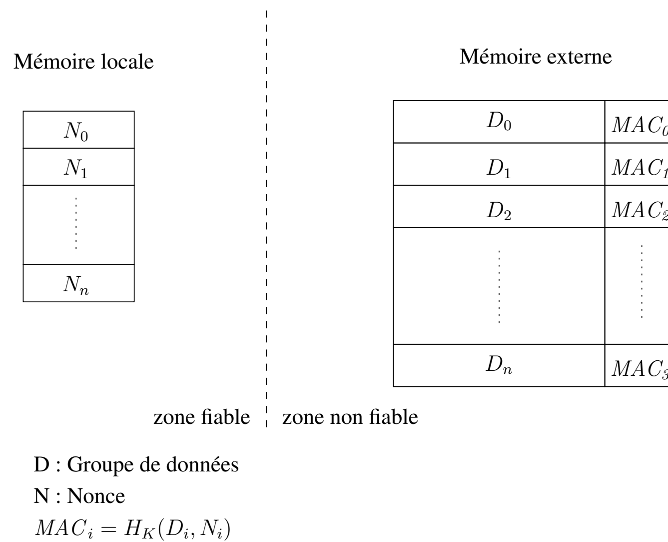


FIGURE 2.9 – Protection de la mémoire externe avec des MAC et `nonce`

à l'intérieur de la puce (figure 2.10). Cette méthode propose de stocker les `nonces` dans une mémoire locale pour protéger la mémoire externe contre les attaques par rejeu. Les avantages de cette méthode est qu'elle permet, en plus de la protection de l'intégrité contre toutes les attaques actives, le chiffrement des données dans la mémoire externe. Par contre, elle augmente le coût du système du fait de la mémoire locale. Les auteurs évaluent un surcout entre 25 % et 50 % en fonction de l'algorithme de chiffrement implémenté.

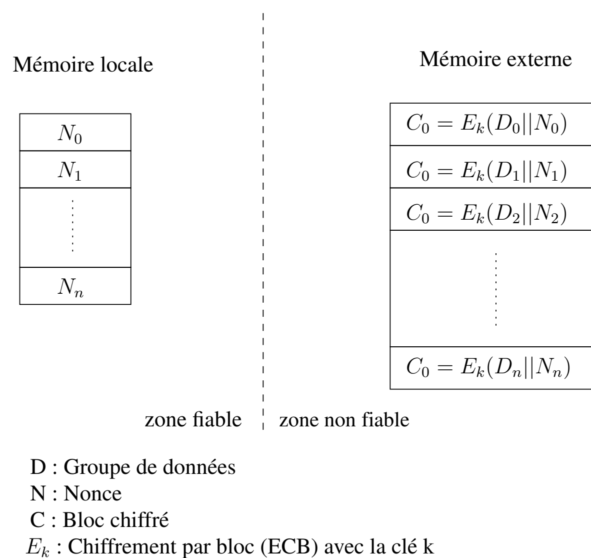


FIGURE 2.10 – Protection de la mémoire externe avec la technique AREA

Les techniques décrites précédemment sont basées sur le stockage des informations (des MAC ou des `nonces`) dans une mémoire digne de confiance à l'intérieur de la puce afin de protéger le système contre les attaques par rejeu. Cette condition impose un surcout du système du fait de la mémoire locale. Par exemple, dans les conditions étudiées par les auteurs de [24],

en considérant une mémoire RAM de 1 Gio, les techniques de MAC (avec `nonces`) et de AREA au niveau bloc, exigent respectivement 128 Mio et 256 Mio de mémoire locale.

Dans la littérature, une technique a été proposée afin d'éviter le stockage dans la mémoire locale. Elle est basée sur l'utilisation des arbres de Merkle, initialement introduits par Merkle *et al.* [25] pour des calculs efficaces dans les cryptosystèmes à clé publique, et ensuite adaptés par Blum *et al.* [26] pour la protection d'intégrité du contenu de mémoires. L'arbre de Merkle (de condensés ou de MAC) protège tout l'ensemble de données de manière hiérarchique selon une organisation en arbre. Les feuilles sont les données à protéger et les nœuds sont les condensés. Tous les nœuds de l'arbre sont stockés dans la zone mémoire non fiable sauf la racine qui est stocké dans une zone sécurisée (figure 2.11).

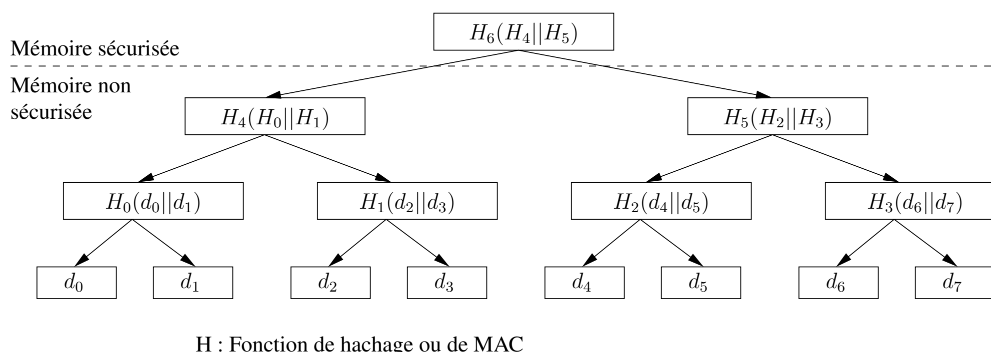


FIGURE 2.11 – Arbre de Merkle

Les arbres de Merkle réguliers sont traités de manière approfondie dans la prochaine section.

2.3 Les arbres de Merkle réguliers

Un Arbre de Merkle Régulier (AMR) est une structure de données permettant de calculer hiérarchiquement un condensé cryptographique d'une zone mémoire (2.12). Il protège les données de la mémoire contre toute altération volontaire ou involontaire, et notamment contre le rejeu. La zone mémoire à protéger est divisée en groupes de données de tailles identiques et chaque groupe est décomposé en blocs. Les blocs correspondent aux feuilles de l'arbre. Chaque nœud (y compris la racine) contient le condensé de ses blocs fils obtenu à l'aide d'une fonction de hachage (avec ou sans clé). La racine de l'arbre est stockée dans une zone protégée et inaccessible par un adversaire. Quand aux nœuds, ils peuvent être stockés dans la zone mémoire non fiable, et toute altération avec ces nœuds ou des feuilles sera détectée, grâce au condensé racine.

La figure 2.12 présente un arbre de Merkle régulier, d'arité a et de profondeur s, qui protège un ensemble de a^s blocs de données. Les nœuds de l'arbre sont numérotés (i, j) où i est l'indice du niveau et j l'indice du nœud dans chaque niveau. Le niveau $i = 0$ correspond au niveau des données et le niveau $i = s$ correspond au niveau de la racine. Les nœuds dans un niveau i sont

numérotés de $(i, 0)$ à $(i, a^{s-i} - 1)$. j peut aussi être représenté par $a \times j_0 + j_1$ où j_0 est l'indice du groupe et j_1 est l'indice du nœud dans son groupe.

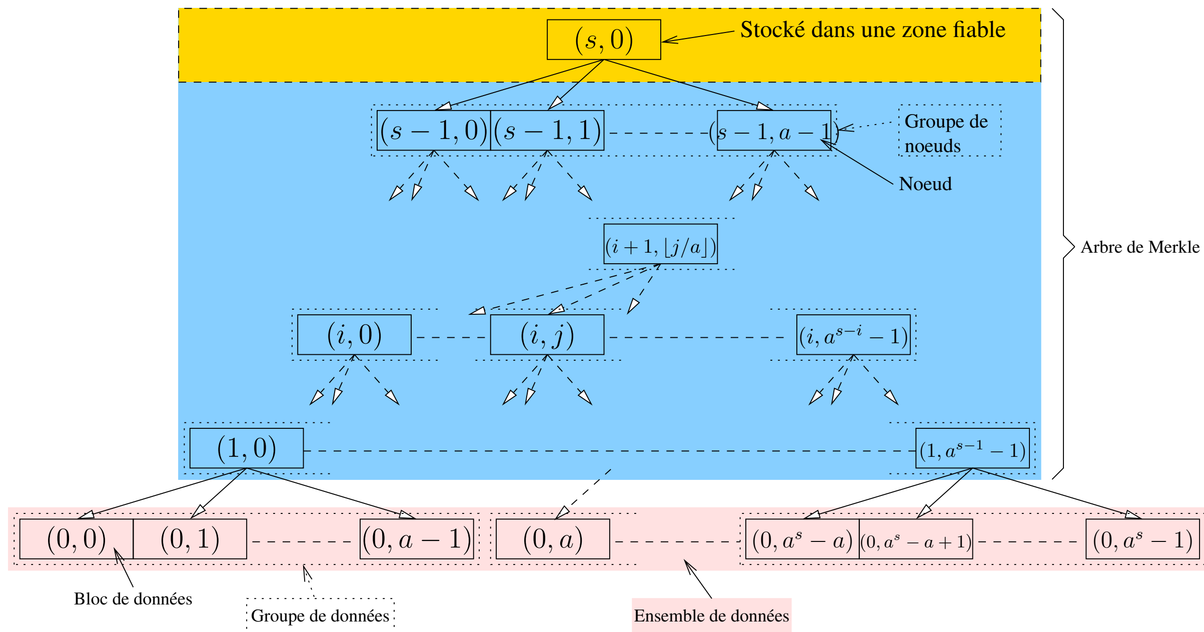


FIGURE 2.12 – Arbre de Merkle protégeant l'intégrité d'un ensemble de données

Trois opérations permettent de manipuler les AMR :

- `Initialisation` : permet d'initialiser l'arbre avant d'effectuer des accès de lecture et d'écriture dans la mémoire externe.
- `Vérification` ou lecture vérifiée : permet de lire une valeur et vérifier son intégrité.
- `Mise à jour` ou écriture vérifiée : permet de mettre à jour une valeur en mémoire.

Par la suite, nous allons détailler ces trois opérations et estimer leur complexité. Trois fonctions sont introduites :

— `Read(i,j)` : la lecture régulière non vérifiée de la valeur du nœud (i, j).

— `Write(i,j,V)` : l'écriture régulière non vérifiée de la valeur V dans le nœud (i, j).

— `Digest`$(i, j, X_0, X_1, ..., X_{a-1})$: le calcul du condensé destiné à être stocké dans le nœud (i, j) avec les valeurs concaténées de ses nœuds fils $X_0, X_1, ..., X_{a-1}$.

2.3.1 Opérations

2.3.1.1 Initialisation

L'intégrité de l'ensemble des données n'est protégée que lorsque l'AMR est complètement initialisé. Un attaquant peut falsifier les données avant ou pendant le processus d'initialisation sans être détecté. La protection de l'ensemble des données et des nœuds de l'arbre repose sur la racine de l'arbre (stockée dans la zone sécurisée) qui est la seule référence fiable pour vérifier l'intégrité, et tant que cette racine n'est pas encore construite, la protection ne peut être garantie. En conséquence, les données utilisées pour construire l'AMR sont considérées comme non

Algorithme 1 `Init-Régulier`

```
1: for i ∈ {1, ..., s} do                          ▷ Pour les niveaux d'arbre sauf les données
2:     for j_0 ∈ {0, ..., a^{s−i} − 1} do          ▷ Pour les nœuds d'un niveau
3:         for k ∈ {0, ..., a − 1} do              ▷ Pour les a nœuds fils
4:             X_k ← Read(i − 1, a × j_0 + k)      ▷ Lecture non vérifiée
5:         end for
6:         D ← Digest(i, j_0, X_0, ..., X_{a−1})   ▷ Calcul du nœud père
7:         Write(i, j_0, D)                        ▷ Stocker le nœud
8:     end for
9: end for
```

fiables. Seules les données qui sont stockées *après* l'initialisation complète de l'AMR peuvent être dignes de confiance.

L'initialisation de l'AMR est un processus itératif qui calcule les condensés cryptographiques de chaque groupe de nœuds de l'arbre, depuis le niveau juste au dessus des feuilles (données) jusqu'à la racine, comme illustré par l'algorithme 1. Comme déjà indiqué, le dernier nœud (la racine de l'arbre) est stocké (ligne 7) dans une zone sécurisée tandis que les autres nœuds peuvent être stockés dans la zone non sécurisée.

2.3.1.2 Vérification

À chaque lecture d'un nœud (i, j), l'AMR est utilisé pour vérifier son intégrité. Le processus de vérification consiste à lire le nœud demandé ainsi que ses $(a - 1)$ nœuds frères, calculer leur condensé et le comparer avec le nœud père. Si la comparaison échoue, une erreur est levée, l'opération est abandonnée et des actions appropriées sont prises pour traiter l'attaque détectée. Sinon, le processus de vérification passe au niveau supérieur et continue jusqu'à ce qu'une comparaison échoue ou que la racine soit calculée et comparée avec succès avec la référence fiable. Si toutes les comparaisons réussissent, la valeur demandée est retournée. L'algorithme 2 présente la fonction de `Lecture-Vérifiée` du nœud $(i \neq s, a \times j_0 + j_1)$ [4]. La même opération peut aussi être représentée de manière récursive mais la version itérative est souvent plus facile à implémenter et plus efficace, particulièrement en matériel.

2.3.1.3 Mise à jour

L'opération de mise à jour d'AMR est utilisée chaque fois un nœud (i, j) est modifié (écrit). Toute la branche au dessus du nœud (i, j) doit être mise à jour jusqu'à la racine. Ce processus est une combinaison itérative de lectures, vérifications et d'écritures. Il commence à partir du nœud à écrire, lit l'ancienne valeur du nœud ainsi que ses $(a - 1)$ frères depuis la zone non sécurisée et calcule deux condensés avec l'ancienne et la nouvelle valeur du nœud (i, j). L'ancien condensé est comparé avec le nœud père. Si la comparaison échoue, une erreur est levée. Sinon, la donnée cible est écrite et le même processus se répète pour l'écriture du nouveau condensé dans le nœud père. Le processus se termine lorsque la racine est lue et mise à jour. L'algorithme 3 détaille, de manière itérative, la fonction d'`Écriture-Vérifiée` de la valeur V dans le nœud $(i \neq s, j = a \times j_0 + j_1)$.

4. $i \neq s$ parce que, comme il est stocké dans une zone sécurisée, la lecture de $(s, 0)$ ne nécessite pas la vérification

Algorithme 2 `Lecture-Vérifiée`$(i \neq s, j = a \times j_0 + j_1)$

```
1: (u, v0, v1) ← (i, j0, j1)                          ▷ Indices temporaires
2: for k ∈ {0, ..., a − 1} do                         ▷ Pour a nœuds frères
3:     X_k ← Read(u, a × v0 + k)                      ▷ Lecture non vérifiée
4: end for
5: R ← X_{v1}                                         ▷ La valeur à retourner
6: D ← Digest(u + 1, v0, X_0, ..., X_{a−1})           ▷ Calcul du nœud père
7: (u, v0, v1) ← (u + 1, ⌊v0/a⌋, v0 mod a)            ▷ Avancer au niveau supérieur
8: while u < s do                                     ▷ Pour les niveaux d'arbre
9:     for k ∈ {0, ..., a − 1} do                     ▷ Pour a nœuds frères
10:        X_k ← Read(u, a × v0 + k)                  ▷ Lecture non vérifiée
11:    end for
12:    if X_{v1} ≠ D then                             ▷ Si la comparaison échoue
13:        error                                      ▷ Abandonner
14:    end if
15:    D ← Digest(u + 1, v0, X_0, ..., X_{a−1})       ▷ Calcul du nœud père
16:    (u, v0, v1) ← (u + 1, ⌊v0/a⌋, v0 mod a)        ▷ Avancer au niveau supérieur
17: end while
18: X_0 ← Read(s, 0)                                  ▷ Lire à partir de la zone sécurisée
19: if X_0 ≠ D then                                   ▷ Si la comparaison échoue
20:     error                                         ▷ Abandonner
21: end if
22: return R                                          ▷ Retourner le nœud requis et arrêter
```

2.3.2 Évaluation des performances

2.3.2.1 Notations

Nous prendrons le cas de l'utilisation d'un AMR pour protéger la mémoire externe d'un système embarqué comme exemple pour évaluer les performances. Le processeur à l'intérieur du SoC communique avec la mémoire externe à travers, dans l'ordre :

- le réseau de communication interne du SoC (interne, donc protégé),
- le contrôleur des arbres de Merkle (interne, donc protégé),
- le contrôleur de mémoire externe (interne, donc protégé) et
- le bus de communication entre le SoC et la mémoire externe (exposé).

Le contrôleur des arbres de Merkle se situe donc à l'intérieur du SoC entre le réseau interne et le contrôleur de mémoire externe.

Nous introduisons les notations suivantes :

- MAC est la fonction utilisée pour calculer le condensé cryptographique d'un groupe de nœuds (ou de données)
- l est la taille d'un nœud d'arbre exprimée en nombre de mots mémoire (par exemple $l = 2$ pour des nœuds de 64 bits et un bus mémoire de 32 bits)
- t_0 est la latence initiale de lecture, en nombre de cycles d'horloge, de la mémoire externe, entre l'émission de la première requête de lecture sur le bus mémoire et le début de la réception la première donnée lue
- t_l est le temps (additionnel à t_0), en nombre de cycles d'horloge, nécessaire au transfert d'un nœud de l'arbre
- Lire un groupe de a nœuds ou blocs de données prend donc $t_{rg} = t_0 + t_l \times a$ cycles d'horloge, entre l'émission de la première requête de lecture sur le bus mémoire et la réception de la dernière donnée lue
- Écrire un nœud ou bloc de données prend donc $t_{wn} = t_l$ cycles d'horloge, entre les

Algorithme 3 Écriture-Vérifiée$((i \neq s, j = a \times j_0 + j_1), V)$

1: $(u, v_0, v_1) \leftarrow (i, j_0, j_1)$ ▷ Indices temporaires
2: **for** $k \in \{0, ..., a-1\}$ **do** ▷ Pour a nœuds frères
3: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Lecture non vérifiée
4: **end for**
5: $D_{old} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père ancien
6: $X_{v_1} \leftarrow V$ ▷ La nouvelle valeur du nœud
7: $\text{Write}((u, a \times v_0 + v_1), X_{v_1})$ ▷ Écriture non vérifiée de la nouvelle valeur du nœud
8: $D_{new} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père nouveau
9: $(u, v_0, v_1) \leftarrow (u+1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ Avancer au niveau supérieur
10: **while** $u < s$ **do** ▷ Pour les niveaux d'arbre
11: **for** $k \in \{0, ..., a-1\}$ **do** ▷ Pour a nœuds frères
12: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Lecture non vérifiée
13: **end for**
14: **if** $X_{v_1} \neq D_{old}$ **then** ▷ Si la comparaison échoue
15: **error** ▷ Abandonner
16: **end if**
17: $D_{old} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père ancien
18: $X_{v_1} \leftarrow D_{new}$ ▷ La nouvelle valeur du nœud
19: $\text{Write}((u, a \times v_0 + v_1), X_{v_1})$ ▷ Écriture non vérifiée de la nouvelle valeur du nœud
20: $D_{new} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père nouveau
21: $(u, v_0, v_1) \leftarrow (u+1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ Avancer au niveau supérieur
22: **end while**
23: $X_0 \leftarrow \text{Read}(s, 0)$ ▷ Lire à partir de la zone sécurisée
24: **if** $X_0 \neq D_{old}$ **then** ▷ Si la comparaison échoue
25: **error** ▷ Abandonner
26: **end if**
27: $\text{Write}((s, 0), D_{new})$ ▷ Écrire la nouvelle racine dans la zone sécurisée

émissions de la première et de la dernière requête d'écriture sur le bus mémoire [5]
- t_{MAC} est le temps de calcul d'un MAC, en nombre de cycles d'horloge
- Nous considérons que la latence pour lire ou écrire la racine à l'intérieur de la puce a un impact négligeable sur les performances, en comparaison de la latence de la mémoire externe qui est typiquement de l'ordre de centaines de cycles d'horloge dans les systèmes à moyenne ou haute performance
- Nous considérons que la latence mémoire est au minimum deux fois plus grande que le temps de calcul d'un MAC ($t_0 > 2 \times t_{MAC}$)

Un arbre de Merkle équilibré, d'arité a et de profondeur s, protège exactement a^s blocs de données et il contient un seul nœud racine dans son dernier niveau. Pour un arbre de Merkle déséquilibré, le dernier niveau contient plus d'un nœud. La figure 2.13 présente un arbre de Merkle déséquilibré quaternaire qui protège un ensemble de données de 128 blocs (différent de a^s), ce qui donne deux nœuds dans le dernier niveau de l'arbre. Le nœud racine de ce type d'arbre se calcule en regroupant un ensemble d'arbres déséquilibrés et en rassemblant les nœuds de leurs derniers niveaux dans un seul groupe, même si le nombre de nœuds est différent de l'arité a. Cette propriété sera utilisée par la suite et on suppose que le temps nécessaire pour calculer le MAC de ce dernier niveau est le même que celui pris pour un groupe complet de a nœuds.

Pour l'évaluation des performances, nous considérons un système avec les caractéristiques suivantes :

- Système à base d'ARM-v7 (32 bits) à 1 GHz, avec des lignes de cache de 256 bits
- Une mémoire externe de 4 Gio, 32 bits, un bus mémoire 32 bits utilisant les deux fronts de son horloge à 500 MHz (soit un débit pic de données de 4 Gio/s), une latence de

5. On suppose que la mémoire n'envoie pas de réponse lors des requêtes d'écriture.

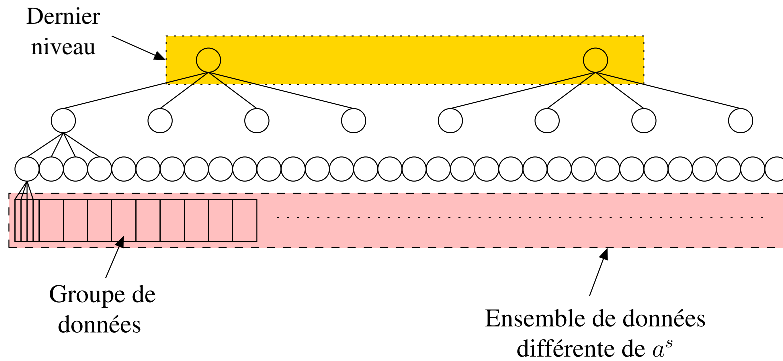


FIGURE 2.13 – Arbre de Merkle déséquilibré protégeant l'intégrité d'un ensemble de données

$t_0 = 100$ cycles d'horloge du processeur (à 1 GHz)

- L'algorithme de chiffrement par bloc utilisé pour calculer le MAC est DES-X, les blocs de données ont une taille de 64 bits ($l = 2$)
- L'arité des arbres de Merkle est $a = 4$ (une ligne de cache)
- Un calcul de DES-X prend 4 cycles d'horloge du processeur, un calcul de MAC prend $t_{MAC} = 20 = 4 \times (1 + a)$ (une adresse plus a blocs de données)
- Il existe quatre tailles de page mémoire différentes, toutes sont protégées avec des arbres déséquilibrés, avec une super racine
  — 4 Kio, $s = 4 + 1 = 5$
  — 64 Kio, $s = 6 + 1 = 7$
  — 1 Mio, $s = 8 + 1 = 9$
  — 16 Mio, $s = 10 + 1 = 11$

2.3.2.2 Initialisation

La fonction itérative, utilisée pour initialiser l'AMR (voir page 48), demande une lecture régulière du groupe de nœuds (lignes 3 à 5), un calcul de MAC (ligne 6) et une écriture régulière du nœud père (ligne 7).

Le temps de calcul du MAC t_{MAC} est négligeable puisque ce calcul peut être effectué en parallèle avec la lecture régulière du groupe de nœuds suivant et que le temps nécessaire pour ce calcul est plus petit que la latence mémoire (figure 2.14). Néanmoins, l'avant dernier calcul de MAC (le nœud $(s - 1, a - 1)$) ne peut pas être parallélisé avec la lecture du groupe suivant $((s - 1, 0), \ldots, (s - 1, a - 1))$ puisqu'il en fait partie (il doit être écrit avant que la lecture de ce groupe soit possible). De même, le calcul du MAC du nœud racine $(s, 0)$ ne peut pas être parallélisé avec une autre opération.

Nous considérons aussi que les opérations d'écriture ne peuvent pas être effectuées en parallèle avec les opérations de lectures régulières car elles utilisent les mêmes ressources matérielles. Par contre, l'écriture dans le nœud $(s - 1, a - 2)$ peut être effectuée en parallèle avec le calcul de MAC du nœud $(s - 1, a - 1)$. La figure 2.15 présente les différentes opérations réalisées pendant l'initialisation complète de l'arbre.

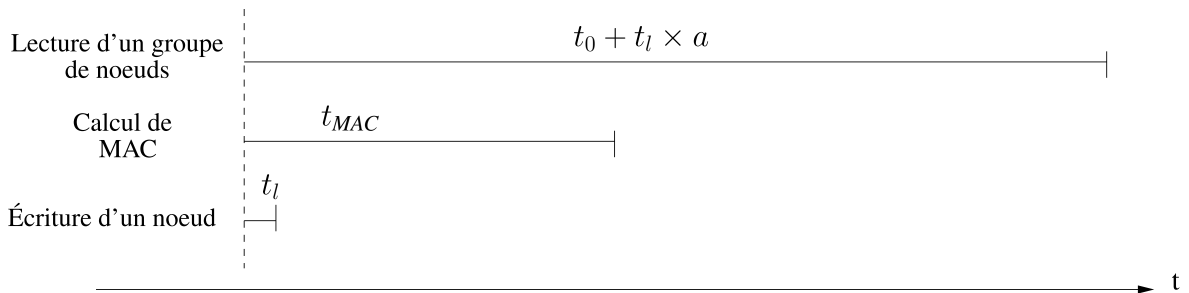


FIGURE 2.14 – Comparaison entre le temps pris pour une lecture d'un groupe de nœuds, un calcul de MAC et un écriture d'un nœud

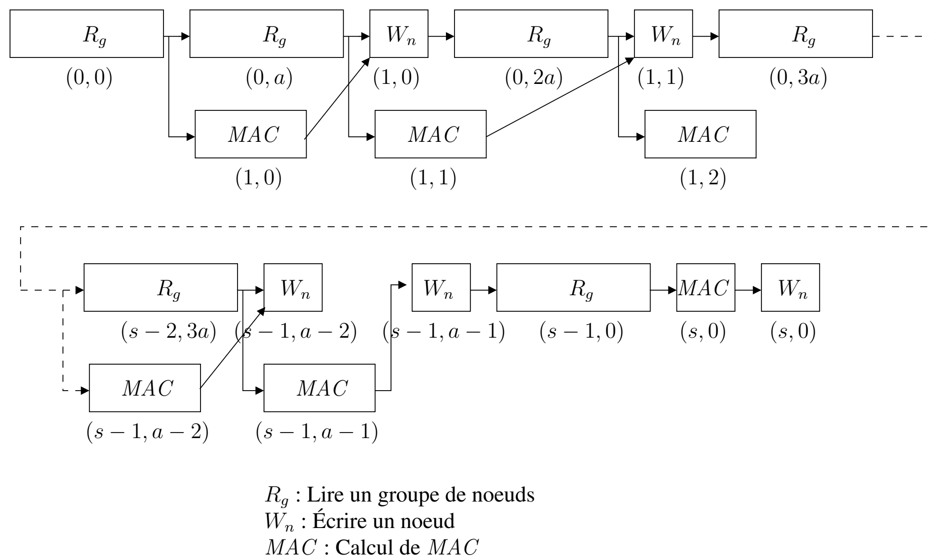


FIGURE 2.15 – Latences des différentes opérations de l'initialisation d'arbre

Le temps pris par l'initialisation du premier niveau ($i = 1$) est donc (il faut noter que la dernière écriture, celle du nœud $(1, a^{s-1} - 1)$ a lieu après la lecture du premier bloc du niveau suivant $((1, 0), \ldots, (1, a - 1)))$:

$$t_{i_1} = (t_{rg} + t_{wn}) \times a^{s-1}$$

Pour les niveaux intermédiaires i avec $1 \leq i < s-1$ (la même remarque que précédemment s'applique) :

$$t_{i_i} = (t_{rg} + t_{wn}) \times a^{s-i}$$

Pour l'avant dernier niveau ($i = s - 1$) (ici la dernière écriture ne peut pas être réordonnée avec la lecture suivante) :

$$t_{i_{s-1}} = (t_{rg} + t_{wn}) \times (a - 1) + t_{rg} + t_{MAC}$$

Enfin, pour l'initialisation du nœud racine :

$$t_{i_s} = t_{rg} + t_{MAC}$$

Au total, pour l'initialisation de tous les niveaux,

$$\begin{aligned} t_i &= (t_{rg} + t_{wn}) \times \sum_{k=2}^{s-1} a^k + (t_{rg} + t_{wn}) \times (a - 1) + t_{rg} + t_{MAC} + t_{rg} + t_{MAC} \\ &= (t_{rg} + t_{wn}) \times \sum_{k=2}^{s-1} a^k + (t_{rg} + t_{wn}) \times (a - 1) + 2 \times (t_{rg} + t_{MAC}) \end{aligned}$$

Avec les différentes tailles des pages utilisées dans notre exemple à base d'ARM, nous obtenons (en cycles d'horloge du processeur), les valeurs suivantes :

- 4 Kio : $t_i \approx 0,044 \cdot 10^6$ cycles
- 64 Kio : $t_i \approx 0,710 \cdot 10^6$ cycles
- 1 Mio : $t_i \approx 11,359 \cdot 10^6$ cycles
- 16 Mio : $t_i \approx 181,753 \cdot 10^6$ cycles

Optimisation Nous pouvons imaginer un autre scénario d'initialisation d'arbre qui va permettre de supprimer totalement les lectures régulières en mémoire externe. Tout d'abord, les blocs de données étant initialement considérés comme non fiables, il est possible de les initialiser (par exemple à zéro) pendant l'initialisation de l'arbre, ce qui évite les lectures de données et les remplacent par des écritures, que l'on peut parfaitement paralléliser avec les calculs cryptographiques.

Pour les lectures des nœuds de l'arbre, ce scénario est basé sur une organisation intelligente du parcours d'arbre permettant de calculer les MAC partiellement et de reprendre le calcul dès que le bloc suivant est disponible. Une fois qu'un calcul de MAC est terminé, sa valeur est écrite dans la mémoire externe dans le nœud correspondant. Il est donc nécessaire de stocker

Algorithme 4 `Init-Optimisée`

```
1: j_0 ← 0                                   ▷ Numéro du nœud de niveau 1 en cours de traitement
2: while j_0 ≠ a^{s−1} do                    ▷ Pour les nœuds de niveau 1 (juste au dessus des données)
3:     D_1 ← Digest(1, j_0, 0, 0, ..., 0)    ▷ Calcul du nœud (1, j_0) avec données nulles
4:     Write((1, j_0), D_1)                  ▷ Écriture non vérifiée du nœud
5:     (u, v) ← (1, j_0)                     ▷ Coordonnées du nœud courant
6:     done ← true                           ▷ On vient de finir le calcul d'un nœud
7:     while done do                         ▷ Tant que l'on vient de terminer le calcul d'un nœud, avancer le calcul de son père
8:         done ← false                      ▷ On ré-initialise le marqueur de fin de calcul d'un nœud
9:         (w, t) ← (u + 1, ⌊v/a⌋)           ▷ Coordonnées du nœud père
10:        if v mod a = 0 then               ▷ Le nœud fils est-il le premier de son bloc ?
11:            D_w ← DigestInit(w, t, D_u)   ▷ Si oui, début du calcul de son père
12:        else
13:            D_w ← DigestUpdate(D_w, D_u)  ▷ Sinon, suite du calcul de son père
14:        end if
15:        if v mod a = a − 1 then           ▷ Le nœud fils est-il le dernier de son bloc ?
16:            Write((w, t), D_w)            ▷ Si oui, calcul du père est terminé donc écriture du père
17:            done ← true                   ▷ On vient de finir le calcul d'un nœud
18:        end if
19:        (u, v) ← (w, t)                   ▷ On avance au niveau suivant
20:    end while
21:    j_0 ← j_0 + 1                         ▷ On avance au nœud suivant
22: end while
23: for j ∈ {0, ..., a^s − 1} do
24:     Write(0, j, 0)                       ▷ Initialisation des blocs de données avec la valeur zéro
25: end for
```

en interne un bloc par MAC en cours de calcul. Pour un arbre équilibré à N niveaux, le coût est, au plus, de N blocs de stockage interne.

L'algorithme 4 présente l'opération optimisée d'initialisation. Les écritures régulières des blocs de données (ligne 24) peuvent être parallélisées avec les calculs de MAC (ligne 3) et les écritures régulières des nœuds (lignes 4 et 16) peuvent également être parallélisées avec les calculs partiels de MAC (lignes 11, 13).

On suppose que le calcul du MAC peut être effectué progressivement à l'aide des fonctions $\mathrm{DigestInit}$ et $\mathrm{DigestUpdate}$ (on suppose que le temps de calcul complet est toujours égal à t_{MAC}) :

$$
\begin{aligned}
D_0 &= \mathrm{DigestInit}(i, j, X_0) \\
D_1 &= \mathrm{DigestUpdate}(D_0, X_1) \\
D_2 &= \mathrm{DigestUpdate}(D_1, X_2) \\
&\cdots \\
\mathrm{Digest}(i, j, X_0, \ldots, X_{a-1}) &= \mathrm{DigestUpdate}(D_{a-2}, X_{a-1})
\end{aligned}
$$

Un tel découpage est facilement réalisable dans le cas de CBC-MAC.

Au total, le temps pris par cette opération optimisée d'initialisation pour tous les niveaux est :

$$t_i = t_{MAC} \times \sum_{k=0}^{s-1} a^k$$

Avec les différentes tailles des pages utilisées dans notre exemple à base d'ARM, nous obtenons (en cycles d'horloge du processeur), les valeurs suivantes :

- 4 Kio : $t_i \approx 6.780$ cycles

- 64 Kio : $t_i \approx 0,109 \cdot 10^6$ cycles
- 1 Mio : $t_i \approx 1,747 \cdot 10^6$ cycles
- 16 Mio : $t_i \approx 27,962 \cdot 10^6$ cycles

2.3.2.3 Lecture vérifiée

La première itération de la lecture vérifiée d'un nœud (i, j), différent de la racine, avec l'algorithme 2 demande une lecture régulière du groupe (lignes 2 à 4) et un calcul de MAC (ligne 6). De même pour les itérations suivantes (lignes 9 à 11 et ligne 15). À chaque itération le temps de calcul du MAC est négligeable puisque ce calcul peut être effectué en parallèle avec la lecture régulière du groupe suivant, à l'exception du calcul du dernier MAC, qui correspond à la racine, et qui n'est pas parallélisable avec une lecture (figure 2.16).

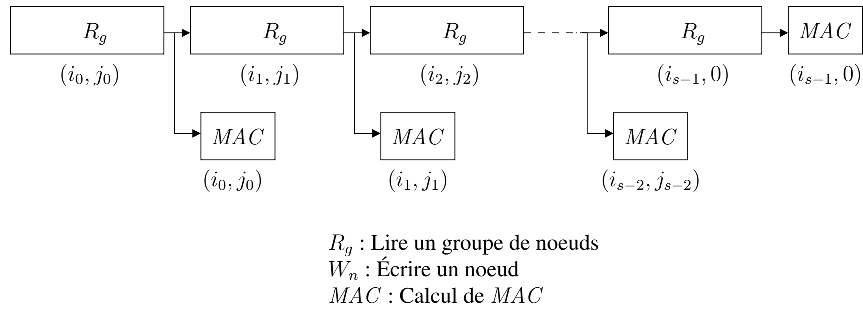


FIGURE 2.16 – Latences des différentes opérations d'une lecture vérifiée

Le temps t_{r_i} nécessaire pour effectuer une lecture vérifiée d'un bloc au niveau $i < s - 1$ est donc :

$$t_{r_i} = t_{rg} + t_{r_{i+1}}$$

tandis que

$$t_{r_{s-1}} = t_{rg} + t_{MAC} + t_{r_s}$$

avec, par hypothèse, $tr_s = 0$.

Au total, pour la lecture d'un groupe de a blocs de données,

$$t_{r_0} = s \times t_{rg} + t_{MAC}$$

Le temps pris par cette opération en l'absence de contrôle d'intégrité, est de t_{rg}. L'algorithme de lecture vérifiée introduit donc une latence supplémentaire (en nombre de cycles) de :

$$(s - 1) \times t_{rg} + t_{MAC}$$

c'est-à-dire un facteur d'accroissement de :

$$(s - 1) + \frac{t_{MAC}}{t_{rg}}$$

Avec les différentes tailles des pages utilisées dans notre exemple à base d'ARM, nous obtenons, pour une lecture d'une ligne de cache, les résultats suivants :

- 4 Kio : surcout en temps de $419\,\%$
- 64 Kio : surcout en temps de $619\,\%$
- 1 Mio : surcout en temps de $819\,\%$
- 16 Mio : surcout en temps de $1019\,\%$

2.3.2.4 Écriture vérifiée

La première itération d'une écriture vérifiée d'un nœud (i, j), différent de la racine, avec l'algorithme 3 demande une lecture régulière du groupe de nœuds (lignes 2 à 4), une écriture régulière du nouveau nœud (ligne 7) et deux calculs de MAC (lignes 5 et 8). De même pour les itérations suivantes (lignes 11 à 13, 19, 17 et 20).

Après la lecture d'un groupe au niveau i, les deux calculs de MAC (ancienne et nouvelle versions du nœud père de niveau $i + 1$) peuvent s'effectuer en parallèle avec l'écriture de la nouvelle valeur du nœud de niveau i et la lecture régulière d'un groupe de niveau $i + 1$, à l'exception du dernier calcul de MAC (la racine), non parallélisable (figure 2.17).

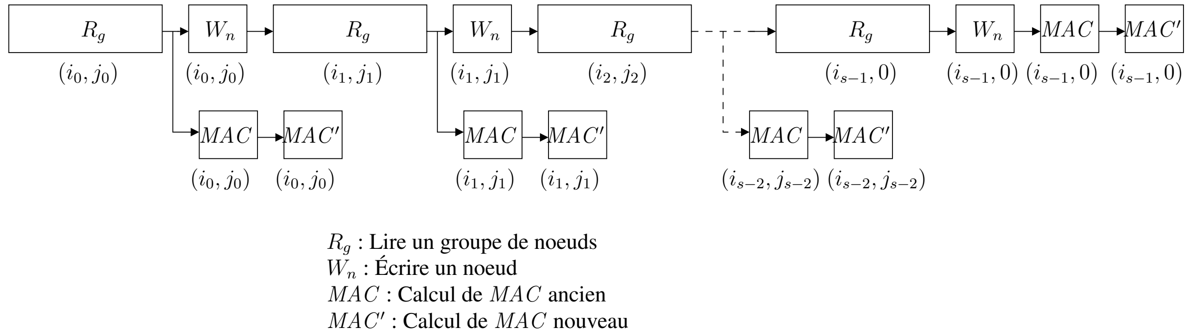


FIGURE 2.17 – Latences des différentes opérations d'une écriture vérifiée

Le temps t_{w_i} pris pour effectuer une écriture vérifiée au niveau $i < s - 1$ est donc :

$$t_{w_i} = t_{rg} + t_{wn} + t_{w_{i+1}}$$

tandis que

$$t_{w_{s-1}} = t_{rg} + t_{wn} + 2 \times t_{MAC} + t_{w_s}$$

avec, par hypothèse, $t_{w_s} = 0$.

Au total,

$$t_{w_0} = s \times (t_{rg} + t_{wn}) + 2 \times t_{MAC}$$

Le temps pris par cette opération en l'absence de contrôle d'intégrité est de t_{wn}. La latence supplémentaire introduite par l'algorithme d'écriture vérifiée est donc :

$$(s - 1) \times (t_{rg} + t_{wn}) + t_{rg} + 2 \times t_{MAC}$$

c'est-à-dire un facteur d'accroissement de :

$$(s - 1) \times \frac{(t_{rg} + t_{wn})}{t_{wn}} + \frac{t_{rg} + 2 \times t_{MAC}}{t_{wn}}$$

Avec les différentes tailles de pages utilisées dans notre exemple à base d'ARM, nous obtenons, pour une écriture d'une ligne de cache (en cycles d'horloge du processeur), les facteurs suivants :

- 4 Kio : surcout en temps de $29400\,\%$
- 64 Kio : surcout en temps de $40400\,\%$
- 1 Mio : surcout en temps de $51400\,\%$
- 16 Mio : surcout en temps de $62400\,\%$

2.3.3 Variantes des arbres de Merkle

Plusieurs variantes basées sur les arbres de Merkle ont été proposées dans la littérature.

2.3.3.1 Parallelizable Authentication Tree

Les arbres d'authentification parallélisables (PAT, *Parallelizable Authentication Tree*) [27] utilisent une fonction de MAC. Chaque nœud de l'arbre contient un MAC et un `nonce`. Les MAC du premier niveau de l'arbre sont calculés à partir des groupes de données en utilisant une clé (stockée à l'intérieur de la puce) et un `nonce` correspondant, c'est-à-dire, $MAC_{K,N}(d_0||...||d_n)$ où K est la clé, N le `nonce` et les d_i les blocs de données. Ensuite, les MAC des niveaux supérieurs sont calculés à partir des groupes de `nonces` qui se trouvent dans les niveaux inférieurs (figure 2.18). Les MAC et les `nonces` sont stockés dans la mémoire externe, à l'exception du dernier `nonce` de la racine qui est stocké à l'intérieur de la puce (et le MAC correspondant est stocké dans la mémoire externe). Avec cette technique, un attaquant ne peut pas générer un MAC sans clé et ne peut pas rejouer un ancien MAC sans le `nonce` stocké à l'intérieur de la puce.

Cette technique permet d'effectuer des opérations de vérification et de mise à jour parallélisables. En effet, toutes les données et `nonces` nécessaires pour la vérification des nœuds d'une branche sont disponibles, le calcul des valeurs de nœuds pour mettre à jour une branche, s'effectue à partir des `nonces` et il n'est donc pas nécessaire d'attendre le calcul du MAC du niveau inférieur.

L'utilisation d'un PAT augmente les besoins en espace mémoire du système d'un facteur $\frac{3}{2\times(a-1)}$ (où a est l'arité de l'arbre) par rapport à la même application n'utilisant aucun mécanisme de protection.

2.3.3.2 TEC-Tree

Une autre technique basée sur les arbres de Merkle est le *Tamper-Evident Counter Tree* (TEC-Tree) [28]. La primitive d'authentification utilisée est la fonction AREA au niveau du bloc [22] : chaque bloc de données (D) est concaténé avec un `nonce` (N) et les deux ($D||N$) sont chiffrés en utilisant un algorithme de chiffrement par bloc pour générer un bloc chiffré. La technique TEC-Tree permet aussi des opérations de vérification et de mise à jour parallélisables,

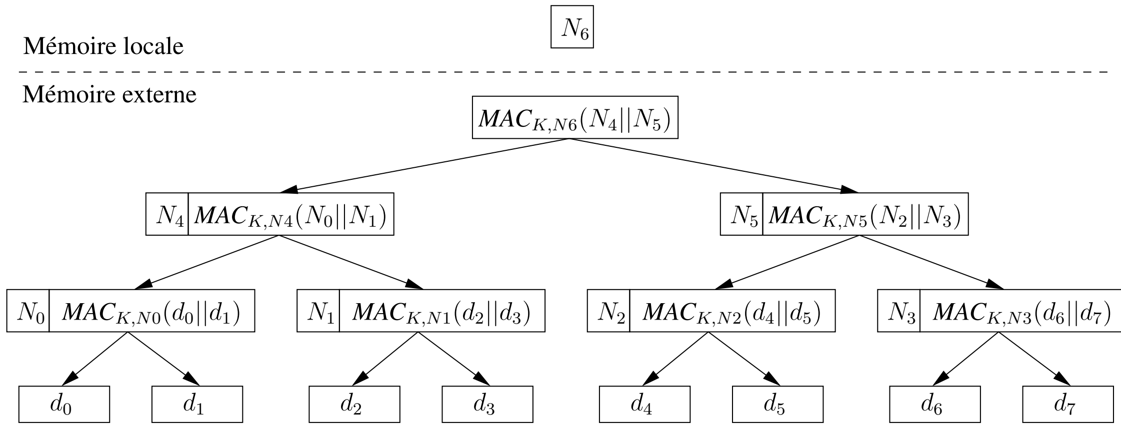


FIGURE 2.18 – *Parallelizable Authentication Tree* (PAT)

parce que les fonctions AREA sont basées sur des entrées indépendantes : les `nonces`. Une autre caractéristique de cette technique est qu'elle permet, en plus de la protection d'intégrité, la protection de la confidentialité des données de la mémoire externe.

Chaque bloc de données (D) est concaténé avec un `nonce` (N, *Number used ONCE*) pour constituer un bloc nommé *Data Chunk* (*DC*). Ces *DC*s représentent les feuilles de l'arbre. Le `nonce` est composé de l'adresse du bloc de données (*addr*) concaténée avec un compteur (*CTR*). Initialisé à zéro, un compteur est incrémenté à chaque fois qu'un bloc de données est écrit dans la mémoire par le SoC. La figure 2.19 représente un *DC* avant le chiffrement.

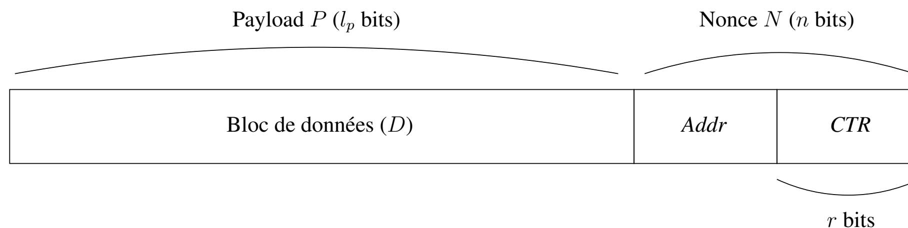


FIGURE 2.19 – Configuration d'un *DC* avant le chiffrement

Chaque nœud du premier niveau de l'arbre est obtenu en calculant la fonction AREA à partir des compteurs de a *DC*s et en utilisant une clé (K) et le `nonce` (N) correspondant. Ensuite, les nœuds des niveaux supérieurs sont obtenus à partir des compteurs de a nœuds des niveaux inférieurs. La figure 2.20 représente un nœud intermédiaire ou *Counter Chunk* (*CC*) avant le chiffrement.

Tous les nœuds de l'arbre sont stockés dans la mémoire externe, à l'exception du dernier compteur de la racine qui est stocké à l'intérieur de la puce (et la fonction AREA correspondante est stockée dans la mémoire externe). Le compteur de la racine joue le rôle de la référence fiable interdisant les attaques par rejeu. La figure 2.21 représente un arbre binaire de TEC-Tree.

Les auteurs de TEC-Tree définissent deux fonctions pour une lecture et écriture vérifiées d'un bloc de données D : *ReadAndCheck* et *WriteAndUpdate*.

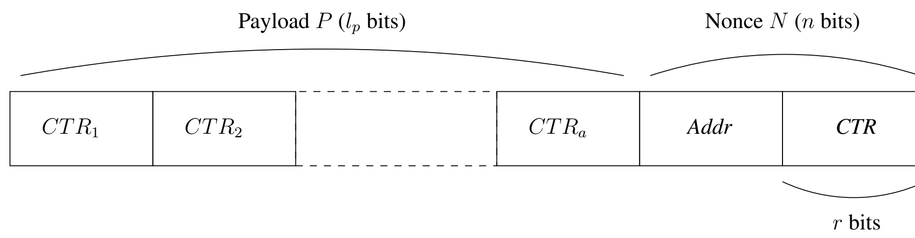


FIGURE 2.20 – Configuration d'un CC avant le chiffrement

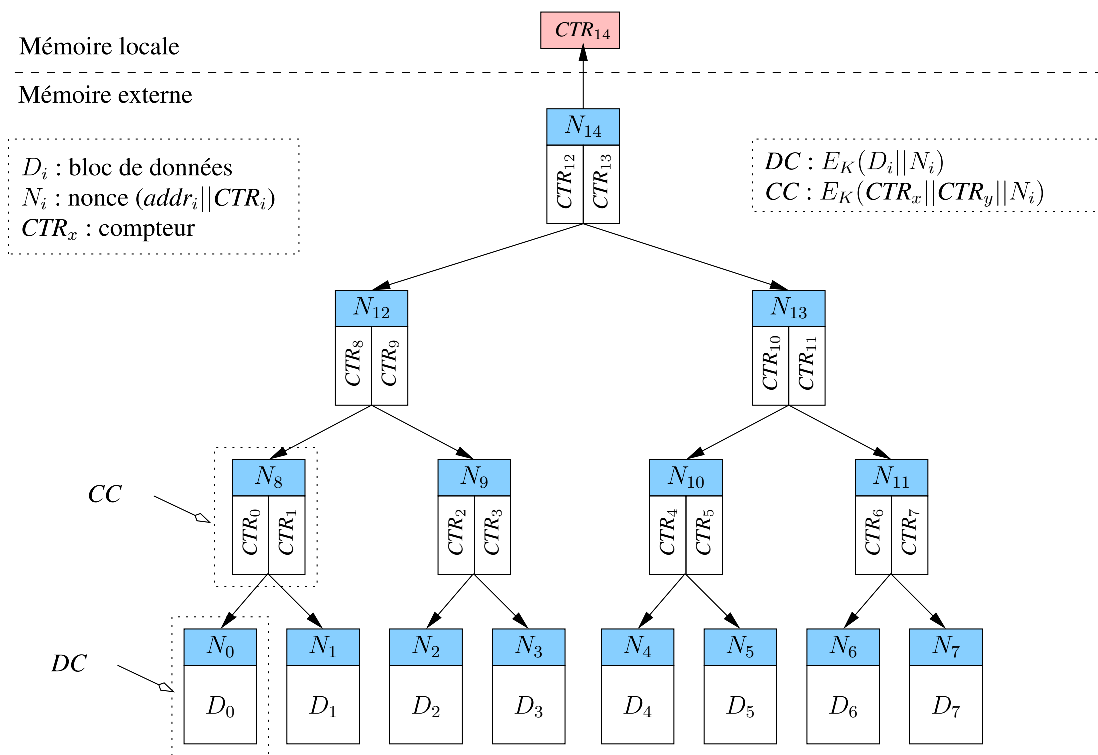


FIGURE 2.21 – *Tamper-Evident Counter Tree* (TEC-Tree)

ReadAndCheck le *DC* correspondant au bloc de données D et le *CC* parent sont récupérés à l'intérieur de la puce et déchiffrés. L'authentification du *DC* est effectuée en comparant le `nonce` reconstruit, à partir de l'adresse et du compteur correspondant dans le payload du *CC* parent déchiffré, avec les derniers n bits du texte en clair produit après le déchiffrement du *DC*. Le bloc de données D est envoyé au processeur (exécution spéculative) s'il n'y a pas d'erreur de comparaison. Ensuite, le reste des nœuds de la branche du *DC* jusqu'à la racine sont authentifiés comme décrit précédemment. L'authentification de la racine est effectuée en comparant le compteur produit après le déchiffrement de la racine avec le compteur stocké à l'intérieur de la puce. Si une erreur est détecté durant la vérification de la branche, une exception de vérification d'intégrité est levée pour prévenir le processeur. Toutes les opérations de déchiffrement sont effectuées en parallèle, puisqu'ils sont basées sur le chiffrement par bloc dans le mode ECB.

WriteAndUpdate le *DC* à mettre à jour est récupéré, déchiffré et authentifié en utilisant le *CC* parent. Ensuite, le payload du *DC* est mis à jour, concaténé avec l'adresse du bloc et le compteur correspondant incrémenté, chiffré et écrit dans la mémoire externe. Le payload du *CC* parent doit aussi être mis à jour avec la nouvelle valeur du compteur. Ce processus est effectué pour tous les parents jusqu'à la mise à jour de la valeur du compteur de la racine stocké à l'intérieur de la puce. Tous les nœuds sont authentifiés avant d'être mis à jour. Les calculs effectués dans ce processus de mise à jour, à savoir les opérations de déchiffrement et de re-chiffrement, sont également parallélisables.

L'utilisation d'un TEC-Tree augmente les besoins en espace mémoire du système d'un facteur $\frac{2}{(a-1)}$ (où a est l'arité de l'arbre) par rapport à la même application n'utilisant aucun mécanisme de protection.

2.3.4 Comparaison des différents arbres

Les trois arbres décrits précédemment offrent une protection de la mémoire externe contre les attaques actives (injection, permutation spatiale et rejeu) décrites dans notre modèle de menace. TEC-Tree offre, en plus, une protection de la confidentialité des données. Les arbres de Merkle réguliers permettent d'avoir des opérations de vérification parallélisables et, PAT et TEC-Tree permettent d'avoir des opérations de vérification et de mise à jour parallélisables. Par contre, PAT et TEC-Tree augmentent l'empreinte de la mémoire externe par rapport aux arbres de Merkle réguliers. En effet, l'empreinte mémoire externe supplémentaire introduite par les arbres de PAT est $\frac{3}{2\times(a-1)}$ et celle de TEC-Tree est $\frac{2}{(a-1)}$ où a est l'arité de l'arbre. Par exemple, pour un arbre quaternaire (c'est-à-dire $a = 4$), PAT et TEC-Tree augmentent l'empreinte de la mémoire externe par les facteurs de $50\,\%$ et $66\,\%$ respectivement et, alors que les arbres de Merkle ont une augmentation de $33\,\%$ de l'empreinte de la mémoire externe. Le tableau 2.1 résume les différentes propriétés des trois arbres.

TABLE 2.1 – Résumé des propriétés des arbres existants [24]

	Arbre de Merkle	**PAT**	**TEC-Tree**
Résistance aux attaques actives	Oui	Oui	Oui
Parallélisabilité	Vérification d'intégrité uniquement	Vérification d'intégrité et mise à jour	Vérification d'intégrité et mise à jour
Confidentialité	Non	Non	Oui
Empreinte mémoire supplémentaire	$1/(a-1)$	$3/2(a-1)$	$2/(a-1)$

2.4 Plates-formes de calcul sécurisées

Dans cette section, nous présenterons quelques plates-formes de calcul sécurisées qui sont utilisées pour protéger la confidentialité et/ou l'intégrité du contenu des mémoires externes dans les systèmes embarqués.

2.4.1 Best

La limitation de la zone fiable au SoC et la protection de la mémoire externe par un module localisé à l'intérieur de la puce est un concept introduit par Best [29–32], en 1979, avec un microprocesseur utilisant un mécanisme de chiffrement de bus : les données sont chiffrées avant d'être transmises vers la mémoire externe lors des opérations d'écriture et sont déchiffrées uniquement à l'intérieur de la puce lors des opérations de lecture, ce qui rend les données stockées dans la mémoire externes incompréhensibles (théoriquement) par un adversaire. Les algorithmes de chiffrement utilisent une clé secrète qui est également stockée à l'intérieur du SoC, accessible uniquement par le mécanisme de protection. L'objectif est donc de garantir la confidentialité du code et des données d'un programme stocké en mémoire contre un attaquant pouvant espionner le bus mémoire ou la mémoire externe elle-même.

Ce microprocesseur cryptographique garantit uniquement la confidentialité des données en utilisant des algorithmes de chiffrement, mais ne fournit pas de mécanismes pour garantir l'intégrité des données. De plus, un seul programme peut être exécuté par ce microprocesseur qui ne dispose d'aucun support pour un éventuel système d'exploitation.

Un mécanisme de destruction de la clé du processeur a été ajouté afin de rendre le programme inutilisable après un certain nombre d'exécutions ou au bout d'un certain temps. De plus, des instructions de destruction de la clé ont été insérées dans le jeu d'instruction du processeur pour rendre le processeur inutilisable si ces instructions ont été déclenchées par un attaquant en essayant de modifier le programme chiffré.

2.4.2 Dallas DS5002FP

La société *Dallas Semiconductor* a intégré des dispositifs de sécurité dans des microcontrôleurs de la famille DS5000 afin de permettre le chiffrement du code et des données d'un programme stocké en mémoire, et donc d'empêcher qu'un adversaire accède aux données ou aux instructions en clair du programme.

Par exemple, le DS5002FP [33] contient une unité de chiffrement pour le bus de données et une autre pour le bus d'adresse. L'algorithme propriétaire de chiffrement utilisé pour les deux unités dépend d'une clé secrète de 64 bits stockée dans un registre spécial interne alimenté par une petite batterie externe. Une entrée particulière permet de programmer le microcontrôleur. Lorsqu'elle est activée, un moniteur génère une nouvelle clé à partir d'un générateur de nombres aléatoires intégré au circuit, puis charge en clair le programme via le port série, le chiffre à la volée et le stocke en mémoire. Une fois la programmation terminée, le moniteur est désactivé et il devient alors impossible d'accéder au contenu en clair de la mémoire ou à la clé sans qu'une nouvelle clé soit générée (rendant ainsi les anciens programmes inutilisables).

Afin de rendre l'observation du bus plus complexe, les adresses sont elles-mêmes chiffrées (ce qui revient à faire une permutation des adresses) et des accès mémoires aléatoires inutiles sont insérés lorsque le microcontrôleur n'utilise pas le bus mémoire.

Le mécanisme de chiffrement associé au dispositif de sécurité logiciel de DS5002FP est présenté dans la figure 2.22.

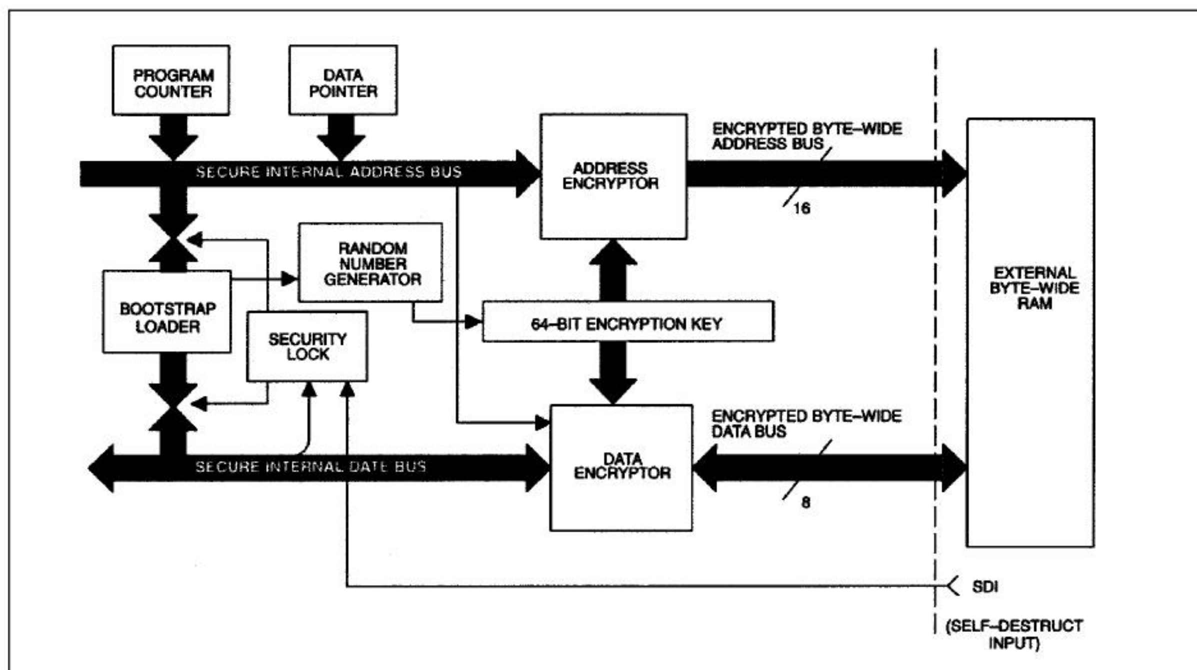


FIGURE 2.22 – Architecture de DS5002FP

Comme l'architecture Best, les dispositifs de sécurité proposés par la société *Dallas Semiconductor* protègent la confidentialité du programme et de ses données mais ne permettent pas

la détection de manipulation des données stockées dans la mémoire externe.

En 1998, Kuhn [2] a mené une attaque pratique contre DS5002FP. L'attaque se base sur le fait que l'intégrité de la mémoire n'est pas protégée et que l'algorithme de chiffrement des données (et des instructions) se base sur des blocs d'un octet. Pour une clé donnée et pour une adresse donnée (le chiffrement dépendant de l'adresse), la fonction de chiffrement est une permutation sur l'ensemble des 256 valeurs possibles de l'octet. L'attaque consiste tout d'abord à essayer toutes les valeurs possibles d'une suite de trois octets et à observer le comportement du microcontrôleur. À un moment donné, le clair correspondant à ces trois octets correspond à l'instruction d'écriture d'une valeur en clair sur le port parallèle du circuit. Il devient alors possible de tabuler complètement la fonction de déchiffrement pour une adresse donnée (le troisième octet contenant l'opérande). L'attaque se répète ensuite en essayant de décaler d'un mot les adresses (en trouvant l'équivalent d'une instruction ne faisant rien) afin de tabuler la fonction de déchiffrement pour un ensemble d'adresses consécutives. Il devient alors possible d'injecter en mémoire un petit programme lisant le contenu de toute la mémoire et le sortant en clair sur le port parallèle du microcontrôleur.

2.4.3 XOM

XOM (*eXecute-Only Memory*) [34–36] est une architecture sécurisée qui permet de garantir la confidentialité et l'intégrité du code et des données de programmes. Dans cette architecture, le système d'exploitation et la mémoire externe sont considérés comme non fiables. Le SoC est la seule zone fiable. La protection est effectuée uniquement par le processeur (à l'intérieur du SoC).

Le processeur est modifié en ajoutant de nouvelles extensions matérielles afin de protéger l'environnement d'exécution contre des intrusions logicielles malveillantes. En effet, les applications sont compartimentées : chaque programme sécurisé est associé à un identifiant XOM et une clé de session. Le but de cette organisation est d'interdire à une application d'accéder aux données des autres applications. Dans l'architecture XOM, un seul processus est exécuté à un instant donné et l'identifiant XOM associé à ce processus est appelé l'identifiant XOM actif. Avant chaque utilisation d'une donnée par un programme, le processeur vérifie que l'identifiant XOM de la donnée correspond bien à l'identifiant actif, sinon l'accès à celle-ci est interdit. Dans le cas où le programme produit une donnée, celle-ci est marquée par l'identifiant XOM actif.

L'utilisation du mode d'exécution sécurisé par un programme est permis à l'aide de deux instructions : `enter_xom` et `exit_xom`. L'instruction `enter_xom` permet à un programme de basculer en mode sécurisé en passant en argument l'adresse de la zone mémoire contenant la clé de session. Cette clé, qui est chiffrée à l'aide de la clé publique du processeur, est utilisée pour déchiffrer le programme et chiffrer ou déchiffrer ses données. L'instruction `exit_xom` est exécutée lorsqu'un programme veut sortir du mode chiffré.

Deux autres instructions sont introduites par XOM, `mv_to_null` et `mv_from_null`, qui permettent la communication entre les processus. L'instruction `mv_to_null` remplace l'identifiant XOM actif d'une donnée par l'identifiant nul, c'est-à-dire une donnée non chiffrée. L'instruction `mv_from_null` remplace l'identifiant nul d'une donnée par un identifiant XOM

actif, c'est-à-dire une donnée chiffrée.

La figure 2.23 présente l'architecture globale de XOM. Le module XVMM (*XOM Virtual Machine Monitor*), localisé à l'intérieur du SoC (zone fiable), est responsable d'implémenter les quatre instructions spéciales (`enter_xom`, `exit_xom`, `mv_to_null` et `mv_from_null`).

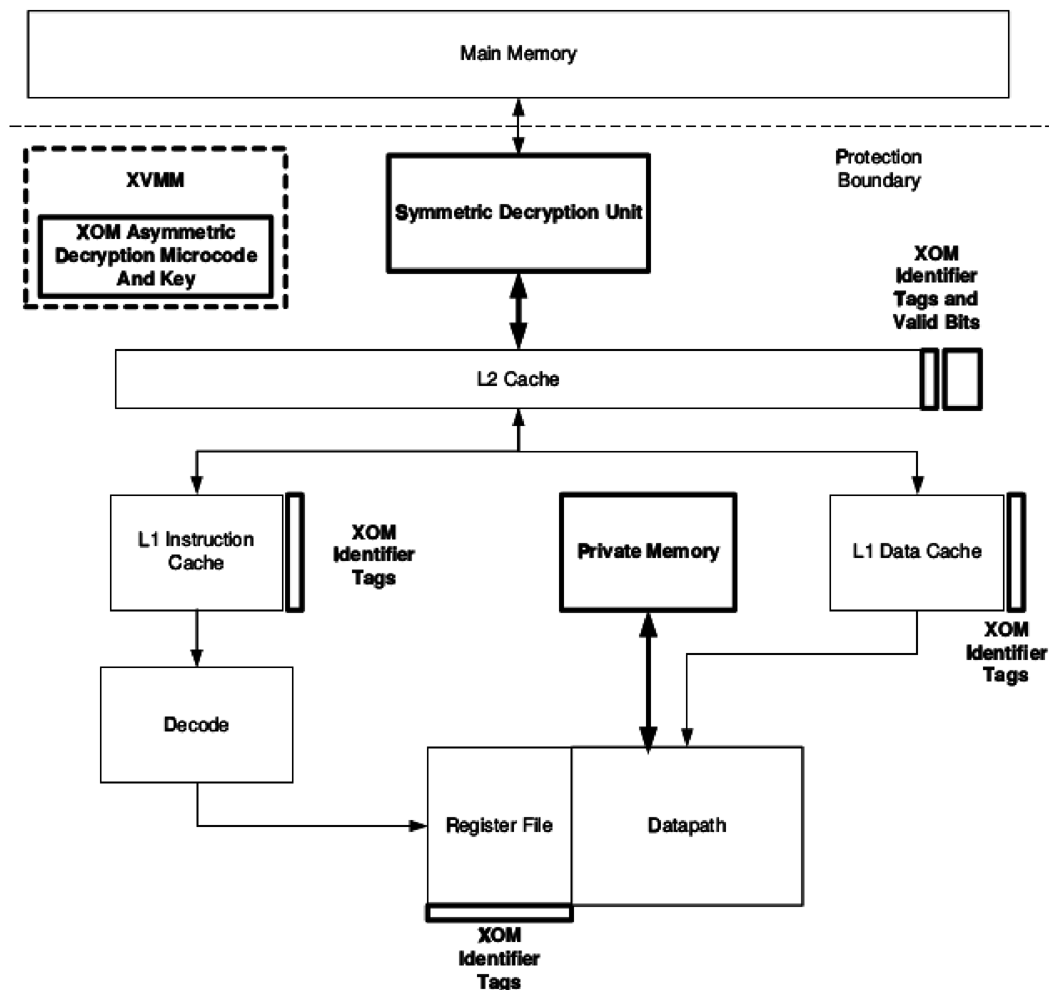


FIGURE 2.23 – Architecture de XOM

La confidentialité des données en mémoire externe est protégée pas un algorithme de chiffrement par bloc. Dans [37, 38], Yang *et al.* proposent une optimisation de l'architecture XOM en utilisant l'algorithme en mode compteur. Le masque produit par l'opération de chiffrement est combiné (xor) avec un bloc de données afin de générer le bloc chiffré. La graine utilisée pour calculer ce masque varie en fonction du type des données chiffrées. Pour les instructions, elle est égale à l'adresse virtuelle de la ligne correspondante à l'instruction. Dans ce cas, la graine est bien utilisée qu'une seule fois durant l'exécution d'un programme car les instructions ne sont pas modifiées. Quant aux données, la graine est égale à la concaténation de l'adresse virtuelle du bloc données et d'un compteur (incrémenté à chaque chiffrement). À chaque fois que le compteur déborde, la clé de chiffrement doit être changée, ce qui impose le re-chiffrement du programme. L'utilisation du mode compteur permet un gain en performance en parallélisant

le calcul des masques et la récupération des données chiffrées depuis la mémoire. D'après les résultats d'expérimentation effectuées par les auteurs, cette technique d'optimisation permet une réduction de la dégradation de performance de $20,79$ (architecture XOM de base) à $1,28\%$ (à l'aide de l'utilisation du mode compteur).

Pour ce qui concerne la protection d'intégrité, les auteurs utilisent des MAC en incluant l'adresse de la donnée, ce qui protège les données de la mémoire externe contre les attaques d'injection et les attaques par permutation spatiale, mais pas contre les attaques par rejeu.

2.4.4 AEGIS

Dans [39], Suh *et al.* présentent une plate-forme de calcul, baptisée AEGIS, sécurisée contre les attaques matérielles et logicielles. Ils considèrent que tout ce qui est localisé à l'extérieur du processeur (comme la mémoire externe) est non fiable. Deux variantes d'AEGIS sont proposées : une première dans laquelle une partie du système d'exploitation, en plus du processeur, est digne de confiance et une seconde pour laquelle le système d'exploitation (comme le reste des autres composants à l'extérieur du processeur) n'est pas digne de confiance et est exposé aux différentes attaques. Comme dans l'architecture XOM, AEGIS est basée sur des nouvelles instructions du processeur pour offrir un environnement d'exécution sécurisé. Dans AEGIS, deux environnements d'exécution sont ajoutés (en plus de l'environnement d'exécution normal où aucune protection n'est appliquée) :

— *Tamper-evident environment* (TE) dont l'objectif est de détecter toute tentative d'altération du code ou des données (intégrité uniquement).

— *Private and authenticated Tamper-Resistant environment* (PTR) dont l'objectif, en plus du TE, est de garantir la confidentialité des données (intégrité et confidentialité).

Dans l'architecture d'AEGIS, la confidentialité est initialement protégée par l'algorithme AES utilisé en mode CBC, puis, dans [40], Suh *et al.* proposent d'utiliser le mode compteur. Quant à l'intégrité, les auteurs utilisent un algorithme de MAC pour les données en lecture seule et, pour les données en écriture et lecture, ils utilisent les arbres de `hash`. Les attaques par rejeu sont donc détectées. Un autre mécanisme a été proposé par Clarke *et al.* [41] pour la protection de la mémoire en intégrité : les fonctions de hachage incrémentales sur des multiensembles (*Incremental multiset hash functions*). L'inconvénient est que dans ce cas l'opération de vérification n'est réalisée que de temps en temps (vérification *hors-ligne*).

Afin de réduire la dégradation des performances due aux arbres de `hash`, Gassend *et al.* [42] proposent de stocker les nœuds des arbres les plus souvent utilisés dans un cache au sein de la zone de confiance. Quand un nœud de l'arbre est stocké dans le cache, il est considéré comme digne de confiance et donc comme la racine du sous-arbre en dessous, ce qui réduit la latence et le nombre d'accès à la mémoire du processus de vérification.

Dans [40], Suh *et al.* ont évalué les performances dans les deux nouveaux environnements d'exécution. Les résultats de simulation montrent que la pénalité moyenne apportée par l'environnement TE est de 20% (50% dans le pire cas), causée principalement par la primitive des arbres de `hash`. Quand à l'environnement PTR, la pénalité moyenne apportée est de 40% (60% dans le pire cas).

2.4.5 CryptoPage

Le projet CryptoPage est similaire à XOM et à Aegis. Dans [43] Duc *et al.* considèrent que l'attaquant dispose d'un contrôle total sur tout ce qui est à l'extérieur du circuit intégré contenant le processeur, comme par exemple les bus de données et d'adresses du processeur, la mémoire, les périphériques de stockage de masse, les périphériques d'entrée/sortie, etc. Il peut ainsi, par exemple, surveiller les communications entre la mémoire et le processeur, modifier des valeurs en mémoire, etc. Au niveau logiciel, les auteurs supposent que l'attaquant peut également modifier le fonctionnement du système d'exploitation (en exploitant un bogue, ou en remplaçant le système d'exploitation par un autre volontairement erroné, s'il a accès à la machine) et de toutes les applications non sécurisées, et peut exécuter les applications sécurisées de son choix (y compris des applications sécurisées créées par lui-même) [44].

L'objectif de l'architecture CryptoPage est de permettre l'exécution de processus sécurisés [44]. Elle doit garantir à ces processus la propriété de confidentialité (un attaquant doit pouvoir obtenir le moins d'information possible sur le code ou les données manipulées par un processus sécurisé) et d'intégrité (l'exécution correcte d'un processus sécurisé ne doit pas pouvoir être altérée par une attaque, et en cas d'attaque, le processeur doit interrompre l'exécution du processus). Le processeur doit être capable d'exécuter en parallèle des processus sécurisés et des processus normaux. Le système d'exploitation n'a pas besoin d'être sécurisé, ni même d'être de confiance et peut être malicieux.

Les auteurs ont également présenté comment cette architecture peut déléguer une partie de certains mécanismes de sécurité au système d'exploitation, bien que celui-ci puisse être sous le contrôle d'un attaquant, sans néanmoins compromettre la sécurité. Cette délégation permet de réduire les modifications matérielles à apporter et permet plus de flexibilité.

L'architecture CryptoPage dispose de trois environnements d'exécution [44]. Le premier permet l'exécution de processus normaux, sans propriétés de sécurité particulières. Le deuxième permet de garantir la confidentialité et l'intégrité aux processus sécurisés qui s'exécutent dans cet environnement. Enfin, le troisième environnement garantit uniquement la propriété d'intégrité aux processus sécurisés. Des processus s'exécutant dans chacun de ces trois environnements d'exécution peuvent cohabiter simultanément sur l'architecture CryptoPage.

Afin d'obtenir l'objectif souhaité, cette architecture combine un mécanisme de chiffrement mémoire basé sur le mode compteur d'un algorithme de chiffrement par bloc (avec un mécanisme permettant de garantir que la graine utilisée pour le calcul des masques est toujours disponible permettant ainsi de toujours paralléliser le calcul des masques et l'accès mémoire pour récupérer les données chiffrée), un mécanisme de protection de l'intégrité mémoire basé sur une combinaison de MAC et d'arbres de Merkle et un mécanisme de protection contre les fuites d'informations sur le bus d'adresse [43].

Les simulations qui ont été effectuées montrent un surcout (en terme de nombre d'instructions exécutées par cycle d'horloge) de 3% à 8% des différents mécanismes de sécurité, par rapport à une architecture classique non sécurisée.

2.4.6 MESA

MESA (*Memory-Centric Security Architecture*) [45–48] est une architecture sécurisée qui protège la confidentialité et l'intégrité de tout ce qui est à l'extérieur du SoC en utilisant un système d'exploitation de confiance et d'autres fonctionnalités de l'architecture qui prennent en charge l'exécution des programmes sécurisés.

Un processus utilisateur fait souvent appel à des composants logiciels (bibliothèques, intergiciel, etc.) provenant de sources hétérogènes ayant des besoins et des caractéristiques de sécurité variés. Pour cette raison, les différents segments de la mémoire virtuelle sont associés avec différents niveaux de protection. Dans MESA, une capsule de mémoire est définie pour chaque segment de mémoire virtuelle contenant un ensemble d'attributs de sécurité connexes. Le système d'exploitation sécurisé gère à l'exécution toutes ces capsules de mémoire. À l'aide des paramètres de sécurité des capsules de mémoire, la protection de la confidentialité et l'intégrité des accès à la mémoire est exécutée sur des lignes de cache du processeur.

La protection de la confidentialité dans MESA utilise également la primitive OTP du chiffrement de flux. Pour accélérer la génération du masque, une technique de prédiction de la graine jetable est exploitée. Pour la vérification de l'intégrité MESA est basée sur les arbres de `hash`/MAC. Lorsqu'une ligne de cache (données ou instructions) est lue depuis la mémoire dans le processeur sécurisé, elle est déchiffrée et l'intégrité de tout l'espace de mémoire virtuel est vérifiée en utilisant un arbre de `hash` ou un arbre de MAC. Lorsqu'une ligne de cache est évincée, elle est chiffrée et l'arbre de `hash`/MAC est mis à jour.

En résumé, l'architecture de sécurité MESA effectue une protection diversifiée des différents composants logiciels. Mais cette protection raffinée et encore au niveau grain élevé, ainsi que l'utilisation des adresses virtuelles, nécessite la modification du processeur. La personnalisation du processeur et la nécessité de développer un système d'exploitation sécurisé font de MESA une solution plutôt complexe.

2.4.7 PE-ICE

Dans [22, 49, 50], Elbaz *et al.* proposent une architecture, baptisée PE-ICE (*Parallelized Encryption and Integrity Checking Engine*), qui permet la vérification d'intégrité et un chiffrement parallélisable afin de protéger le contenu de la mémoire. Dans leur contexte, seules les attaques matérielles (comme les attaques par injection ou *Probing Attacks*) sont prises en comptes. Les menaces de sécurité provenant des attaques logicielles ne sont pas considérées car la protection monolithique de l'ensemble de la mémoire ne nécessite pas l'intervention du système d'exploitation qui n'a donc pas besoin d'être de confiance.

Le processus de protection exploite le parallélisme de certains modes d'opération du chiffrement par bloc pour accélérer la protection de confidentialité et d'intégrité. La figure2.24 présente les opérations d'écriture et de lecture dans la mémoire externe [49]. Avant d'écrire une donnée *Pu* dans la mémoire, le mode ECB chiffre les données du message concaténées avec un `nonce` T (un nombre aléatoire, utilisé une seule fois). Finalement, le bloc chiffré C est stocké dans la mémoire et le `nonce` associé est stocké à l'intérieur du SoC (dans un cache

local). Pour un accès en lecture, le bloc chiffré C est déchiffré et, si le nonce T associé est égal à celui de référence T', stocké dans le cache local, l'intégrité est validée.

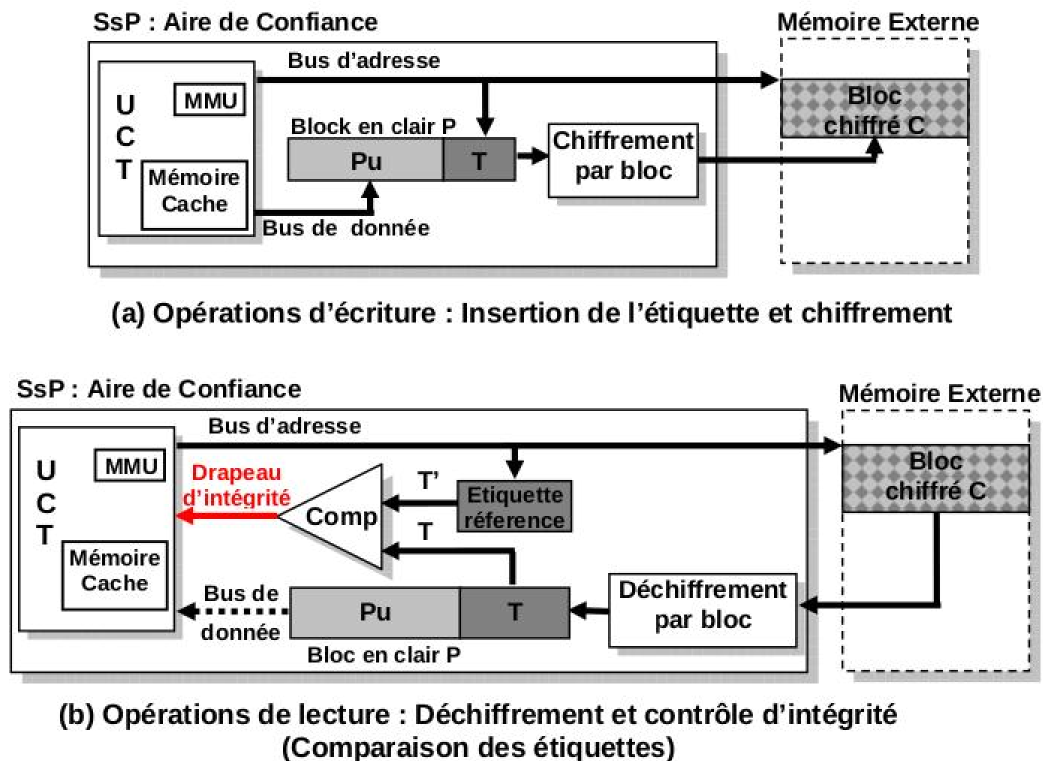


FIGURE 2.24 – Principe de fonctionnement de PE-ICE

La génération du nonce est différente pour chaque type de données traitées par le processeur, à savoir les données en lecture seule (*Read Only*, RO) et les données en lecture/écriture (*Read/Write*, RW). D'une part, les nonces générés pour les données RO sont fixes et, sont constitués des bits les plus significatifs de l'adresse de chaque donnée *Pu* stockée en mémoire. D'autre part, pour les données RW, les nonces sont composés d'une valeur aléatoire (changé à chaque écriture) concaténée avec les bits les plus significatifs de l'adresse.

Comme aucun module logiciel n'est impliqué dans la gestion de la protection, toutes les données de la mémoire sont protégées et tous les blocs de données sont concaténés avec des nonces, ce qui augmente la taille du cache de nonces. Pour cette raisons, dans [28], les auteurs proposent un nouveau mécanisme, baptisé TEC-Tree (voir page 57), basé sur la primitive d'arbre de Merkle où les nonces peuvent être stockés dans la mémoire externe, avec les blocs de données. La racine de l'arbre, bien sûr, est conservée à l'intérieur du SoC.

2.5 Conclusion

Dans ce chapitre nous avons présenté notre modèle de menace dans lequel un adversaire peut avoir accès aux données contenues dans la mémoire externe (problème de confidentialité)

et peut également les modifier (problème d'intégrité). Le composant de calcul principal (SoC) est supposé inattaquable, ce qui permet d'appliquer la protection du contenu de la mémoire externe depuis l'intérieur du SoC. Ensuite, nous avons présenté des méthodes de protection de la confidentialité et de détection de violations d'intégrité, ainsi que plusieurs plates-formes de calcul sécurisé dans le domaine des systèmes embarqués proposées dans la littérature.

Dans la section 2.3 nous avons étudié en détail la primitive des arbres de Merkle. Du point de vue de la sécurité, les arbres de Merkle offrent une protection contre toute attaque qui menace l'intégrité des données et particulièrement les attaques par rejeu. Par contre, ils introduisent une dégradation significative des performances du système. Comme le montre l'évaluation des performances, toutes les opérations des arbres de Merkle sont concernées par cette dégradation. En effet, pendant l'initialisation, la fonction utilisée pour initialiser tous les nœuds de l'arbre prend un temps significatif. Après l'initialisation, tout accès à la mémoire externe (lecture ou écriture) introduit des accès supplémentaires et des calculs de condensés cryptographiques.

Une solution courante pour atténuer cette dégradation est d'utiliser une mémoire cache [42] qui est située dans la zone sécurisée (proche du composant de calcul) et qui contient les nœuds intermédiaires les plus fréquemment et ou récemment utilisés, ce qui réduit le nombre d'accès à la mémoire externe et aussi le nombre de calculs de condensés cryptographiques. En effet, les nœuds stockés dans le cache sont des nœuds sûrs et le processus de vérification peut s'arrêter avec le premier nœud intermédiaire rencontré dans le cache, donc sûr ; il devient inutile de vérifier les nœuds parents jusqu'à la racine. Le processus de mise à jour peut, quant à lui, s'arrêter à la première écriture d'un nouveau nœud dans le cache, la mise à jour des nœuds parents, jusqu'à la racine, étant retardée. Cette seconde optimisation, plus complexe mais aussi plus prometteuse, accélère la mise à jour répétée d'une même portion d'arbre en la limitant aux premiers nœud rencontrés dans le cache.

Pour la phase d'initialisation d'arbre, nous avons décrit une opération optimisée qui permet de supprimer les lectures régulières des groupes de nœuds et donc accélérer cette phase. Une autre solution possible, que nous proposons dans ce manuscrit, concerne également l'accélération de la phase d'initialisation en utilisant une variante des arbres de Merkle : Arbres de Merkle Creux (AMC).

Le prochain chapitre présente de manière détaillée les AMC et leurs différences avec les AMR. Ensuite, nous étudierons l'impact d'utilisation de la mémoire cache sur la gestion de cohérence des arbres de Merkle réguliers et creux.

Chapitre 3

Les arbres de Merkle creux et la gestion de la mémoire

Sommaire

3.1 Les arbres creux initialisés 73
3.1.1 Opérations 73
3.1.2 Évaluation des performances 75
3.2 Les arbres creux non initialisés 76
3.2.1 Opérations 77
3.2.2 Évaluation des performances 79
3.3 Utilisation d'un cache d'arbres de Merkle 81
3.3.1 Stratégie ASAP 82
3.3.2 Stratégie ALAP 82
3.3.3 Cache d'arbre de Merkle ALAP et *Write-Through* 82
3.3.4 Cache d'arbre de Merkle ALAP et *Write-Back-Allocate* 83
3.3.5 Combinaison des arbres creux et du cache d'arbre 99
3.4 Conclusion 100

Comme nous l'avons vu dans le chapitre précédent, les arbres de Merkle peuvent avoir un impact négatif important sur les performances du système. Dans ce chapitre nous allons tout d'abord étudier une première optimisation, les Arbres de Merkle Creux (AMC), dans deux variantes différentes.

Les Arbres de Merkle Creux (AMC) sont des arbres de Merkle dans lesquels on introduit une valeur de nœud particulière, `NULL`, qui n'est pas un condensé cryptographique et qui ne peut pas être le résultat d'un calcul de condensé cryptographique. Un nœud de valeur `NULL` dans une branche signifie que tous les nœuds fils, ainsi que leurs descendants, jusqu'aux feuilles, n'ont pas encore été initialisés. La figure 3.1 représente un AMC avec une zone non-initialisée (en dessous du nœud $(i, a^{s-i} - 1)$ de valeur `NULL`).

L'initialisation d'un AMR est basée sur une fonction récursive de lecture, calcul du condensé et d'écriture pour générer tous les nœuds de l'arbre, à partir du premier niveau (au-dessus des données) jusqu'au nœud racine.

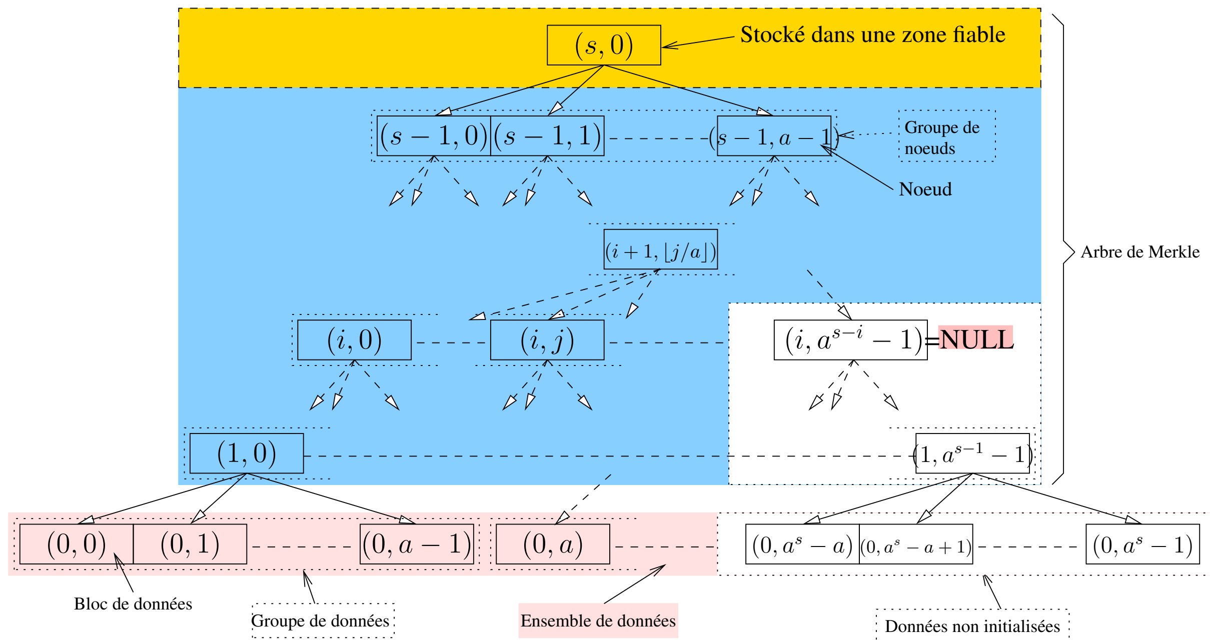


FIGURE 3.1 – AMC protégeant l'intégrité d'un ensemble de données

Avec l'AMC, l'initialisation s'effectue avec une fonction récursive d'écriture des valeurs `NULL` dans les nœuds de l'arbre, sans lecture de groupe ni calcul de condensé. Il y a deux possibilités pour initialiser l'arbre :

- AMC Initialisés (AMC-I) : tous les nœuds de l'arbre sont initialisés avec des valeurs `NULL`, sauf les feuilles (les données). Nous considérons que l'initialisation des blocs de données avec des `NULL` n'est pas nécessaire. En effet, pour écrire (ou lire) une donnée dans la mémoire au démarrage d'un programme, nous n'avons pas besoin de vérifier la valeur précédente du bloc de données, qui n'est pas utile.
- AMC Non-Initialisés (AMC-NI) : le nœud racine est initialisé avec une valeur `NULL` et les nœuds de l'arbre restent avec des valeurs inconnues, dont les feuilles. L'initialisation de ces nœuds se fait progressivement pendant les opérations de mise à jour ultérieures.

L'introduction des nouvelles valeurs `NULL` impose la modification des différentes opérations de gestion de l'arbre : initialisation, vérification d'intégrité ou mise à jour d'un nœud.

Nous présenterons ensuite une seconde optimisation basée sur l'utilisation d'un cache d'arbres de Merkle. L'utilisation d'un cache d'arbres de Merkle promet des gains significatifs lors des vérifications d'intégrité car celles-ci peuvent s'interrompre dès qu'un nœud de la branche parcourue est trouvé dans le cache. Symétriquement, la mise à jour d'une branche d'arbre peut s'arrêter au premier nœud écrit dans le cache. Cependant, comme nous le verrons, les choses ne sont pas aussi simples qu'avec un cache classique, car les nœuds d'un arbre de Merkle sont dépendants les uns des autres. La question de la cohérence ne se limite donc pas à la synchronisation entre le cache et la mémoire externe. Elle porte également sur la cohérence interne des arbres, dont les nœuds peuvent ponctuellement, lors des opérations de mise à jour, ne plus être en cohérence avec leurs descendants ou leurs parents.

Enfin, dans un troisième temps, nous expliquerons comment les arbres creux et le cache

Algorithme 5 `Init-Null` : Initialisation de l'arbre avec des nœuds `NULL`

```
1: for i ∈ {1, ..., s} do                          ▷ Pour les niveaux d'arbre sauf les données
2:     for j ∈ {0, ..., a^(s−i)} do                 ▷ Pour les nœuds d'un niveau
3:         Write(i, j, NULL)                        ▷ Initialiser le nœud avec NULL
4:     end for
5: end for
```

d'arbres de Merkle peuvent être utilisés conjointement.

3.1 Les arbres creux initialisés

Dans un AMC-I, à l'étape d'initialisation, tous les nœuds de l'arbre sont initialisés avec des valeurs `NULL` y compris le nœud racine. Au fur et à mesure des écritures dans la mémoire protégée, l'arbre se remplit avec des condensés à la place des nœuds `NULL` jusqu'au moment où l'arbre ne contient plus aucun nœud `NULL`. La figure 3.2 montre ces trois états de l'AMC-I.

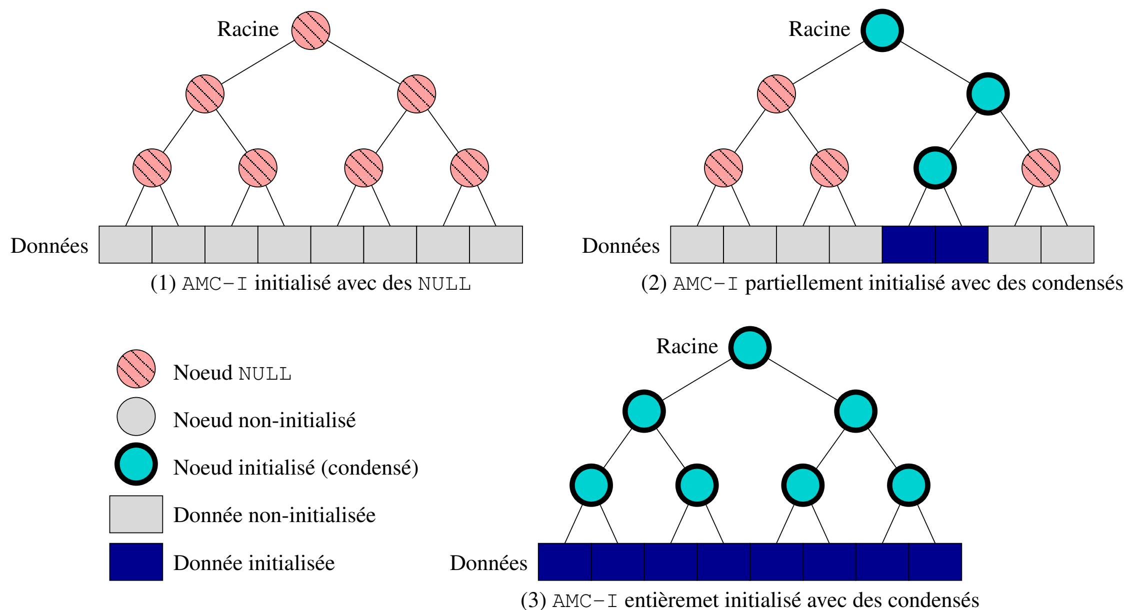


FIGURE 3.2 – Différents états d'un AMC-I

3.1.1 Opérations

3.1.1.1 Initialisation

Cette étape repose sur une fonction itérative qui initialise tous les nœuds de l'arbre avec la valeur `NULL`, du premier niveau jusqu'au niveau de la racine, à l'exception des feuilles. Ce processus est basé sur des écritures non vérifiées, sans calculs de condensés cryptographiques et sans lectures des nœuds. L'algorithme 5 illustre cette étape.

Algorithme 6 `Lecture-Vérifiée`$(i \neq s, j = a \times j_0 + j_1)$

```
1: (u, v_0, v_1) ← (i, j_0, j_1)                          ▷ Indices temporaires
2: for k ∈ {0, ..., a − 1} do                             ▷ Pour a nœuds frères
3:     X_k ← Read(u, a × v_0 + k)                         ▷ Lecture non vérifiée
4: end for
5: R ← X_{v_1}                                            ▷ La valeur à retourner
6: D ← Digest(u + 1, v_0, X_0, ..., X_{a−1})              ▷ Calcul du nœud père
7: (u, v_0, v_1) ← (u + 1, ⌊v_0/a⌋, v_0 mod a)            ▷ Avancer au niveau supérieur
8: while u < s do                                         ▷ Pour les niveaux d'arbre
9:     for k ∈ {0, ..., a − 1} do                         ▷ Pour a nœuds frères
10:        X_k ← Read(u, a × v_0 + k)                     ▷ Lecture non vérifiée
11:    end for
12:    if X_{v_1} ≢ D then                                ▷ Si la comparaison échoue
13:        error                                          ▷ Abandonner
14:    end if
15:    D ← Digest(u + 1, v_0, X_0, ..., X_{a−1})          ▷ Calcul du nœud père
16:    (u, v_0, v_1) ← (u + 1, ⌊v_0/a⌋, v_0 mod a)        ▷ Avancer au niveau supérieur
17: end while
18: X_0 ← Read(s, 0)                                      ▷ Lire à partir de la zone sécurisée
19: if X_0 ≢ D then                                       ▷ Si la comparaison échoue
20:     error                                             ▷ Abandonner
21: end if
22: return R                                              ▷ Retourner le nœud requis et arrêter
```

3.1.1.2 Vérification

Le processus de vérification permet de vérifier l'intégrité de la valeur d'un nœud. Dans le cas des AMC-I, ce processus est un peu différent de celui des AMR. En effet, la comparaison entre la valeur de référence R d'un nœud père et un condensé calculé D, notée $D \equiv R$ (opposé $D \not\equiv R$) dans la suite, tient compte des nœuds `NULL` en ceci qu'elle réussit toujours si l'un des deux nœuds est `NULL`, comme résumé par l'équation 3.1.

$$D \equiv R \Leftrightarrow (D = \texttt{NULL}) \text{ or } (R = \texttt{NULL}) \text{ or } (D = R) \tag{3.1}$$

Rappelons qu'un calcul de condensé ne peut pas avoir `NULL` pour résultat et que D, dans cet exemple, ne peut donc jamais être `NULL`. La comparaison réussit donc uniquement si $(R = \texttt{NULL}) \text{ or } (R = D)$ et la définition de $D \equiv R$ pourrait tout aussi bien être prise asymétrique (équation 3.2) sans modifier le résultat.

$$D \equiv R \Leftrightarrow (R = \texttt{NULL}) \text{ or } (D = R) \tag{3.2}$$

Dans la suite on retiendra la version symétrique, plus naturelle.

L'algorithme 6 présente la fonction de lecture vérifiée du nœud $(i \neq s, j = a \times j_0 + j_1)$. C'est le même algorithme que celui des AMR à la seule différence de la comparaison utilisée.

3.1.1.3 Mise à jour

L'algorithme 7 présente une écriture vérifiée de la valeur V dans le nœud $(i \neq s, j = a \times j_0 + j_1)$. Là encore, la seule différence avec l'algorithme des AMR est la comparaison utilisée.

Algorithme 7 `Écriture-Vérifiée`$((i \neq s, j = a \times j_0 + j_1), V)$

1: $(u, v_0, v_1) \leftarrow (i, j_0, j_1)$ ▷ Indices temporaires
2: **for** $k \in \{0, ..., a-1\}$ **do** ▷ Pour a nœuds frères
3: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Lecture non vérifiée
4: **end for**
5: $D_{old} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père ancien
6: $X_{v_1} \leftarrow V$ ▷ La nouvelle valeur du nœud
7: $\text{Write}((u, a \times v_0 + v_1), X_{v_1})$ ▷ Écriture non vérifiée de la nouvelle valeur du nœud
8: $D_{new} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père nouveau
9: $(u, v_0, v_1) \leftarrow (u+1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ Avancer au niveau supérieur
10: **while** $u < s$ **do** ▷ Pour les niveaux d'arbre
11: **for** $k \in \{0, ..., a-1\}$ **do** ▷ Pour a nœuds frères
12: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Lecture non vérifiée
13: **end for**
14: **if** $X_{v_1} \not\equiv D_{old}$ **then** ▷ Si la comparaison échoue
15: **error** ▷ Abandonner
16: **end if**
17: $D_{old} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père ancien
18: $X_{v_1} \leftarrow D_{new}$ ▷ La nouvelle valeur du nœud
19: $\text{Write}((u, a \times v_0 + v_1), X_{v_1})$ ▷ Écriture non vérifiée de la nouvelle valeur du nœud
20: $D_{new} \leftarrow \text{Digest}(u+1, v_0, X_0, ..., X_{a-1})$ ▷ Calcul du nœud père nouveau
21: $(u, v_0, v_1) \leftarrow (u+1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ Avancer au niveau supérieur
22: **end while**
23: $X_0 \leftarrow \text{Read}(s, 0)$ ▷ Lire à partir de la zone sécurisée
24: **if** $X_0 \not\equiv D_{old}$ **then** ▷ Si la comparaison échoue
25: **error** ▷ Abandonner
26: **end if**
27: $\text{Write}((s, 0), D_{new})$ ▷ Écrire la nouvelle racine dans la zone sécurisée

3.1.2 Évaluation des performances

3.1.2.1 Initialisation

L'initialisation d'un AMC-I, avec l'algorithme 5, demande une écriture régulière des nœuds (lignes 2 à 4) du premier niveau de l'arbre jusqu'à la racine.

Le temps pris par une initialisation du premier niveau ($i = 1$), est donc :

$$ti_1 = t_{wn} \times a^{s-1} \tag{3.3}$$

tandis que

$$ti_{s-1} = t_{wn} \times a \tag{3.4}$$

et, par hypothèse,

$$ti_s = 0 \tag{3.5}$$

Au total, pour l'initialisation de tous les niveaux,

$$ti = t_{wn} \times \sum_{i=1}^{s-1} a^i \tag{3.6}$$

Avec les différentes tailles des pages utilisées dans notre exemple à base d'ARM, nous obtenons (en nombre de cycles d'horloge du processeur), les valeurs suivantes :

— 4 Kio : $t_i \approx 676$ cycles
— 64 Kio : $t_i \approx 0,011 \cdot 10^6$ cycles

— 1 Mio : $t_i \approx 0,524 \cdot 10^6$ cycles
— 16 Mio : $t_i \approx 2,796 \cdot 10^6$ cycles

Ces résultats montrent que l'opération d'initialisation des AMC-I est 65 fois plus rapide que celle des AMR et 10 fois plus rapide que l'opération optimisée d'initialisation des AMR.

3.1.2.2 Lecture vérifiée

Lire et vérifier un nœud $(i \neq s, j = a \times j_0 + j_1)$, différent du nœud racine, avec l'algorithme 6 demande, pour chaque itération, un calcul de MAC (lignes 6 ou 15) et une lecture régulière du groupe (lignes 2 à 4 ou 9 à 11). Le temps de calcul du MAC est négligeable puisque ce calcul peut être effectué en parallèle avec la prochaine lecture régulière (à $i + 1$) du groupe (lignes 9 à 11), et que le temps nécessaire pour ce calcul est plus petit que la latence mémoire. Par contre, le dernier calcul du MAC, qui correspond à la dernière lecture instantanée du nœud racine, n'est pas négligeable.

Le temps pris par une lecture vérifiée au niveau $i = 0$ est le même que celui des AMR, présenté par l'équation (2.3.2.3) (voir page 55).

L'algorithme de lecture vérifiée dans le cas d'un AMC-I a les même performances que dans le cas d'un AMR. Cela est dû du fait qu'un nœud de valeur `NULL` est considéré dans le traitement de l'arbre comme un nœud normal qui contient un condensé.

3.1.2.3 Écriture vérifiée

Une écriture vérifiée d'un nœud $(i \neq, j = a \times j_0 + j_1)$, différent du nœud racine, avec l'algorithme 7 demande, par itération, une lecture régulière du groupe de nœuds (lignes 2 à 4 ou 11 à 13) et deux calculs de MAC (lignes 5 et 8 ou 17 et 20) qui s'effectuent en parallèle avec la prochaine lecture régulière du groupe (lignes 11 à 13) puisque la latence mémoire est deux fois plus grande que le temps de calcul du MAC ($l_0 > 2 \times t_{MAC}$). Par contre, le dernier calcul du MAC, qui correspond à la dernière lecture instantanée du nœud racine (ligne 23), n'est pas négligeable.

Le temps pris par une écriture vérifiée au niveau $i = 0$ est le même que celui des AMR, présenté par l'équation (2.3.2.4) (voir page 56).

Comme pour l'algorithme de lecture vérifiée, celui d'écriture vérifiée dans le cas des AMC-I n'introduit aucune pénalité par rapport à l'algorithme pour les AMR.

3.2 Les arbres creux non initialisés

Avec la méthode des AMC-NI, à l'étape d'initialisation, seul le nœud racine est initialisé à `NULL` et le reste des nœuds de l'arbre contiennent des valeurs aléatoires. La figure 3.3 représente l'état de l'arbre à l'étape d'initialisation, puis, quand il y a des accès à la mémoire où

l'arbre devient partiellement initialisé avec des condensés cryptographiques et ensuite, quand l'arbre devient entièrement initialisé (plus aucun nœud NULL dans l'arbre).

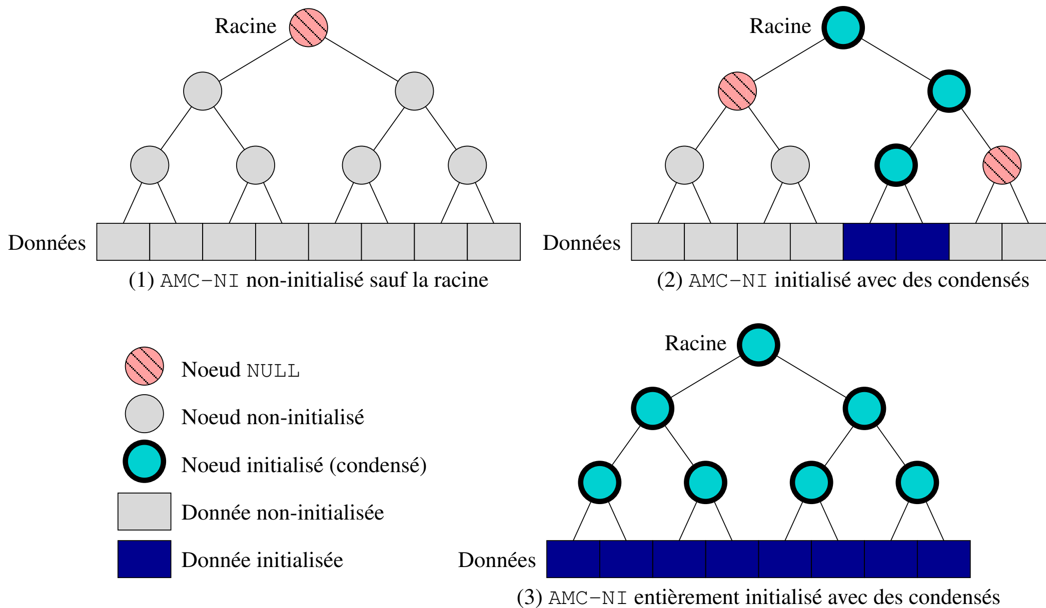


FIGURE 3.3 – Différents états d'un AMC-NI

3.2.1 Opérations

3.2.1.1 Initialisation

L'initialisation de l'arbre dans cette méthode repose simplement sur l'initialisation du nœud racine avec la valeur NULL.

3.2.1.2 Vérification

Le traitement de la vérification d'intégrité avec l'AMC-NI est différent de celui de l'AMR. La détection d'une erreur lors d'une comparaison entre la valeur d'un nœud père en mémoire et celle recalculée, ne doit déclencher une interruption qu'au dernier niveau de l'arbre, si aucun nœud de valeur NULL n'est rencontré sur le chemin vers la racine. En effet, sur une branche non initialisée (sous un nœud de valeur NULL), les valeurs des différents nœuds sont aléatoires. Un drapeau noté NOK est activée à chaque fois qu'une comparaison échoue (sauf pour la dernière comparaison de la racine) et reste activé tant qu'aucun nœud de valeur NULL n'est rencontré en montant vers la racine. Tant que le drapeau NOK est activé, le calcul de condensé est inutile. Si un nœud de valeur NULL est rencontré (c'est-à-dire que nous sommes dans une branche non initialisée), le drapeau NOK est désactivé et la vérification d'intégrité commence à partir de ce niveau, en calculant le condensé à chaque niveau et en le comparant avec l'ancienne valeur du nœud père.

Algorithme 8 `Lecture-Vérifiée`$(i \neq s, j = a \times j_0 + j_1)$

```
1: (u, v0, v1) ← (i, j0, j1)                                   ▷ Indices temporaires
2: NOK ← false                                                 ▷ Initialiser le drapeau
3: while u ≤ s do                                              ▷ Pour les niveaux d'arbres
4:     if u = s then                                           ▷ Si le nœud racine
5:         X0 ← Read(s, 0)                                     ▷ Lire le nœud racine dans la zone sécurisée
6:         if X0 ≠ NULL and (NOK or X0 ≠ D) then               ▷ Si la comparaison échoue
7:             error                                           ▷ Abandonner
8:         end if
9:         return R                                           ▷ Retourner le nœud requis et arrêter
10:    end if
11:    for k ∈ {0, ..., a − 1} do                              ▷ Pour les a nœuds frères
12:        Xk ← Read(u, a × v0 + k)                            ▷ Lecture non vérifiée
13:    end for
14:    if u = i then                                           ▷ Si le niveau du départ, pas de vérification
15:        R ← X_{v1}                                          ▷ La valeur à retourner
16:    else if X_{v1} = NULL then                              ▷ Si le nœud est NULL
17:        NOK ← false                                         ▷ Effacer le drapeau
18:    else if X_{v1} ≠ D then                                 ▷ Si la comparaison échoue
19:        NOK ← true                                          ▷ Assigner le drapeau
20:    end if
21:    if ¬NOK then                                            ▷ Si NOK est désactivé
22:        D ← Digest(X0, ..., X_{a−1})                        ▷ Calcul du nœud père
23:    end if
24:    (u, v0, v1) ← (u + 1, ⌊v0/a⌋, v0 mod a)                 ▷ Avancer au niveau supérieur
25: end while
```

L'algorithme 8 présente une lecture vérifiée du nœud $(i \neq s, j = a \times j_0 + j_1)$ et fonctionne de la même manière que l'algorithme 2 (voir page 49) des AMR, à la différence que la détection d'une erreur d'intégrité ne déclenche l'arrêt du système qu'au dernier niveau de la racine (dans la zone fiable) si le drapeau NOK est activé (ligne 6). Dans un niveau plus bas, si une erreur est détectée (ligne 18), nous activons juste le drapeau NOK qui reste activé tant que nous n'avons pas rencontré un nœud de valeur `NULL` dans notre chemin en montant dans la branche d'arbre vers le nœud racine.

3.2.1.3 Mise à jour

Le processus de mise à jour avec l'AMC-NI est également différent de celui de l'AMR. La détection d'erreur ne peut déclencher une interruption qu'au dernier niveau de l'arbre, si aucun nœud de valeur `NULL` n'est rencontré dans notre chemin vers la racine. Le drapeau NOK est activé à chaque fois qu'une comparaison échoue (sauf pour la dernière comparaison de la racine) et reste activé tant qu'aucun nœud de valeur `NULL` n'est rencontré en montant vers la racine. Si un nœud de valeur `NULL` est rencontré (c'est-à-dire que nous sommes dans une branche non encore initialisée), le drapeau NOK est désactivé et nous retournons au nœud du départ pour mettre à jour la sous-branche non initialisée. Les nœuds pères de cette sous-branche sont mis à jour avec des nouveaux condensés et leurs nœuds frères avec des valeurs `NULL`.

L'algorithme 9 présente une écriture vérifiée de la valeur V dans le nœud $(i \neq s, j = a \times j_0, j_1)$. Comme l'algorithme de lecture vérifiée 8, cet algorithme ne déclenche l'arrêt du système qu'au dernier niveau de la racine si le drapeau NOK est activé. Si la branche concernée ne contient pas de nœud de valeur `NULL`, cet algorithme fonctionne de la même manière que l'algorithme 3 (voir page 50) des AMR. Sinon, lorsque l'on rencontre un nœud `NULL` (ligne 17) sur notre chemin, nous retournons au nœud du départ au niveau i et initialisons, avec

Algorithme 9 Écriture-Vérifiée$((i, j = a \times j_0 + j_1), V)$

```
1: (u, v_0, v_1) ← (i, j_0, j_1)                                ▷ Indices temporaires
2: NOK ← false                                                 ▷ Initialiser le drapeau
3: while u ≤ s do                                              ▷ Pour les niveaux d'arbre
4:     if u = s then                                           ▷ Si le nœud racine
5:         X_0 ← Read(s, 0)                                    ▷ Lire le nœud racine dans la zone sécurisée
6:         if X_0 ≠ NULL and (NOK or X_0 ≠ D_old) then         ▷ Si la comparaison échoue
7:             error                                           ▷ Abandonner
8:         end if
9:         Write((s, 0), D_new)                                ▷ Écrire la nouvelle valeur du nœud racine dans la zone sécurisée
10:        exit                                                ▷ Et arrêter
11:    end if
12:    for k ∈ {0, ..., a − 1} do                              ▷ Pour les a nœuds frères
13:        X_k ← Read(u, a × v_0 + k)                          ▷ Lecture non vérifiée
14:    end for
15:    if u = i then                                           ▷ Si le niveau du départ, pas de vérification
16:        D_new ← V                                           ▷ La valeur à écrire
17:    else if X_{v_1} = NULL then                             ▷ Si le nœud est NULL, recommencer dès le début
18:        (m, n_0, n_1) ← (i, j_0, j_1)                       ▷ Indices temporaires
19:        NOK ← false                                         ▷ Initialiser le drapeau
20:        D_new ← V                                           ▷ La nouvelle valeur du nœud
21:        while m < u do                                      ▷ Pour les niveaux d'arbre en dessous de u
22:            Y_{n_1} ← D_new                                 ▷ La nouvelle valeur du nœud
23:            Write((m, a × n_0 + n_1), Y_{n_1})              ▷ Écriture non vérifiée
24:            for k ∈ {0, ..., a − 1} \ {n_1} do              ▷ Pour les a − 1 nœuds frères
25:                Y_k ← NULL                                  ▷ Assigner le nœud à NULL
26:                Write((m, k), NULL)                         ▷ Écriture non vérifiée de NULL
27:            end for
28:            D_new ← Digest(Y_0, ..., Y_{a−1})               ▷ Calcul du nœud père nouveau
29:            (m, n_0, n_1) ← (m + 1, ⌊n_0/a⌋, n_0 mod a)     ▷ Avancer au niveau supérieur
30:        end while
31:    else if X_{v_1} ≠ D_old then                            ▷ Si la comparaison échoue
32:        NOK ← true                                          ▷ Assigner le drapeau
33:    end if
34:    D_old ← Digest(X_0, ..., X_{a−1})                       ▷ Calcul de l'ancienne valeur du nœud père
35:    X_{v_1} ← D_new                                         ▷ La nouvelle valeur du nœud
36:    Write((u, a × v_0 + v_1), X_{v_1})                      ▷ Écriture non vérifiée de la nouvelle valeur du nœud
37:    D_new ← Digest(X_0, ..., X_{a−1})                       ▷ Calcul de la nouvelle valeur du nœud père
38:    (u, v_0, v_1) ← (u + 1, ⌊v_0/a⌋, v_0 mod a)             ▷ Avancer au niveau supérieur
39: end while
```

des condensés après calcul (sauf le bloc de données avec sa valeur demandée), tous les nœuds $(i, j), (i+1, \frac{j}{a}), ..., (i+(n-1), \frac{j}{(n-1)\times a})$ où n est le niveau du nœud de valeur NULL (ligne 21), et remplaçons les valeurs de leurs nœuds frères par des nœuds de valeur NULL (ligne 26).

3.2.2 Évaluation des performances

3.2.2.1 Initialisation

L'étape d'initialisation des AMC-NI consiste à initialiser la racine avec une valeur NULL dans la zone sécurisée. Le temps pris par cette opération est négligeable puisqu'elle s'effectue à l'intérieur de la puce.

3.2.2.2 Lecture vérifiée

Lire et vérifier un nœud $(i \neq s, j = a \times j_0 + j_1)$, différent du nœud racine, avec l'algorithme 8 demande un calcul de MAC (ligne 22) et une lecture régulière du groupe (lignes 11 à 13). Le temps de calcul du MAC est négligeable puisque ce calcul se fait en parallèle avec la prochaine lecture régulière (à $i + 1$) du groupe (lignes 11 à 13), et le temps nécessaire pour ce calcul est plus petit que la latence mémoire. Par contre, le dernier calcul du MAC, qui correspond à la dernière lecture instantanée du nœud racine, n'est pas négligeable.

Le temps pris par une lecture vérifiée au niveau $i = 0$ est le même que celui des AMR, présenté par l'équation (2.3.2.3) (voir page 55).

Les performances de l'algorithme de lecture vérifiée dans le cas des AMC-NI sont les mêmes que celles de l'algorithme pour les AMR.

3.2.2.3 Écriture vérifiée

Une écriture vérifiée d'un nœud $(i \neq s, j = a \times j_0 + j_1)$, différent de la racine, avec l'algorithme 9, demande, si la branche concernée ne contient aucun nœud `NULL`, deux calculs de MAC (lignes 34 et 37) qui s'effectuent en parallèle avec la prochaine lecture régulière du groupe (lignes 12 à 14) puisque la latence mémoire est deux fois plus grande que le temps de calcul du MAC ($l_0 > 2 \times t_{MAC}$). Par contre, le dernier calcul du MAC, qui correspond à la dernière lecture instantanée de la racine (ligne 5), n'est pas négligeable.

Le temps pris par une écriture vérifiée au niveau $i = 0$ est le même que celui des AMR, présenté par l'équation (2.3.2.4) (voir page 56).

Dans le cas où le nœud (i, j) se trouve dans une branche n'est pas encore initialisée, la partie entre le niveau i et le niveau du nœud de valeur `NULL` est mise à jour deux fois. La première est une écriture vérifiée régulière et la deuxième est une écriture non vérifiée (lignes 17 à 31). Cette dernière, demande un calcul de MAC (lignes 28) et une écriture régulière d'un groupe de nœuds (lignes 23 à 27) qui s'effectue en parallèle avec les calculs des MAC.

Le temps pris par l'écriture vérifiée au niveau $i < s - 1$, noté tw_i, est donc :

$$tw_i = tw_{i \to s} + tw_{i \to n} \tag{3.7}$$

où n est le niveau du nœud de valeur `NULL`, tandis que

$$tw_{s-1} = t_{rg} + t_{wn} + (2 + k) \times t_{MAC} + tw_s \tag{3.8}$$

et, par hypothèse

$$tw_s = 0 \tag{3.9}$$

où $k = 1$ si la valeur de la racine est `NULL`, sinon $k = 0$.

Au total,

$$tw_0 = s \times (t_{rg} + t_{wn}) + (2 + n - i) \times t_{MAC} \tag{3.10}$$

et l'équation suivante montre la latence supplémentaire introduite par une écriture vérifiée de l'AMC-NI par rapport à une écriture régulière en l'absence de contrôle d'intégrité (c'est-à-dire t_{wn}) :

$$(s-1) \times (t_{rg} + t_{wn}) + t_{rg} + (2 + n - i) \times t_{MAC} \tag{3.11}$$

Avec les différentes tailles des pages utilisées dans notre exemple à base d'ARM, pour une lecture d'une ligne de cache (en nombre de cycles d'horloge du processeur), nous allons présenter deux facteurs pour chaque page, un facteur minimal où la branche ne contient aucun nœud `NULL` et un facteur maximal qui correspond à la première écriture dans l'arbre où la valeur du nœud racine est `NULL` :

- 4 Kio : latence supplémentaire minimale de 588 et maximale de 688 cycles d'horloge
- 64 Kio : latence supplémentaire minimale de 808 et maximale de 948 cycles d'horloge
- 1 Mio : latence supplémentaire minimale de 1028 et maximale de 1208 cycles d'horloge
- 16 Mio : latence supplémentaire minimale de 1248 et maximale de 1468 cycles d'horloge

3.3 Utilisation d'un cache d'arbres de Merkle

Dans la section précédente, nous avons présenté deux variantes de l'AMR permettant d'accélérer la phase d'initialisation, au prix d'une certaine complexité supplémentaire pendant les opérations de vérification d'intégrité et de mise à jour d'un nœud de l'arbre. Cette section explore les optimisations pour atténuer l'impact des arbres de Merkle sur les performances lors des accès à la mémoire externe (zone non sécurisée).

La vérification d'intégrité et la mise à jour sont des opérations itératives qui parcourent une branche d'arbre depuis une feuille jusqu'à ce qu'un nœud de référence sûr soit rencontré. Les arbres de Merkle ont souvent une arité faible car plus l'arité est grande, plus le nombre de nœuds à accéder pour calculer les condensés cryptographiques est important. En conséquence, les arbres de Merkle sont généralement de grandes structures de données, de taille comparable à l'ensemble des données qu'ils protègent. Par exemple, dans un arbre 4-aire où les nœuds ont la taille des blocs de données protégés, la taille totale de l'ensemble des nœuds de l'arbre est $1/3$ de l'ensemble de données. Pour cette raison, l'arbre complet est généralement stocké dans la mémoire externe, à l'exception de la racine qui doit être stockée dans une zone sécurisée pour des raisons évidentes de sécurité. Les accès supplémentaires induits par les opérations de vérification et de mise à jour augmentent significativement l'utilisation du bus mémoire et peuvent constituer un goulot d'étranglement.

L'utilisation d'un cache dédié pour stocker les nœuds d'arbre les plus fréquemment accédés est une manière très naturelle d'améliorer les performances du système. Le cache diminue la bande passante avec la mémoire et, lorsqu'il se trouve dans la zone sécurisée (à l'intérieur de la puce), il permet également de réduire le nombre de calculs de condensés.

3.3.1 Stratégie ASAP

En effet, avec l'utilisation du cache, la racine n'est plus le seul nœud fiable : tout nœud stocké dans le cache est un nœud sûr et peut être utilisé comme une référence fiable qu'un attaquant ne peut pas altérer. La vérification de l'intégrité peut donc être arrêtée le plus tôt possible, avec le premier nœud intermédiaire trouvé dans le cache et il n'y a pas besoin de vérifier le reste des nœuds de la branche jusqu'à la racine pour retourner la donnée demandée. Par la suite, nous nous référerons à cette optimisation par la stratégie de vérification le-plus-tôt-possible (ASAP, *As-Soon-As-Possible*).

La stratégie ASAP est relativement simple à appliquer, sans effet indésirable, la vérification de l'arbre s'arrête tout simplement avec le premier cache `hit`. Elle est compatible avec toutes les architectures de cache, associatifs ou non, et quelles que soient leurs politiques d'écriture (*Write-Through* ou *Write-Back*), d'écriture en cas de `miss` (*Write-Allocate* ou *No-Write-Allocate*) et de remplacement (*Least Recently Used*, *FIFO*...).

Nous ne la détaillerons donc pas plus avant, sa mise en œuvre est évidente.

3.3.2 Stratégie ALAP

De même que la vérification peut être prématurément interrompue, la mise à jour d'une branche d'arbre peut également être stoppée avec l'écriture du premier nœud de la branche dans le cache et il est possible de repousser à plus tard la mise à jour des nœuds parents. Bien entendu, les nœuds parents, que ce soit dans le cache ou en mémoire externe, ne sont plus en cohérence avec leurs descendants et ils devront être mis à jour ultérieurement. On dira dans la suite qu'ils sont *obsolètes*. Cette remise à plus tard est particulièrement utile lorsque le même nœud de l'arbre est modifié à plusieurs reprises : conserver toute la branche à jour (à partir du nœud modifié jusqu'au nœud racine) serait un gaspillage de calculs de condensés et entraînerait des accès inutiles à la mémoire non fiable. Dans la suite, cette optimisation est nommée stratégie de mise à jour le-plus-tard-possible (ALAP, *As-Late-As-Possible*). Remarquons qu'elle est indépendante de la stratégie ASAP et qu'une gestion de cache d'arbre de Merkle peut utiliser l'une, l'autre, les deux ou aucune.

L'utilisation des caches ALAP semble tout aussi prometteuse que la stratégie ASAP pour limiter la dégradation des performances due à l'utilisation des arbres de Merkle. Cependant, elle remet en cause la cohérence interne des arbres de Merkle qui peuvent désormais contenir des nœuds obsolètes. Comme nous allons le voir dans les sections suivantes, le type de cache utilisé n'est plus indifférent et la correction fonctionnelle d'un cache ALAP ne va pas de soi.

3.3.3 Cache d'arbre de Merkle ALAP et *Write-Through*

La politique d'écriture *Write-Through* consiste, lors de la mise à jour d'un nœud d'arbre, à écrire le nœud dans la mémoire externe. Pour décrire complètement le comportement du cache il convient de préciser aussi quelle est sa politique d'écriture en cas de `miss` :

— Écriture dans le cache même en cas de `miss` (politique *Write-Allocate*).

— Écriture dans le cache uniquement en cas de `hit` (politique *No-Write-Allocate*).
En cas de `hit` avec la politique *No-Write-Allocate* ainsi que dans tous les cas avec la politique *Write-Allocate*, lors d'une écriture, le nœud est écrit dans le cache. La stratégie ALAP impose alors l'arrêt de la mise à jour et les nœuds parents deviennent donc obsolètes, tant dans le cache que dans la mémoire externe.

La figure 3.4 présente un exemple de mise à jour d'une branche dans un arbre binaire de quatre niveaux en utilisant la politique d'écriture *Write-Through*. Après la mise à jour du bloc de données $D(0,4)$, nous écrivons le nouveau nœud père $N(1,2)$ (après calcul du condensé sur le nouveau groupe de données) dans la mémoire cache et dans la mémoire externe. Ce processus s'arrête à ce niveau.

L'ancienne valeur du nœud $N(1,2)$ est définitivement perdue car elle a été écrasée par la nouvelle dans le cache comme dans la mémoire externe. Si le nœud père (obsolète) $N(2,1)$ ne se trouve pas dans le cache, ce qui est tout à fait possible, lors du prochain parcours de cette branche il sera impossible de procéder à sa lecture vérifiée car celle-ci fait intervenir l'ancienne valeur perdue de $N(1,2)$. La première comparaison entre le condensé calculé, qui inclut le nœud mis à jour $N(1,2)$, avec le nœud père obsolète $N(2,1)$ va donc déclencher une erreur.

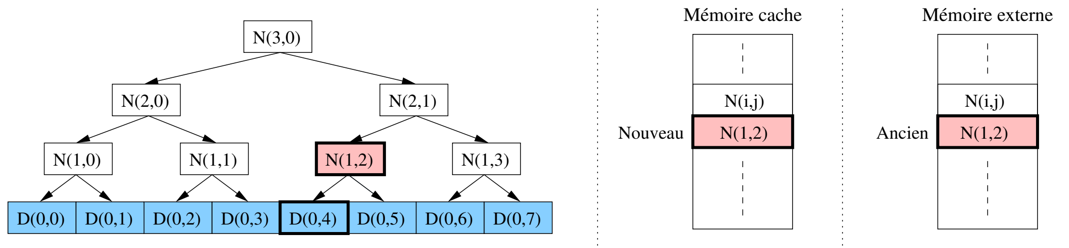


FIGURE 3.4 – Politique d'écriture Write-Through

Il n'est donc pas possible d'utiliser la stratégie ALAP avec un cache d'arbre de Merkle *Write-Through*. La même remarque s'applique à un cache *Write-Back-No-Allocate*, c'est à dire à un cache qui n'écrit qu'en mémoire externe lors d'une écriture avec cache `miss`. En effet, lors d'une écriture avec cache `miss` le même scénario que celui décrit pour un cache *Write-Through* pourrait se produire et l'arbre deviendrait alors définitivement incohérent.

3.3.4 Cache d'arbre de Merkle ALAP et *Write-Back-Allocate*

La politique *Write-Back* consiste, lors de la mise à jour d'un nœud, à l'écrire dans le cache et à retarder son écriture dans la mémoire externe. L'utilisation de la stratégie ALAP a alors pour conséquence que la branche reste cohérente dans la mémoire externe mais n'est plus à jour : tous les nœuds à partir du nœud mis à jour dans le cache et jusqu'à la racine sont obsolètes. Dans le cache, en revanche, le nœud modifié est à jour mais ses parents, s'ils sont également dans le cache, sont obsolètes, ce qui mène à une incohérence interne de l'arbre.

La figure 3.5 présente un exemple de mise à jour d'une branche dans un arbre binaire de quatre niveaux en utilisant la politique d'écriture *Write-Back*. Après la mise à jour du bloc de

données $D(0,4)$, nous écrivons le nouveau nœud père $N(1,2)$ (après calcul du condensé sur le nouveau groupe de données) dans la mémoire cache. Ce processus s'arrête à ce niveau. Les nœuds de la branche qui sont stockés dans la mémoire externe sont cohérents mais ne sont plus à jour. Quant aux nœuds supérieurs $N(2,1)$ et $N(3,0)$, s'ils sont stockés dans la mémoire cache, ils deviennent obsolètes.

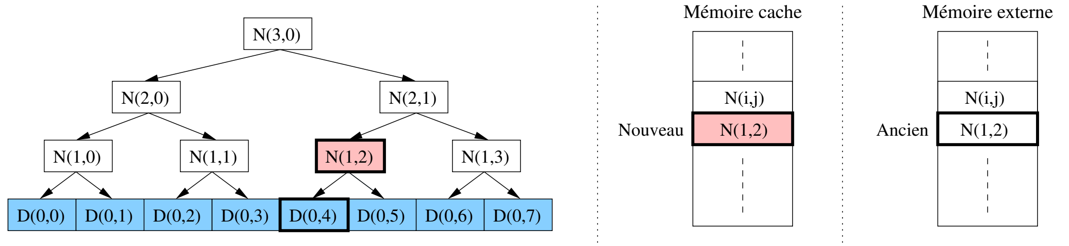


FIGURE 3.5 – Politique d'écriture Write-Back

Comme l'ancienne valeur du nœud modifié est toujours disponible en mémoire externe, le problème rencontré précédemment avec un cache *Write-Through* ne se pose plus avec un cache *Write-Back-Allocate*. Nous allons donc étudier la faisabilité d'un tel cache.

La discussion sur la faisabilité porte uniquement sur les aspects fonctionnels. Les performances seront abordées au chapitre 5. Le système sans cache d'arbre de Merkle que nous avons étudié jusqu'à présent, et sa version modifiée par l'ajout du cache sont représentées sur la figure 3.6.

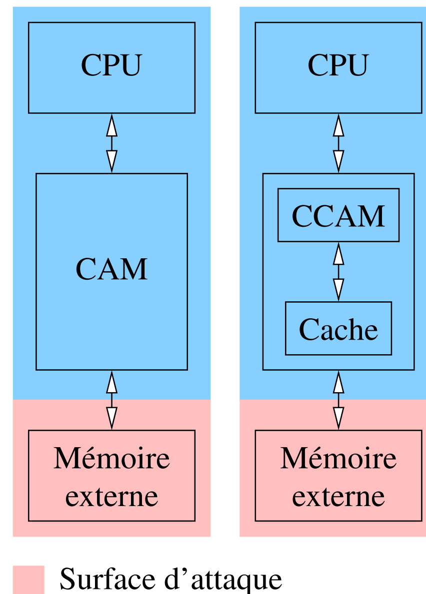


FIGURE 3.6 – Système étudié avec ou sans cache d'arbre de Merkle

Dans la version sans cache on distingue la mémoire externe, seule attaquable, l'émetteur de requêtes d'accès à la mémoire (CPU) et le Contrôleur d'Arbres de Merkle (CAM). Dans la version modifiée, le CAM est remplacé par un sous-système composé du cache proprement dit

et du Contrôleur de Cache d'Arbre de Merkle (CCAM). La gestion du cache est assurée par le CCAM qui pilote le cache à l'aide d'opérations élémentaires que nous définirons. En retour, le cache communique des informations sur son état au CCAM afin de lui permettre l'arrêt au plus tôt des vérifications ou des mises à jour et d'effectuer des opérations de maintenance (évictions...).

3.3.4.1 Les opérations élémentaires du cache

La stratégie ALAP nous conduit à nous intéresser à des arbres de Merkle où certains nœuds ont deux versions différentes, une version `dirty` stockée dans le cache et une autre version stockée en mémoire externe. Dans la suite nous les nommerons *Arbres de Merkle Cachés* (AM-Cached). Pour tout nœud $N(i,j)$ en cache on notera $N^c(i,j)$ la version en cache et $N^m(i,j)$ la version en mémoire externe. Si le nœud est `clean` on a donc $N^c(i,j) = N^m(i,j)$ alors que s'il est `dirty` on a $N^c(i,j) \neq N^m(i,j)$. Par extension, pour les nœuds qui ne sont pas en cache on considérera que $N^c(i,j) = N^m(i,j)$.

La stratégie ALAP stoppe la mise à jour d'une branche dès qu'un nœud de la branche est écrit dans le cache et devient donc `dirty`. Un nœud `dirty` a donc toujours un père obsolète, calculé à partir de l'ancienne valeur de ses fils `dirty`, celle qui se trouve en mémoire externe. Cette remarque est à la base du fonctionnement détaillé de la stratégie ALAP : lors de la vérification d'un groupe de nœuds, c'est toujours la version en mémoire externe des nœuds `dirty` qui est utilisée. Le nœud père, lui, est lu depuis le cache s'il s'y trouve, qu'il soit `dirty` ou non.

En plus des opérations de cache classiques de lecture et d'écriture, nous ajoutons donc une nouvelle opération de lecture modifiée (`RESTORE`) destinée à lire la version en mémoire externe des nœuds fils `dirty`.

La capacité du cache n'étant pas infinie il est inévitablement nécessaire de procéder à des évictions et, lorsqu'elles ciblent un nœud `dirty`, de synchroniser celui-ci avec la mémoire externe en l'y écrivant. Nous introduisons donc également l'opération dédiée à cette synchronisation (`SYNCHRONIZE`).

La politique de remplacement du cache, quelle qu'elle soit, est utilisée lorsque c'est nécessaire pour sélectionner un emplacement dans le cache susceptible d'accueillir un nœud. Selon le degré d'associativité du cache le nombre d'emplacements candidats peut être plus ou moins grand, depuis un seul pour les caches directs jusqu'au nombre total d'emplacements pour les caches totalement associatifs. Lorsque un ou plusieurs emplacements candidats sont libres, l'un d'entre eux est sélectionné. Lorsque aucun emplacement candidat n'est libre mais que un ou plusieurs sont occupés par un nœud `clean`, l'un d'entre eux est sélectionné par la politique de remplacement du cache. Si aucun emplacement candidat n'est libre ni occupé par un nœud `clean`, l'opération échoue et renvoie la coordonnée (i,j) d'un nœud victime `dirty` occupant un emplacement candidat. L'auteur de la requête doit alors utiliser cette information pour procéder à une éviction avant de soumettre à nouveau sa requête.

Au total, le cache ALAP *Write-Back-Allocate* implémente les 4 opérations de base suivantes :

— `READ` (adresse) : lecture classique d'un nœud dans le cache s'il s'y trouve (cache `hit`), dans la mémoire externe sinon (cache `miss`). En cas de `miss`, le nœud lu est stocké dans le cache si c'est possible. Sinon il n'est pas stocké en cache. `READ` renvoie $(V, \texttt{status})$, où V est la valeur lue et `status` un indicateur valant `hit-clean`, `hit-dirty` ou `miss`.
— `WRITE` (adresse, nœud) : écriture classique d'un nœud dans le cache si c'est possible. Le nœud n'est pas écrit en mémoire externe (politique *Write-Back*) et devient `dirty`. Les nœuds parents deviennent obsolètes, qu'ils soient en cache ou non. `WRITE` renvoie $(\texttt{status}, (i, j))$ où `status` est un indicateur valant `ok` ou `fail` et (i, j) la coordonnée d'un nœud victime à évincer lorsque l'opération échoue (signalé par $\texttt{status} = \texttt{fail}$).
— `RESTORE` (adresse) : lecture modifiée d'un nœud. Le comportement dépend de l'état du cache :
  — Le nœud est en cache et `dirty` : lecture dans la mémoire externe, le nœud lu n'est pas stocké dans le cache car il entrerait en conflit avec la version `dirty` du même nœud.
  — Dans tous les autres cas (`clean` ou `miss`), l'opération se comporte comme `READ` (adresse).

  `RESTORE` renvoie uniquement la valeur lue V.
— `SYNCHRONIZE` (adresse) : si le nœud est en cache et `dirty`, écriture en mémoire externe, le nœud devient `clean`. Sinon l'opération n'a aucun effet. `SYNCHRONIZE` ne renvoie rien.

Précisons que ces opérations sont atomiques, c'est à dire qu'elles ne peuvent être interrompues et qu'aucune autre interaction avec le cache ou la mémoire externe ne peut avoir lieu pendant leur déroulement.

Notons que la spécification de ces 4 opérations comporte déjà des choix d'implémentation non directement contraints par la stratégie ALAP ou la politique *Write-Back-Allocate*. En effet, `READ` et `RESTORE` pourraient, tout comme `WRITE`, échouer en cas de `miss` lorsque le nœud lu en mémoire externe ne peut pas être stocké dans le cache faute d'emplacement disponible. Nous verrons ultérieurement pourquoi ceci n'est pas nécessaire et comment nous contrôlons la saturation du cache en nœuds `dirty` de façon à disposer toujours d'emplacements disponibles lors des `READ` et des `RESTORE`.

Par extension et pour simplifier les algorithmes présentés nous considérerons que les opérations `READ` et `RESTORE` s'appliquent également aux données protégées par l'AMCached. Comme les données ne sont jamais stockées dans le cache d'AMCached, `READ` et `RESTORE` de données sont équivalentes à la lecture classique non vérifiée en mémoire externe.

C'est sur ces 4 opérations que repose la gestion du cache.

3.3.4.2 Cohérence interne partielle

Dans les figures d'AMCacheds qui suivent, les nœuds `dirty` sont dédoublés afin de représenter les versions en cache et en mémoire externe. Les extrémités des arêtes représentées précisent quelle version est le parent de quelle autre.

Avant de décrire la gestion du cache nous allons tout d'abord définir une propriété particulière sur les AMCacheds : la Cohérence Interne Partielle (CIP).

Définition 1. *Un AMCached est dit **CIP** si et seulement si, pour chacun de ses nœuds $N(i+1, j)$ on a :*

$$N^c(i+1, j) = \text{Digest}(N^m(i, a \times j), N^m(i, a \times j + 1), ..., N^m(i, a \times j + a - 1))$$

La figure 3.7 représente un AMCached CIP. On notera que la CIP ne fait aucune différence entre les nœuds qui ne sont pas en cache et ceux qui sont en cache mais `clean`. Dans les deux cas ils doivent être le condensé des versions en mémoire externe de leurs fils. Dans le cas des nœuds `dirty` la propriété précise que c'est la version en cache - et non pas la version en mémoire externe - qui doit être le condensé des versions en mémoire externe de leurs fils.

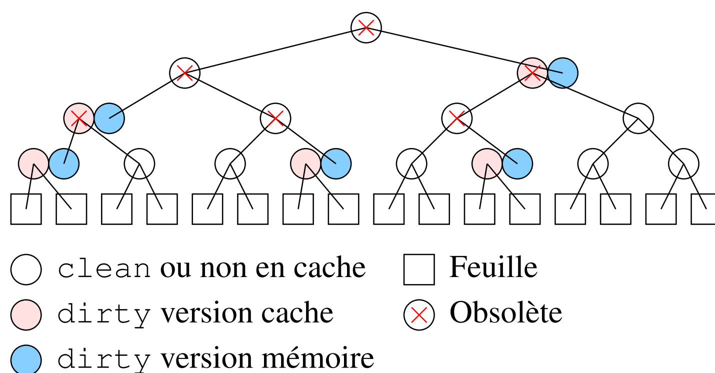


FIGURE 3.7 – AMCached CIP

La CIP va nous servir à démontrer la correction fonctionnelle de la gestion du cache, c'est à dire l'absence de fausses alertes de défaut d'intégrité et la détection systématique des attaques. Pour alléger les raisonnements nous supposons que la fonction de calcul de condensés utilisée est idéale, c'est à dire qu'elle n'est sensible ni aux attaques en collision ni aux attaques en premier ou second antécédent. Bien sûr, de telles fonctions n'existent pas, et l'utilisation d'une fonction réelle se traduira par le remplacement de la notion d'impossibilité par une notion plus réaliste de difficulté. Toujours dans un but de simplicité de présentation, et sans perte de généralité, nous supposerons que les attaques, si elles se produisent, ont lieu à l'instant même où le système lit une donnée en mémoire externe. Bien entendu, la donnée lue peut en réalité avoir été modifiée n'importe quand avant cette lecture, l'effet est le même.

3.3.4.3 Gestion du cache

La gestion du cache est assurée par le Contrôleur de Cache des Arbres de Merkle (CCAM) qui pilote le cache à l'aide des 4 opérations élémentaires déjà mentionnées. En retour, le cache informe le CCAM des `hits` afin de lui permettre l'arrêt au plus tôt des vérifications et des échecs (`fail`) pour lui signifier la nécessité d'une éviction. Lorsque que `WRITE` échoue, le CCAM met alors l'opération en attente et procède à l'éviction du nœud sélectionné. Il relance l'opération interrompue dès que l'éviction est terminée. La gestion du cache comporte donc

4 volets : initialisation de l'AMCached, lecture vérifiée, écriture vérifiée et éviction vérifiées. Toutes quatre sont atomiques, c'est à dire qu'elles ne peuvent être interrompues et qu'aucune autre interaction avec le cache ou la mémoire externe ne peut avoir lieu pendant leur déroulement.

Étudions tout d'abord l'initialisation du AMCached. L'algorithme 4 optimisé d'initialisation d'AMR ne peut tirer aucun bénéfice de la présence d'un cache d'arbre car il ne procède à aucune lecture en mémoire externe, la stratégie ASAP n'apporte donc rien, et que tous les calculs de condensés doivent absolument être menés à bien jusqu'à la racine, faute de quoi l'AMCached serait incomplet. La stratégie ALAP n'apporte donc rien non plus. La gestion du cache que nous proposons n'active donc le cache d'arbre qu'après l'initialisation. Dans son état initial tous ses emplacements sont libres et la racine conservée en interne est cohérente.

Lemme 1. *A l'issue de l'initialisation l'AMCached est CIP.*

Démonstration. Rappelons que, sans perte de généralité, on considère que les attaques ont lieu lors des lectures en mémoire externe. L'initialisation ne fait aucune lecture en mémoire externe. Tous les nœuds $N_m(i, j)$ sont donc non modifiés par l'attaquant éventuel. Pour tous les nœuds $N(i, j)$ de l'arbre on a $N_c(i, j) = N_m(i, j)$ car le cache est vide. □

L'algorithme d'éviction vérifiée d'un nœud $(i \notin \{0, s\}, j = a \times j_0 + j_1)$, dans sa version la plus naturelle (algorithme 10), va nous permettre de mettre en évidence une difficulté liée à la saturation du cache en nœuds `dirty` et de proposer le moyen d'y remédier. Le processus d'éviction provoqué par `WRITE` lorsqu'aucun emplacement n'est disponible dans le cache consiste à mettre à jour le nœud père du nœud victime sélectionné et à synchroniser ses nœuds fils avec la mémoire externe. Le cache sélectionne toujours un nœud victime `dirty` : évincer un nœud `clean` n'apporterait rien car si le cache a signalé l'échec de l'opération c'est qu'il n'existe aucun emplacement libre ou occupé par un nœud `clean` susceptible d'accueillir le nœud à stocker dans le cache. De plus, il sélectionne un nœud `dirty` occupant un emplacement possible pour le nœud à écrire. La synchronisation du nœud sélectionné va donc nécessairement libérer un emplacement adéquat. Si certains de ses frères sont également `dirty`, leur synchronisation va aussi libérer des emplacements (mais ils ne seront pas nécessairement utilisables pour l'écriture considérée).

La difficulté que nous devons tout d'abord résoudre provient du `WRITE` de la ligne 18. En effet, des emplacements (au moins un, celui du nœud (i, j) de départ) ont été libérés par les opérations de synchronisation (`SYNCHRONIZE`, ligne 6) mais rien ne garantit que ces emplacements sont adéquats pour accueillir le nœud père. Si le cache n'est pas totalement associatif, il est possible que le `WRITE` du père échoue et nécessite une nouvelle éviction. Nous avons en apparence résolu ce problème à l'aide d'une description récursive (ligne 20) mais ce n'est qu'une apparence car chaque appel à l'éviction laisse en attente un nouveau `WRITE`. De proche en proche, dans un cas très défavorable, on pourrait ainsi accumuler un grand nombre de `WRITE` en attente. Il serait sans doute possible de remédier au problème en stockant les `WRITE` interrompus dans une mémoire tampon mais la complexité de gestion d'un tel mécanisme, associée à la difficulté de prouver la préservation de la CIP pendant la série d'évictions, nous pousse à explorer une autre voie : le contrôle dynamique de saturation du cache.

Algorithme 10 Éviction-vérifiée($i \notin \{0, s\}, j = a \times j_0 + j_1$)

```
1: (u, v0, v1) ← (i, j0, j1)                                   ▷ Indices temporaires
2: for k ∈ {0, ..., a − 1} do                                  ▷ Pour a nœuds frères
3:     (Xk, status) ← READ(u, a × v0 + k)                      ▷ Lire avec READ
4:     if status = hit-dirty then                              ▷ Si cache hit et dirty
5:         Yk ← RESTORE(u, a × v0 + k)                         ▷ Lire avec RESTORE la version en mémoire externe
6:         SYNCHRONIZE(u, a × v0 + k)                          ▷ Synchronisation avec SYNCHRONIZE (écriture de Xk en mémoire externe)
7:     else                                                    ▷ Nœud non dirty
8:         Yk ← Xk                                             ▷ La version en mémoire externe est égale à Xk
9:     end if
10: end for
11: Dold ← Digest(Y0, ..., Ya−1)                               ▷ Calcul de l'ancien nœud père du nœud victime
12: Dnew ← Digest(X0, ..., Xa−1)                               ▷ Calcul du nouveau nœud père du nœud victime
13: (u, v0, v1) ← (u + 1, ⌊v0/a⌋, v0 mod a)                    ▷ Avancer au niveau supérieur
14: D = Lecture-vérifiée(u, a × v0 + v1)                       ▷ Lecture vérifiée du nœud père
15: if Dold ≠ D then                                           ▷ Si la comparaison échoue
16:     error                                                  ▷ Abandonner
17: else
18:     (status, (x, y)) ← WRITE((u, a × v0 + v1), Dnew)       ▷ Écrire le nouveau nœud père du nœud victime avec WRITE
19:     while status = fail do                                 ▷ Si WRITE échoue faute d'emplacement disponible
20:         Éviction-vérifiée(x, y)                            ▷ Relancer une éviction
21:         (status, (x, y)) ← WRITE((u, a × v0 + v1), Dnew)   ▷ Relancer le WRITE qui a échoué
22:     end while
23: end if
```

Quelle que soit l'associativité du cache, dans chaque ensemble $\mathcal{S}_s$ du cache [1] nous allons garder le compte C_s du nombre d'emplacements occupés par un nœud `dirty` et le comparer à un seuil τ. Les C_s sont modifiés uniquement par `WRITE` (qui augmente d'une unité le C_s de l'ensemble $\mathcal{S}_s$ destination) et par `SYNCHRONIZE` (qui diminue d'une unité le C_s l'ensemble $\mathcal{S}_s$ destination). A chaque `WRITE`, si $C_s = \tau$ pour l'ensemble $\mathcal{S}_s$ destination, on dira que $\mathcal{S}_s$ est *saturé* et une alerte sera communiquée au CCAM, ainsi que la coordonnée d'un nœud à évincer pour faire redescendre le compteur C_s en dessous de τ. La valeur maximum de τ est égale à la taille T_{set} des ensembles du cache (1 pour un cache direct, 4 pour un cache associatif à 4 voies, nombre total d'emplacements pour un cache totalement associatif) et sa valeur minimale est 1. Même si $\tau = T_{set}$, il va nous permettre de garantir qu'il existe toujours au moins un emplacement disponible (libre ou occupé par un nœud `clean`) dans chacun des ensembles du cache. Pour prendre en compte le seuil introduit nous redéfinissons les opérations du cache comme suit :

— `READ` (adresse) : lecture classique d'un nœud dans le cache s'il s'y trouve (cache `hit`), dans la mémoire externe sinon (cache `miss`). En cas de `miss`, le nœud lu est stocké dans le cache (le seuil garantit qu'il existe au moins un emplacement disponible). L'emplacement devient `clean`. Le compteur de l'ensemble concerné ne change pas. `READ` renvoie $(V, \texttt{status})$, où V est la valeur lue et `status` un indicateur valant `hit-clean`, `hit-dirty` ou `miss`.

— `WRITE` (adresse, nœud) : écriture classique d'un nœud dans le cache (le seuil garantit qu'il existe au moins un emplacement disponible). Le nœud n'est pas écrit en mémoire externe (politique *Write-Back*) et devient `dirty`. Les nœuds parents deviennent obsolètes, qu'ils soient en cache ou non. Le compteur de l'ensemble destination est incrémenté. `WRITE` renvoie $(\texttt{status}, (i, j))$ où `status` est un indicateur valant `ok` ou `threshold` et (i, j) la coordonnée d'un nœud victime à évincer lorsque le seuil est atteint (signalé par $\texttt{status} = \texttt{threshold}$).

1. Ensemble d'emplacements susceptibles d'accueillir un nœud de coordonnée donnée (*Set* en Anglais).

— `RESTORE` (adresse) : lecture modifiée d'un nœud. Le comportement dépend de l'état du cache :
  — Le nœud est en cache et `dirty` : lecture dans la mémoire externe, le nœud lu n'est pas stocké dans le cache car il entrerait en conflit avec la version `dirty` du même nœud.
  — Dans tous les autres cas (`hit` et `clean` ou `miss`), l'opération se comporte comme `READ`.

  `RESTORE` renvoie uniquement la valeur lue V.
— `SYNCHRONIZE` (adresse) : si le nœud est en cache et `dirty`, écriture en mémoire externe, le nœud devient `clean`, le compteur de l'ensemble concerné est décrémenté. Sinon l'opération n'a aucun effet. `SYNCHRONIZE` ne renvoie rien.

Seules les opérations `WRITE` et `RESTORE` sont réellement modifiées par rapport à notre version initiale par l'ajout de la gestion du seuil et la disparition des cas où le `WRITE` échouait. Les opérations `READ` et `RESTORE` ne sont modifiées qu'en apparence par le fait qu'elles peuvent désormais toujours stocker un nœud lu dans le cache alors qu'elles ne le faisaient que si un emplacement était disponible dans la première version. Le « *le nœud lu est stocké dans le cache si c'est possible* » est devenu « *le nœud lu est stocké dans le cache* ».

La valeur du seuil τ a un impact sur les performances du cache. Un seuil trop élevé se traduira, lorsque la saturation sera partout presque atteinte, par une faible efficacité des trop peu nombreux emplacements `clean` du cache lors des `READ` et des `RESTORE`. Une valeur trop faible augmentera la fréquence des synchronisation et réduira l'efficacité de la stratégie ALAP. Nous étudierons ces aspects dans le chapitre 5 consacré à la validation.

Ajoutons dès maintenant quelques lemmes concernant les opérations élémentaires du cache en l'absence d'attaque.

Lemme 2. *En l'absence d'attaque* `READ` *préserve la CIP.*

Démonstration. `READ` ne modifie pas la mémoire externe. Si un AMCached était CIP avant `READ`, alors :

— Soit le nœud est une donnée. `READ` ne la stocke pas dans le cache. L'AMCached n'est pas modifié et il est toujours CIP.
— Soit `READ` a trouvé dans le cache le nœud recherché (`hit-clean` ou `hit-dirty`). L'AMCached n'est pas modifié et il est toujours CIP.
— Soit `READ` a lu le nœud recherché en mémoire externe (`miss`) et l'a stocké dans un emplacement libre du cache. Le nœud est devenu `clean` ($N_c = N_m$). Les ensembles de nœuds $N_c(i,j)$ et $N_m(i,j)$ ne sont donc pas modifiés (rappelons que, par extension, pour les nœuds qui ne sont pas en cache, on définit $N_c(i,j) = N_m(i,j)$). L'AMCached n'est pas modifié et il est toujours CIP.
— Soit `READ` a lu le nœud recherché en mémoire externe (`miss`) et l'a stocké dans un emplacement du cache occupé par un nœud `clean`. Le nœud évincé étant `clean`, il vérifie toujours $N_c = N_m$. Le nouveau nœud est également `clean` et vérifie toujours $N_c = N_m$. Les ensembles de nœuds $N_c(i,j)$ et $N_m(i,j)$ ne sont donc pas modifiés. L'AMCached n'est pas modifié et il est toujours CIP.

□

Lemme 3. *En l'absence d'attaque* `WRITE` *préserve la CIP si le nœud écrit est le condensé des versions en mémoire externe de ses fils (hypothèse* ***HCIP****).*

Démonstration. `WRITE` ne modifie pas la mémoire externe. Si un AMCached était CIP avant `WRITE`, alors :

— Soit `WRITE` a stocké le nœud dans un emplacement libre du cache. La seule modification de l'AMCached est l'introduction d'un nouveau nœud $N_c(i, j)$, éventuellement différent de sa version en mémoire. Par hypothèse **HCIP** l'AMCached est toujours CIP.

— Soit `WRITE` a stocké le nœud dans un emplacement du cache occupé par un nœud `clean`. Le nœud évincé étant `clean`, il vérifie toujours $N_c = N_m$. La seule modification de l'AMCached est l'introduction d'un nouveau nœud $N_c(i, j)$, éventuellement différent de sa version en mémoire. Par hypothèse **HCIP** l'AMCached est toujours CIP.

□

Lemme 4. *En l'absence d'attaque* `RESTORE` *préserve la CIP.*

Démonstration. `RESTORE` ne modifie pas la mémoire externe. Si un AMCached était CIP avant `RESTORE`, alors :

— Soit `RESTORE` a lu le nœud recherché en mémoire externe et l'a stocké dans un emplacement libre du cache. Le nœud est devenu `clean` ($N_c = N_m$). Les ensembles de nœuds $N_c(i, j)$ et $N_m(i, j)$ ne sont donc pas modifiés. L'AMCached n'est pas modifié et il est toujours CIP.

— Soit `RESTORE` a lu le nœud recherché en mémoire externe et l'a stocké dans un emplacement du cache occupé par un nœud `clean`. Le nœud évincé étant `clean`, il vérifie toujours $N_c = N_m$. Le nouveau nœud est également `clean` et vérifie toujours $N_c = N_m$. Les ensembles de nœuds $N_c(i, j)$ et $N_m(i, j)$ ne sont donc pas modifiés. L'AMCached n'est pas modifié et il est toujours CIP.

□

Nous pouvons maintenant proposer un algorithme d'éviction vérifiée (algorithme 11) qui évite le problème de la saturation du cache en emplacements `dirty`.

Seul le `WRITE` peut saturer un ensemble et, si cela survient, il ne sature qu'un seul ensemble à la fois. La gestion du cache que nous proposons lance une éviction dès qu'un `WRITE` sature un ensemble et les évictions se poursuivent tant qu'un ensemble est saturé. Au tout début d'une suite d'évictions on peut donc affirmer qu'un ensemble et un seul est saturé. Le nœud à évincer désigné par le `WRITE` responsable de la saturation appartient à cet ensemble et sa synchronisation avec la mémoire externe (ligne 9) libère au moins un emplacement dans l'ensemble (d'autres emplacements peuvent éventuellement être libérés par la synchronisation des frères `dirty` s'il y en a). A la fin de la synchronisation (ligne 14) tous les ensembles du cache ont leur compteur à nouveau strictement inférieur au seuil et possèdent donc tous au moins un emplacement libre ou `clean`. L'écriture du nœud père mis à jour (`WRITE`, ligne 19) n'échoue plus jamais. En revanche, elle introduit un nouveau nœud `dirty` dans le cache. Il est donc possible qu'un nouvel ensemble (le même que précédemment ou un autre) atteigne son seuil et qu'une nouvelle éviction soit nécessaire (ligne 20). Ceci pose la question de la convergence de l'algorithme.

Algorithme 11 Éviction-vérifiée($i \notin \{0, s\}, j = a \times j_0 + j_1$)

```
1:  (u, v) ← (i, j)                                   ▷ Indices temporaires
2:  repeat
3:      (v0, v1) ← (⌊v/a⌋, v mod a)                   ▷ Indices de groupe et de nœud dans le groupe
4:      D = Lecture-vérifiée(u + 1, v0)               ▷ Lecture vérifiée du nœud père
5:      for k ∈ {0, ..., a − 1} do                    ▷ Pour a nœuds frères
6:          (Xk, status) ← READ(u, a × v0 + k)        ▷ Lire avec READ
7:          if status = hit-dirty then                ▷ Si cache hit et dirty
8:              Yk ← RESTORE(u, a × v0 + k)           ▷ Lire avec RESTORE la version en mémoire externe
9:              SYNCHRONIZE(u, a × v0 + k)            ▷ Synchronisation avec SYNCHRONIZE (écriture de Xk en mémoire externe)
10:         else                                      ▷ Nœud non dirty
11:             Yk ← Xk                               ▷ La version en mémoire externe est égale à Xk
12:         end if
13:     end for
14:     D_old ← Digest(Y0, ..., Ya−1)                 ▷ Calcul de l'ancien nœud père du nœud victime
15:     if D_old ≠ D then                             ▷ Si la comparaison échoue
16:         error                                     ▷ Abandonner
17:     end if
18:     D_new ← Digest(X0, ..., Xa−1)                 ▷ Calcul du nouveau nœud père du nœud victime
19:     (status, (u, v)) ← WRITE((u + 1, v0), D_new)  ▷ Écrire le nouveau nœud père du nœud victime avec WRITE
20: until status = ok                                 ▷ Relancer une éviction tant que nécessaire
```

Lemme 5. *L'éviction vérifiée se termine nécessairement.*

Démonstration. La terminaison de la lecture vérifiée est garantie par un critère de monotonie stricte. Soit $\mathcal{D}$ l'ensemble des nœuds `dirty` du cache et $|\mathcal{D}|$ le nombre total de nœuds `dirty` du cache. $|\mathcal{D}|$ diminue strictement lorsque certains frères du nœuds évincé étaient également `dirty` (et sont donc devenus `clean`) et / ou si le nœud père était déjà `dirty` avant l'éviction, auquel cas il le reste tandis que tous ses fils `dirty`, dont au moins le nœud évincé, deviennent `clean`. Dans le pire cas du point de vue de la convergence aucun des frères n'était `dirty` et le père ne l'était pas non plus. Le nœud évincé passe donc de `dirty` à `clean` tandis que son nœud père devient `dirty` alors qu'il ne l'était pas. Le père a remplacé le fils dans $\mathcal{D}$ et $|\mathcal{D}|$ n'a pas changé. Plutôt que $|\mathcal{D}|$, considérons Δ, la somme des distances à la racine de tous les nœuds `dirty` du cache, mesurée en nombre d'arêtes d'arbre :

$$\Delta = \sum_{N(i,j) \in \mathcal{D}} (s - i)$$

Cette somme Δ vaut 12 dans l'exemple de la figure 3.7. Après une itération de l'algorithme d'éviction, même si le nombre de nœuds `dirty` n'a pas changé, le remplacement du fils par le père dans l'ensemble des nœuds `dirty` fait globalement progresser l'ensemble des nœuds `dirty` en direction de la racine de l'AMCached - ils « montent ». Δ diminue donc au minimum d'une unité à chaque éviction.

On peut remarquer que la racine n'appartient pas au cache et que pour tout nœuds `dirty` $N(i, j)$ on a $i < s$ et donc $s - i \geq 1$. Il vient donc :

$$\Delta = \sum_{N(i,j) \in \mathcal{D}} (s - i) \geq \sum_{N(i,j) \in \mathcal{D}} (1) = |\mathcal{D}|$$

Δ est toujours supérieur ou égal à $|\mathcal{D}|$ ($\Delta = 12 \geq |\mathcal{D}| = 5$ dans l'exemple de la figure 3.7). Notons également que le compteur de nœuds `dirty` C_s d'un ensemble $\mathcal{S}_s$ est, par définition,

Algorithme 12 `Lecture-vérifiée`$(i \neq s, j = a \times j_0 + j_1)$

```
1:  (u, v0, v1) ← (i, j0, j1)                                   ▷ Indices temporaires
2:  (X_v1, status) ← READ(u, a × v0 + v1)                        ▷ Lire le nœud avec READ
3:  R ← X_v1                                                    ▷ Donnée à renvoyer
4:  while status = miss do                                      ▷ Tant que nous n'avons pas un nœud sûr
5:      for k ∈ {0, ..., a − 1} \ {v1} do                        ▷ Pour les a − 1 nœuds frères
6:          X_k ← RESTORE(u, a × v0 + k)                         ▷ Lire avec RESTORE
7:      end for
8:      D_ref ← Digest(X_0, ..., X_{a−1})                        ▷ Calcul du condensé
9:      (u, v0, v1) ← (u + 1, ⌊v0/a⌋, v0 mod a)                  ▷ Avancer au niveau supérieur
10:     if u = s then                                           ▷ Si on a atteint le nœud racine
11:         (X_v1, status) ← (Read(s, 0), hit-clean)            ▷ Lire le nœud nœud racine, positionner status à hit-clean
12:     else
13:         (X_v1, status) ← READ(u, a × v0 + v1)                ▷ Lire le nœud avec READ
14:     end if
15:     if X_v1 ≠ D_ref then                                    ▷ Si la comparaison échoue
16:         error                                               ▷ Abandonner
17:     end if
18: end while
19: return R                                                    ▷ Retourner le nœud requis et arrêter
```

toujours inférieur ou égal à $|\mathcal{D}|$. Comme Δ diminue de façon strictement monotone au fil des évictions, elle devient nécessairement inférieure à τ et :

$$\forall\, s, C_s \leq |\mathcal{D}| \leq \Delta < \tau$$

Aucun ensemble n'est plus saturé et la série d'évictions est nécessairement terminée. □

Notons que l'éviction fonctionne avec une valeur du seuil τ égale au maximum T_{set}. Si $\tau < T_{set}$, il est possible d'améliorer la gestion du cache en ne déclenchant l'éviction à coup sûr que si le compteur C_s d'un ensemble atteint T_{set}, car c'est indispensable pour garantir la fonctionnalité. Lorsque $\tau \leq C_s < T_{set}$ pour l'un des ensembles $\mathcal{S}_s$, on pourrait déclencher une éviction uniquement si une autre opération n'est pas déjà en cours et que le CCAM est inactif. Cette gestion modifiée pourrait améliorer les performances en parallélisant les opérations de maintenance du cache (évictions) avec les opérations qui n'utilisent pas le cache.

L'algorithme 12 présente la lecture vérifiée d'un nœud $N(i \neq s, j = a \times j_0 + j_1)$ et montre l'arrêt au plus tôt de la vérification grâce à la stratégie ASAP (ligne 4).

Lemme 6. *En absence d'attaque la lecture vérifiée préserve la CIP.*

Démonstration. Les seules opérations susceptibles de modifier l'AMCached utilisées par la lecture vérifiée sont `READ` et `RESTORE` qui préservent toutes deux la CIP (lemmes 2 et 4). □

Lemme 7. *En absence d'attaque l'éviction vérifiée préserve la CIP.*

Démonstration. Les seules opérations susceptibles de modifier l'AMCached utilisées par l'éviction vérifiée sont :
- — La lecture vérifiée (ligne 4) qui est lancée alors que la CIP est vérifiée et qui préserve la CIP (lemme 6).
- — `READ` et `RESTORE` (lignes 6 et 8) qui préservent la CIP (lemmes 2 et 4).

— `SYNCHRONIZE` (ligne 9) qui écrit les versions en cache des nœuds `dirty` dans la mémoire externe, sans modifier la valeur du nœud père et ne préserve donc pas la CIP. Chaque nœud du groupe subit $N_m \leftarrow N_c$ et devient `clean`. La version en cache de ces nœuds n'est pas modifiée, pas plus que les versions en mémoire externe de leurs fils respectifs. Leurs versions en cache sont donc toujours les condensés des versions en mémoire externe de leurs fils respectifs. Mais leur nœud père, qu'il soit en cache ou non, n'est plus le condensé des versions en mémoire externe de ses fils, puisque certaines d'entre elles ont été modifiées. Cependant cette situation est temporaire car le `WRITE` (ligne 19) qui suit a justement pour effet de mettre le nœud père à jour dans le cache (uniquement) avec le condensé des ses fils (qui sont tous `clean` et donc tels que $N_c = N_m$). Comme l'éviction vérifiée est atomique et que la CIP est restaurée dès le `WRITE`, la combinaison des `SYNCHRONIZE` et du `WRITE` suivant préserve la CIP.

— `WRITE` d'un nœud avec le condensé D_{new} des version en mémoire externe des nœuds fils (ligne 19). La CIP est préservée (lemme 3 avec l'hypothèse **HCIP**).

□

Lemme 8. *En absence d'attaque la lecture vérifiée ne déclenche pas d'alerte de défaut d'intégrité.*

Démonstration. Prouvons ce lemme par l'absurde en supposant que l'AMCached est CIP au début de la lecture vérifiée, qu'aucune des valeurs lues en mémoire externe n'est modifiée mais que la lecture déclenche l'alerte. Pour la comparaison qui échoue entre le condensé calculé sur les fils et le nœud père (ligne 15), les nœuds fils ont tous été lus en mémoire externe par un `READ` avec `miss` et $a - 1$ `RESTORE`. Ils proviennent donc tous de la mémoire externe. Le nœud père X_{v_1}, lui, est obtenu par `READ` et c'est donc sa version en cache, s'il s'y trouve, qui est utilisée. Sinon c'est sa version en mémoire externe, pour laquelle, par définition, $N_c = N_m$. La CIP garantit donc qu'il est le condensé de ses fils en mémoire externe et que la comparaison ne peut pas échouer. □

Lemme 9. *Si l'AMCached est CIP avant l'appel à la lecture vérifiée et que les valeurs lues en mémoire externe sont modifiées par une attaque, alors l'alerte de défaut d'intégrité est toujours déclenchée.*

Démonstration. Prouvons ce lemme par l'absurde en supposant que l'AMCached est CIP au début de la lecture vérifiée, que certaines (au moins une) valeurs lues en mémoire externe sont modifiées par l'attaquant mais que la lecture ne déclenche pas l'alerte et renvoie normalement la valeur $N(i, j)$ demandée. Considérons la dernière comparaison entre un condensé calculé et le nœud père correspondant (ligne 15). Notons $N_c(u, v)$ le nœud père et $N'_m(u - 1, a \times v + k)$ les nœuds fils, éventuellement modifiés par l'attaque, impliqués dans cette comparaison.

Le nœud père provient soit du cache soit de la racine, condition d'arrêt de la lecture vérifiée. Les modifications du cache provoquées par les `READ` et `RESTORE` précédents ne peuvent avoir modifié ce nœud père car elles portent toutes sur des nœuds strictement descendants du nœud père, donc de coordonnées différentes. Le nœud père trouvé en cache ou dans la racine s'y trouvait donc déjà au début de la lecture vérifiée. Par hypothèse de CIP il est donc bien le condensé des valeurs $N_m(u - 1, a \times v + k)$ de ses fils en mémoire externe telles qu'elles étaient

au début :

$$N_c(u,v) = \text{Digest}(N_m(u-1, a \times v), ..., N_m(u-1, a \times v + a - 1)) \tag{3.12}$$

Les valeurs $N'_m(u-1, a \times v + k)$ lues des nœuds fils proviennent toutes de la mémoire externe. La comparaison n'échouant pas, on a donc :

$$N_c(u,v) = \text{Digest}(N'_m(u-1, a \times v), ..., N'_m(u-1, a \times v + a - 1)) \tag{3.13}$$

Par hypothèse d'idéalité de la fonction Digest on conclut donc de 3.12 et 3.13 que :

$$\forall\, 0 \leq k < a, N'_m(u-1, a \times v + k) = N_m(u-1, a \times v + k) \tag{3.14}$$

Les valeurs lues des nœuds fils sont donc telles qu'elles étaient au début de la lecture vérifiée. Elles n'ont donc pas été modifiées par l'attaquant. Le même raisonnement peut alors s'appliquer à leurs fils respectifs. De proche en proche, en descendant vers le niveau i de départ de la lecture vérifiée, on voit donc que tous les nœuds sont non modifiés, ce qui contredit l'hypothèse de l'attaque. □

Lemme 10. *En absence d'attaque l'éviction vérifiée ne déclenche pas d'alerte de défaut d'intégrité.*

Démonstration. La preuve est proche de celle du lemme 8. Supposons, par l'absurde, que l'AMCached est CIP au début de l'éviction vérifiée, qu'aucune des valeurs lues en mémoire externe n'est modifiée mais que l'éviction déclenche l'alerte. Le lemme 8 garantit que l'alerte n'est pas déclenchée par la lecture vérifiée de la ligne 4 car la CIP est vérifiée lors de l'appel. C'est donc une comparaison entre le condensé calculé sur les fils et le nœud père (ligne 16) qui échoue. Pour cette comparaison les nœuds fils Y_k ont tous été lus soit :

- — en cache à un emplacement `clean` par un `READ`, donc tel que $N_c = N_m$,
- — en mémoire externe par un `READ` avec `miss`, donc tel que $N_c = N_m$,
- — en mémoire externe par un `RESTORE`, donc tel que $N_c = N_m$.

Les Y_k sont donc tous égaux à la valeur du nœud en mémoire externe. Le nœud père $D = N(u+1, v_0)$, lui, est obtenu par lecture vérifiée et c'est donc sa version en cache, s'il s'y trouve, qui est utilisée. Sinon c'est sa version en mémoire externe, validée, pour laquelle, par définition, $N_c = N_m$. La CIP garantit donc qu'il est le condensé de ses fils en mémoire externe et que la comparaison ne peut pas échouer. □

Lemme 11. *Si l'AMCached est CIP avant l'appel à l'éviction vérifiée et que les valeurs lues en mémoire externe sont modifiées par une attaque, alors l'alerte de défaut d'intégrité est toujours déclenchée.*

Démonstration. La preuve est semblable à celle du lemme 9. Supposons par l'absurde que l'AMCached est CIP au début de l'éviction vérifiée, que certaines (au moins une) valeurs lues

en mémoire externe sont modifiées par l'attaquant mais que l'éviction ne déclenche pas l'alerte. Considérons la dernière comparaison entre un condensé calculé et le nœud père correspondant (ligne 15). Le nœud père D a été lu par lecture vérifiée et le lemme 9 garantit que, puisque la lecture ne déclenche pas l'alerte, il est valide et provient du cache s'il s'y trouve, de la mémoire externe sinon. On a donc $D = N_c(u+1, v_0)$. Notons $N'_m(u, a \times v_0 + k)$ les nœuds fils, éventuellement modifiés par l'attaque, impliqués dans cette comparaison.

Le nœud père provient soit du cache soit de la racine, condition d'arrêt de la lecture vérifiée. Les modifications du cache provoquées par les `READ` et `RESTORE` précédents ne peuvent avoir modifié ce nœud père car elles portent toutes sur des nœuds strictement descendants du nœud père, donc de coordonnées différentes. Le nœud père trouvé en cache ou dans la racine s'y trouvait donc déjà au début de la lecture vérifiée. Par hypothèse de CIP il est donc bien le condensé des valeurs $N_m(u-1, a \times v + k)$ de ses fils en mémoire externe telles qu'elles étaient au début :

$$N_c(u, v) = \text{Digest}(N_m(u-1, a \times v), ..., N_m(u-1, a \times v + a - 1)) \tag{3.15}$$

Les valeurs $N'_m(u-1, a \times v + k)$ lues des nœuds fils proviennent toutes de la mémoire externe. La comparaison n'échouant pas, on a donc :

$$N_c(u, v) = \text{Digest}(N'_m(u-1, a \times v), ..., N'_m(u-1, a \times v + a - 1)) \tag{3.16}$$

Par hypothèse d'idéalité de la fonction Digest on conclut donc de 3.15 et 3.16 que :

$$\forall\, 0 \leq k < a, N'_m(u-1, a \times v + k) = N_m(u-1, a \times v + k) \tag{3.17}$$

Les valeurs lues des nœuds fils sont donc telles qu'elles étaient au début de la lecture vérifiée. Elles n'ont donc pas été modifiées par l'attaquant. Le même raisonnement peut alors s'appliquer à leurs fils respectifs. De proche en proche, en descendant vers le niveau i de départ de la lecture vérifiée, on voit donc que tous les nœuds sont non modifiés, ce qui contredit l'hypothèse de l'attaque. □

L'algorithme 13 présente l'écriture vérifiée d'une donnée V dans le nœud $N(0, j = a \times j_0 + j_1)$ et montre également l'arrêt au plus tôt de la mise à jour grâce à la stratégie ALAP. Note : l'écriture vérifiée d'un nœud interne d'arbre n'est pas utile.

Lemme 12. *En absence d'attaque l'écriture vérifiée préserve la CIP.*

Démonstration. Le cache, la racine et la mémoire externe ne sont pas modifiés par les lectures non vérifiées des données (ligne 2). Si l'AMCached était CIP au début de l'écriture vérifiée il l'est donc toujours lors de l'appel à la lecture vérifiée (ligne 5). Comme la lecture vérifiée préserve la CIP (lemme 6), l'AMCached est toujours CIP après. Les seules autres opérations susceptibles de compromettre la CIP sont donc :

Algorithme 13 Écriture-vérifiée$((0, j = a \times j_0 + j_1), V)$

```
1: for k ∈ {0, ..., a − 1} do                          ▷ Lecture non vérifiée du groupe de données
2:     X_k ← Read(0, a × j_0 + k)
3: end for
4: D ← Digest(X_0, ..., X_{a−1})                       ▷ Calcul du condensé
5: D_ref ← Lecture-vérifiée(1, j_0)                    ▷ Lecture vérifiée du père
6: if D ≠ D_ref then                                   ▷ Si la comparaison échoue
7:     error                                           ▷ Abandonner
8: end if
9: X_{j_1} ← V                                         ▷ La nouvelle valeur de la donnée
10: Write((0, j), V)                                   ▷ Écriture non vérifiée de la nouvelle valeur de la donnée
11: V ← Digest(X_0, ..., X_{a−1})                      ▷ Calcul du nouveau nœud père du groupe de données
12: (status, (u, v)) ← WRITE((1, j_0), V)              ▷ Écrire avec WRITE
13: if status = threshold then                         ▷ Si un ensemble est saturé
14:     Éviction-vérifiée(u, v)                        ▷ Lancer une éviction vérifiée
15: end if
```

— L'écriture non vérifiée de donnée (ligne 10) qui modifie une donnée en mémoire externe sans modifier la valeur du nœud père et ne préserve donc pas la CIP. Cependant cette situation est temporaire car le condensé calculé ligne 11 utilise la nouvelle valeur en mémoire externe du nœud écrit et les valeurs lues en mémoire externe de ses frères. Lors du `WRITE` (ligne 12) du nœud père il est donc bien le condensé de ses fils en mémoire externe et la CIP est rétablie (lemme 3 avec l'hypothèse **HCIP**).

— `WRITE` d'un nœud avec le condensé des nœuds fils (les données) en mémoire externe. La CIP est préservée (lemme 3 avec l'hypothèse **HCIP**).

— L'éviction vérifiée (ligne 14) qui préserve la CIP (lemme 7).

□

Lemme 13. *En absence d'attaque l'écriture vérifiée ne déclenche pas d'alerte de défaut d'intégrité.*

Démonstration. La preuve est proche de celle du lemme 8. Supposons, par l'absurde, que l'AMCached est CIP au début de l'écriture vérifiée, qu'aucune des valeurs lues en mémoire externe n'est modifiée mais que l'éviction déclenche l'alerte. Le lemme 8 garantit que l'alerte n'est pas déclenchée par la lecture vérifiée de la ligne 4 car la CIP est toujours vérifiée lors de l'appel (voir la preuve du lemme 12). De même, le lemme 10 garantit que l'alerte n'est pas déclenchée par l'éviction vérifiée de la ligne 14 car la CIP est également toujours vérifiée lors de l'appel (voir la preuve du lemme 12). C'est donc la comparaison entre le condensé calculé sur les fils et le nœud père (ligne 6) qui échoue.

Pour cette comparaison les nœuds fils X_k ont tous été lus en mémoire externe. Le nœud père de référence $D_{ref} = N(1, j_0)$, lui, est obtenu par lecture vérifiée et c'est donc sa version en cache, s'il s'y trouve, qui est utilisée. Sinon c'est sa version en mémoire externe, validée, pour laquelle, par définition, $N_c = N_m$. La CIP garantit donc qu'il est le condensé de ses fils en mémoire externe et que la comparaison ne peut pas échouer. □

Lemme 14. *Si l'AMCached est CIP avant l'appel à l'écriture vérifiée et que les valeurs lues en mémoire externe sont modifiées par une attaque, alors l'alerte de défaut d'intégrité est toujours déclenchée.*

Démonstration. La preuve est semblable à celle du lemme 9. Supposons par l'absurde que l'AMCached est CIP au début de l'écriture vérifiée, que certaines (au moins une) valeurs lues

en mémoire externe sont modifiées par l'attaquant mais que l'écriture ne déclenche pas l'alerte. Par les lemmes 9 et 11 nous savons qu'aucune des valeurs lues en mémoire externe par la lecture vérifiée ligne 5 et l'éviction vérifiée ligne 14 ne sont modifiées. Ce sont donc les données lues par les lectures non vérifiées ligne 2 (les X_k) qui le sont. Considérons la comparaison entre le condensé calculé et le nœud père correspondant (ligne 6). Le nœud père D_{ref} est lu par lecture vérifiée et le lemme 9 garantit que, puisque la lecture ne déclenche pas l'alerte, il est valide et provient du cache s'il s'y trouve, de la mémoire externe sinon. On a donc, par CIP :

$$\begin{aligned} D_{ref} &= N_c(1, j_0) \\ &= \text{Digest}(N_m(0, a \times j_0), ..., N_m(0, a \times j_0 + a - 1) \end{aligned}$$

Notons $N'_m(0, a \times j_0 + k) = X_k$ les nœuds fils (données), éventuellement modifiés par l'attaque, impliqués dans cette comparaison. Si la comparaison ligne 6 n'échoue pas on a donc :

$$\begin{aligned} &D = D_{ref} \Leftrightarrow \\ &\text{Digest}(X_0, ..., X_{a-1}) = N_c(1, j_0) \Leftrightarrow \\ &\text{Digest}(N'_m(0, a \times j_0), ..., N'_m(0, a \times j_0 + a - 1)) = \\ &\quad \text{Digest}(N_m(0, a \times j_0), ..., N_m(0, a \times j_0 + a - 1)) \end{aligned}$$

Par hypothèse d'idéalité de la fonction Digest on conclut donc que :

$$\forall\, 0 \leq k < a, N'_m(0, a \times j_0 + k) = N_m(0, a \times j_0 + k)$$

Les valeurs lues des données sont donc telles qu'elles étaient au début de l'écriture vérifiée. Elles n'ont donc pas été modifiées par l'attaquant, ce qui contredit l'hypothèse de l'attaque. □

3.3.4.4 Correction fonctionnelle de la gestion du cache proposée

Nous pouvons maintenant conclure par la preuve de correction fonctionnelle de la gestion de cache proposée avec stratégie ALAP et cache *Write-Back*.

Théorème 1. *En l'absence d'attaque la gestion de cache proposée préserve la CIP.*

Démonstration. Après l'initialisation l'AMCached est CIP (lemme 1) et le CPU émet des requêtes de lecture et d'écriture au CCAM, qui les traduit en lectures, écritures et évictions vérifiées, conformément aux algorithmes présentés. Chacune de ces fonctions préserve la CIP en l'absence d'attaque (lemmes 6, 7 et 12). □

Théorème 2. *En l'absence d'attaque la gestion de cache proposée ne déclenche pas d'alerte de défaut d'intégrité.*

Démonstration. Après l'initialisation l'AMCached est CIP et le CPU émet des requêtes de lecture et d'écriture au CCAM, qui les traduit en lectures, écritures et évictions vérifiées, conformément aux algorithmes présentés. Aucune de ces fonctions ne déclenche d'alerte en l'absence d'attaque (lemmes 8, 10 et 13). □

Théorème 3. *Si des valeurs lues en mémoire externe sont modifiées par une attaque, alors l'alerte de défaut d'intégrité est toujours déclenchée par la gestion de cache proposée.*

Démonstration. Après l'initialisation l'AMCached est CIP et le CPU émet des requêtes de lecture et d'écriture au CCAM, qui les traduit en lectures, écritures et évictions vérifiées, conformément aux algorithmes présentés. Chacune de ces fonctions déclenche l'alerte si l'une des valeurs qu'elle lit en mémoire externe est modifiée par un attaquant (lemmes 9, 11 et 14). □

3.3.5 Combinaison des arbres creux et du cache d'arbre

L'utilisation des deux familles d'optimisation présentées (arbres creux et cache ASAP/ALAP) ne pose pas de difficulté particulière.

Les AMC modifient le comportement des arbres de Merkle en introduisant les nœuds de valeur `NULL` ce qui impose des modifications des différentes opérations de traitement d'arbre (lecture et écriture vérifiées). En ce qui concerne la gestion du cache, la méthode des AMC-I utilise la même gestion de cache que les AMR, en utilisant les mêmes opérations élémentaires (`READ`, `WRITE`, `RESTORE` et `SYNCHRONIZE`) afin d'assurer la cohérence de l'arbre entre les nœuds stockés dans le cache et ceux stockés dans la mémoire externe. Les nœuds de valeur `NULL` sont traités de la même manière que les autres. Un nœud `NULL` présent dans le cache est un nœud sûr et le processus de vérification/mise à jour s'arrête à ce niveau.

La méthode des AMC-NI nécessite une petite modification de la gestion du cache. Les nœuds appartenant à une branche non initialisée ne doivent pas stockés dans le cache (qui ne doit contenir que des nœuds vérifiés). Par conséquent, nous modifions l'opération `RESTORE` et ajoutons une nouvelle opération `XREAD`. Ces deux opérations sont décrites comme suit :

- `RESTORE` (adresse) : Si cache `hit` et non-`dirty`, lire la donnée demandée à partir du cache, et s'il est un bloc `dirty`, le récupérer depuis la mémoire externe sans le stocker dans le cache. Autrement, récupérer la donnée demandée à partir de la mémoire externe et la stocker dans le cache dans un bloc `clean` (si le nœud n'est pas `NULL`).
- `XREAD` (adresse) : Si cache `hit`, lire la donnée demandée à partir du cache. Sinon, récupérer la donnée depuis la mémoire externe sans la stocker dans le cache.

L'opération `XREAD` est utilisée durant le parcours dans la partie non initialisée de la branche. Tant que le drapeau d'erreur est positionné, l'opération `XREAD` est utilisée pour lire les nœuds pères et leurs nœud frères afin d'éviter le remplissage du cache avec un nœud de valeur `NULL` ou inconnue.

3.4 Conclusion

Dans ce chapitre nous avons présenté deux variantes des arbres de Merkle en décrivant les opérations d'initialisation, de vérification et de mise à jour pour chaque variante et en évaluant leurs performances. Ces variantes évitent la fonction récursive utilisée par les arbres de Merkle réguliers. La variante des AMC-I permet d'accélérer la phase d'initialisation sans effet indésirable ultérieur sur les opérations de vérification ou de mise à jour. Elle utilise les mêmes opérations de cache que les AMR. Pour la variante des AMC-NI, la phase d'initialisation est beaucoup plus rapide (presque instantanée) par rapport aux AMR ou aux AMC-I. Par contre, les accès de lecture et d'écriture dans la mémoire externe, après l'initialisation, prennent plus de temps et nécessitent plus de calculs de MAC lorsqu'ils tombent dans une branche non initialisée. L'utilisation des arbres de Merkle creux, avec l'une ou l'autre des deux méthodes proposées, permettent d'accélérer la phase d'initialisation par rapport aux arbres de Merkle réguliers, comme le montre notre évaluation des performances.

Le choix entre les variantes AMR, AMC-I et AMC-NI des arbres de Merkle dépend principalement du système sur lequel elles peuvent être utilisées. Par exemple, dans un contexte où une zone mémoire protégée en intégrité par des arbres de Merkle est allouée plusieurs fois pour le compte de processus logiciels différents, il est nécessaire de procéder à plusieurs initialisations d'arbres de Merkle. Les arbres de Merkle creux sont alors potentiellement plus intéressants que les arbres de Merkle réguliers.

Dans un deuxième temps nous avons décrit une gestion de cache d'arbre *Write-Back* avec stratégie ASAP/ALAP. Nous avons démontré que les incohérences introduites par les mises à jour retardées de la stratégie ALAP peuvent être contrôlées. Nous avons démontré la correction fonctionnelle de la gestion de cache proposée, c'est à dire l'absence de fausses alertes de défaut d'intégrité et le déclenchement systématique d'alertes en cas d'attaque. Nous n'avons pas analysé les gains en performance obtenus grâce au cache car, comme pour toute architecture de cache, ils sont trop fortement dépendants des flux d'accès mémoire du CPU. Le chapitre 5 étudiera ces gains en performance par simulation.

Nous avons finalement expliqué comment les deux types d'optimisations peuvent coexister, au prix, dans le cas des arbres creux non initialisés, d'une légère modification de la gestion du cache.

Dans le prochain chapitre nous présenterons le projet SecBus qui permet de protéger en confidentialité et en intégrité le bus mémoire entre un SoC avec ses mémoires externes, ainsi que le contenu des mémoires externes. Cette protection couvre les attaques logicielles et matérielles, y compris les attaques en intégrité par rejeu. Dans le cadre de cette thèse nous avons participé au développement de plusieurs modules matériels de protection. Nous avons également implémenté les deux variantes des arbres de Merkle creux dans le simulateur SystemC de l'architecture SecBus et mesuré par simulations les performances obtenues.

Chapitre 4

Cas d'étude : projet SecBus

Sommaire

4.1 Introduction **101**
4.1.1 Modèle de menace 102
4.1.2 Objectifs de sécurité 103
4.1.3 Architecture et structures de données 108
4.2 Architecture logicielle **111**
4.2.1 Bootloader 111
4.2.2 SSM 112
4.3 Architecture matérielle **114**
4.3.1 Modules d'interface 115
4.3.2 Modules de protection 116
4.4 Conclusion **118**

Les contributions présentées dans les chapitres précédents ont été intégrées dans l'architecture SecBus qui sera présentée dans ce chapitre. Nous introduirons tout d'abord le modèle d'attaque et les objectifs de sécurité considérés. Ensuite, nous détaillerons dans la section 4.2 la partie logicielle et dans la section 4.3 la partie matérielle de l'architecture SecBus.

4.1 Introduction

SecBus est une architecture matérielle et logicielle ayant pour objectif de garantir la confidentialité et l'intégrité du contenu des mémoires externes d'un système à base de microprocesseurs. Comparé aux plates-formes de calcul sécurisées de la littérature (présentées dans la section 2.4, page 61), SecBus est caractérisé par les principes fondateurs suivants :

- Le ou les cœurs de processeur (ainsi que leurs caches et éventuelles MMU) du système ne doivent pas être modifiés.
- La taille totale de la zone protégée de la mémoire externe n'est pas limitée par SecBus.

— Les applications logicielles qui ne nécessitent pas de protection peuvent s'exécuter sans intervention de SecBus.
— La granularité de protection de la mémoire est basée sur des pages mémoire physiques du système.

4.1.1 Modèle de menace

Le projet SecBus [51, 52] s'inscrit dans le contexte de la sécurité des systèmes embarqués. En plus des vulnérabilités logicielles classiques touchant tous les systèmes informatiques (et donc également les systèmes embarqués) comme par exemple les débordements de tampons (*buffer overflow*), ces systèmes sont également vulnérables à des attaques matérielles. Ces attaques exploitent le fait que l'adversaire peut avoir accès physiquement au système (soit suite à un vol, soit, dans certaines conditions, l'utilisateur légitime peut être susceptible d'attaquer le système).

Ces attaques matérielles peuvent viser n'importe quel composant matériel du système embarqué :
— La puce principale (en général un SoC, *System on Chip*, contenant un ou plusieurs cœurs de calcul, des IP spécialisées dans certains traitements, etc.) : par exemple les attaques par analyse des canaux auxiliaires (SCA, *Side-Channel Analysis*), par injection de faute, par micro-probing, etc.
— Les bus de communication entre la puce principale et ses mémoires et différents périphériques : espionnage, modification des données à la volées, etc.

L'architecture SecBus vise principalement à palier toutes les attaques permettant à un adversaire d'espionner ou de modifier le contenu des mémoires externes du système.

Le modèle de menace considéré est le suivant. L'adversaire a accès à l'ensemble du système embarqué et peut donc notamment espionner et modifier les échanges entre le SoC et ses périphériques externes (et notamment la mémoire externe). On suppose néanmoins qu'il n'a pas la possibilité d'attaquer directement le SoC. Les traitements qui sont effectués au sein du SoC et les données qui y sont traitées sont donc inaccessibles et inaltérables par l'adversaire. Ce modèle de menace est illustré par la figure 4.1.

Les attaques visant directement le SoC sont volontairement écartées car elles sont en général plus coûteuses que les attaques contre les bus de communications et les périphériques externes. De plus, il existe déjà dans la littérature de nombreuses contre-mesures, plus ou moins efficaces, permettant de s'en protéger.

Sont donc principalement considérées les attaques visant les bus de communications du SoC et ses périphériques. Ces bus peuvent être tous protégés par des solutions purement logicielles (via des mécanismes de chiffrement et de protection d'intégrité) à l'exception notable du bus mémoire. La protection de ce dernier nécessite la présence d'un module matériel dédié.

D'autres attaques peuvent permettre à un adversaire d'avoir accès au contenu des mémoires externes voire de le modifier. C'est notamment le cas des attaques DMA où des périphériques, comme les GPU (*Graphics Processing Unit*), les contrôleurs HDD (*Hard Disk Drive*) ou les cartes réseaux, exploitent l'accès DMA pour accéder à la mémoire externe. Une telle attaque a

été montée avec succès contre les consoles de jeux vidéos en exploitant leurs lecteurs DVD.

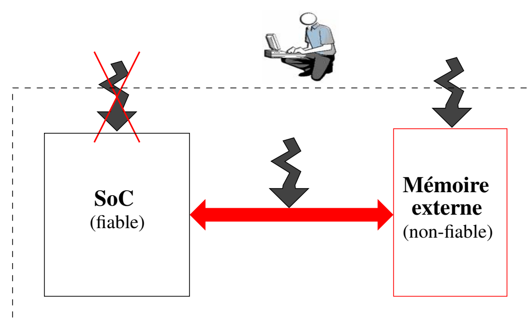


FIGURE 4.1 – Zones d'attaques considérées par SecBus.

Un adversaire capable d'espionner le contenu des mémoires externes peut obtenir de l'information sur le fonctionnement du système (rétro-conception, vol de propriété industrielle, vol de données sensibles comme des clés cryptographique, etc.). Si de plus il est capable d'altérer le contenu des mémoires, il peut perturber son bon fonctionnement et en prendre le contrôle (élévation de privilèges en modifiant les structures de contrôle du système d'exploitation, contournement de contrôle d'accès, etc.).

4.1.2 Objectifs de sécurité

Le principal objectif de sécurité de SecBus est de permettre la conception d'une plate-forme de calcul fiable. Cette plate-forme reste sécurisée même en présence d'un adversaire. La pile logicielle est conçue peut être certifiée à distance. Ce qui permet au propriétaire de la plate-forme d'avoir la garantie que l'intégrité de la plate-forme n'a pas été compromise.

La protection couvre la *confidentialité* et l'*intégrité* des communications entre le SoC et les composants externes. La plupart de ces communications (comme celles avec l'interface réseaux ou celles avec les stockages de masse) peuvent être protégées par des techniques cryptographiques classiques implémentées en logiciel (chiffrement et déchiffrement des données pour garantir la confidentialité et calcul de condensés cryptographiques (à base des fonctions de `hash` ou de MAC) pour vérifier l'intégrité).

Par contre, il existe une exception dans cette liste de communications : les bus entre le microprocesseur et ses mémoires externes. En effet, si l'attaquant falsifie le bus mémoire et / ou les mémoires externes, nous n'aurons pas la garantie que le logiciel et les primitives cryptographiques utilisées sont celles implémentées par le concepteur de la plateforme. Dans de nombreux cas, faire basculer un seul bit du bus d'adresse ou de données durant une transaction de lecture ou d'écriture peut suffire à forcer l'exécution d'un code arbitraire dans un mode privilégié.

Le projet SecBus utilise un module cryptographique matériel (le HSM, *Hardware Security Module*), localisé à l'intérieur du SoC (entre le bus principal et le contrôleur mémoire), chargé d'appliquer à la volée les différentes primitives cryptographiques. Toutes les données sortant du SoC (environnement protégé d'après le modèle d'attaque considéré) passent par le HSM qui, en

fonction de la politique de sécurité à appliquer, les chiffre et / ou protège leur intégrité. Lorsque des données sont récupérées depuis la mémoire, elles passent par le HSM qui éventuellement les déchiffre et vérifie leur intégrité avant qu'elles ne soient utilisées par le SoC.

Afin d'optimiser les performances les zones mémoire sont protégées en fonction des besoins de sécurité. En particulier, les données qui ne nécessitent aucune protection ne sont pas protégées, ce qui réduit l'impact global sur les performances. La granularité de définition des politiques de sécurité de SecBus est la page de mémoire physique. De plus, les primitives utilisées pour la protection de la confidentialité et l'intégrité dépendent de la nature des données sur lesquelles elles sont appliquées. On distingue ainsi les pages mémoires en lecture seule (RO, *Read-Only*) [1] et les pages en lecture et écriture (RW, *Read-Write*). Nous allons maintenant présenter les différentes primitives utilisées dans ces deux cas.

4.1.2.1 Confidentialité

Le choix entre les primitives de chiffrement est effectué suivant le type des pages mémoires. Dans chaque cas, la granularité du chiffrement correspond à une ligne de cache entière : chaque groupe de données correspondant à une ligne de cache est chiffré indépendamment des autres.

4.1.2.1.1 Pages RO Les données des pages RO sont chiffrées à l'aide du chiffrement par bloc en mode compteur. C'est l'adresse du bloc qui sert de compteur (voir détails section 1.2.1.2.3 page 29). Dans le cas d'une lecture (cas le plus pénalisant du point de vue des performances), comme l'adresse est une information disponible dès que la requête parvient au HSM, le calcul des masques qui ne dépendent que du compteur (ici l'adresse) et de la clé de chiffrement) peut commencer dès le début de l'accès mémoire, avant même d'avoir reçu les données chiffrées de la mémoire. Le calcul des masques et l'accès aux données chiffrées en mémoire sont donc parallélisés. Dès que les masques et les données chiffrées sont disponibles, un simple ou exclusif (xor) permet d'obtenir instantanément les données en clair. Dans le cas où la latence de la mémoire est plus grande que le temps de calcul des masques, la latence supplémentaire introduite par le déchiffrement est nulle (opération xor). Comme le mode compteur interdit de chiffrer deux données avec le même masque ce mode ne peut donc être utilisée que pour des pages RO.

4.1.2.1.2 Pages RW Les données des pages RW sont chiffrées à l'aide du chiffrement par bloc en mode CBC (*Cipher Block Chaining*, voir détails section 1.2.1.2.3 page 26). Lors d'une opération de chiffrement, un vecteur d'initialisation (IV, *Initalization Vector*) est choisi aléatoirement et est stocké en mémoire dans une page dédiée. Cet IV sert à chiffrer le premier bloc, constitué de l'adresse de la ligne. Les blocs de données du groupe sont chiffrés à la suite en mode CBC. Cette organisation, qui fait dépendre le chiffrement d'un groupe de son adresse en mémoire, rend moins probable une collision d'IV et son exploitation par un adversaire.

En résumé, dans l'architecture de SecBus, une page mémoire peut être configurée avec une

[1] C'est à dire les pages mémoires qui sont initialisées une fois et une seule lors de leur allocation par le système d'exploitation et dans lesquelles tous les accès suivants, jusqu'à la libération, sont des lectures. Les pages de code ou de données constantes des applications logicielles sont des pages RO.

confidentialité sans protection (NONE), protégée par un chiffrement par bloc en mode compteur (CTR) ou un chiffrement par bloc en mode CBC. Dans le mode CBC, des IV aléatoires et imprévisibles peuvent être utilisés de sorte que deux chiffrements successifs de la même donnée donne des chiffrés différents (sécurité sémantique). L'architecture SecBus prévoit l'utilisation des IV dans les moindres détails mais ils sont optionnels. Dans les simulateurs et démonstrateurs matériels actuellement implémentés les IV ne sont pas utilisés.

Dans les deux modes, compteur et CBC, les adresses mémoires sont impliquées afin de spécialiser la fonction de chiffrement et de la rendre unique pour chaque adresse. Les collisions exploitables sont ainsi moins probables.

Note : lorsqu'une page RO protégée par le mode compteur est libérée, puis qu'une autre page RO, également protégée par le mode compteur, est allouée plus tard au même endroit, il est essentiel et de la responsabilité du logiciel de contrôle d'assurer que la même clef de chiffrement n'est pas réutilisée. En effet, le mode compteur impose qu'un masque ne soit réellement jamais réutilisé, ce qui serait le cas si deux pages RO à la même adresse étaient chiffrées avec la même clef.

4.1.2.2 Intégrité

Comme pour les primitives de chiffrement, les primitives de vérification d'intégrité sont choisies en fonction du type des pages mémoires.

4.1.2.2.1 Pages RO On peut tout d'abord noter que ce type de pages n'est pas menacé par les attaques par rejeu. Dans ces pages, chaque ligne est protégée grâce au calcul d'un MAC, calculé à l'aide de l'algorithme CBC-MAC sur les données en prenant également en compte l'adresse. Cette primitive permet ainsi de palier les attaques par injection et par permutation spatiale. Les MAC sont stockés dans la mémoire externe dans des pages dédiées nommées MS (*MAC Set*). La figure 4.2 montre des pages RO, RW et MS en mémoire.

4.1.2.2.2 Pages RW L'utilisation d'un MAC combiné à l'adresse ne permet pas détecter les attaques par rejeu. C'est pour cette raison que l'intégrité des pages RW est protégé à l'aide d'arbre de MAC (ou arbres de Merkle).

Les MAC (c'est-à-dire les nœuds de l'arbre) sont stockés dans la mémoire externe dans des pages mémoires nommées MT (*MAC Tree*). La figure figure 4.3 montre des pages RO, RW, MS et MT en mémoire. La gestion des arbres de Merkle est détaillée dans la section 2.3 ainsi que des estimations de performance. Les arbres de Merkle permettent de détecter toute altération dans la mémoire externe mais ils dégradent les performances du système car ils augmentent le nombre d'accès à la mémoire et le nombre de calculs de MAC, de façon potentiellement significative. Dans le chapitre 3, nous avons présenté quelques optimisations de la gestion des arbres de Merkle afin de réduire leur impact sur les performances.

Pour résumer, l'intégrité d'une page mémoire peut être configurée sans protection (NONE), protégée avec des ensembles de MAC (MAC-Set) ou des arbres de Merkle (MAC-Tree).

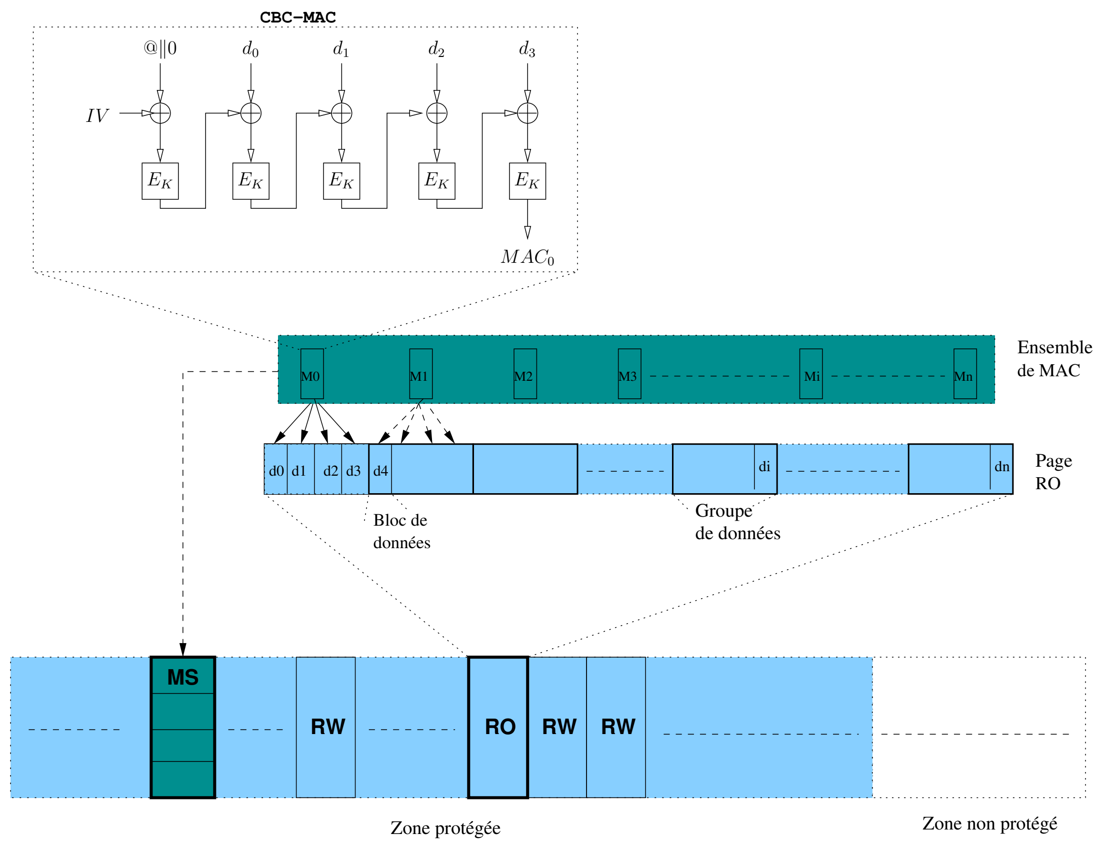


FIGURE 4.2 – Protection des pages RO en intégrité par des MAC.

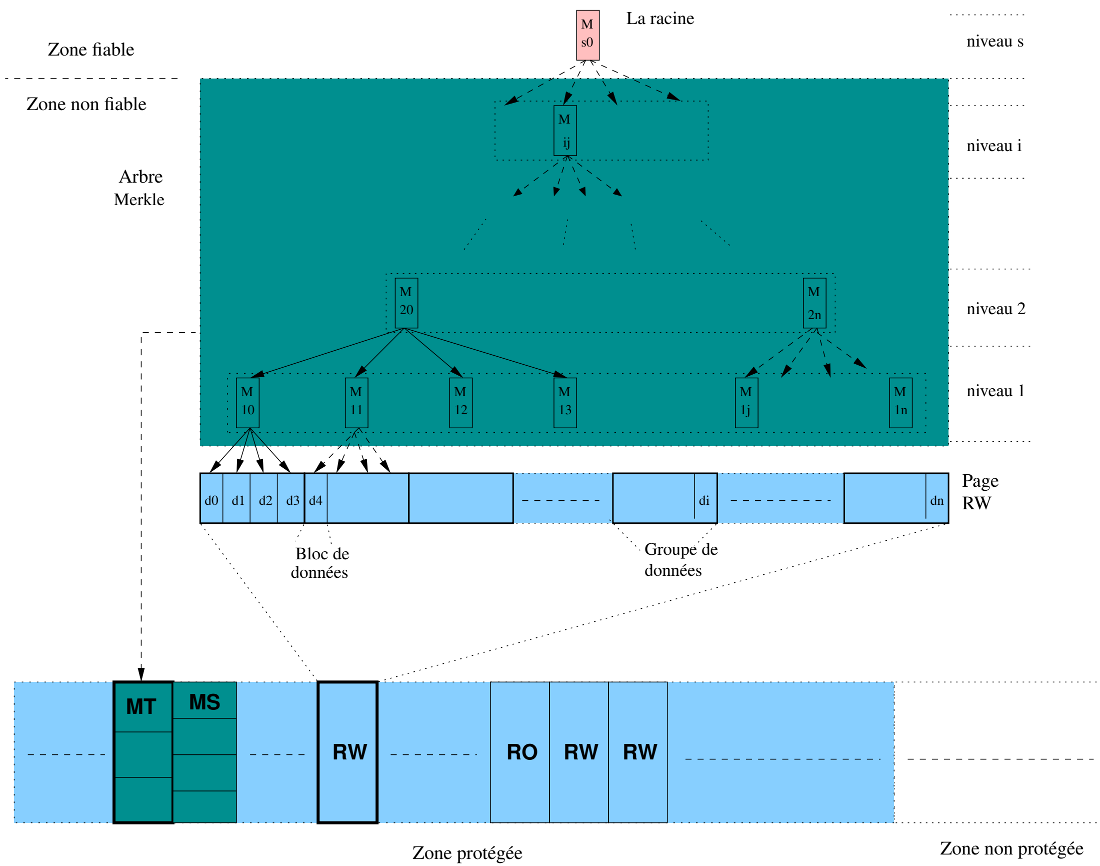


FIGURE 4.3 – Protection des pages RW en intégrité par des arbres de MAC.

Dans le contexte de SecBus, quatre types de pages mémoire cohabitent : les pages RO, RW, MS (ensembles de MAC) qui protègent les pages RO et, MT (arbres de MAC) qui protègent les pages RW, auxquels il convient d'ajouter un cinquième type pour les pages IV si les IV sont utilisés.

Dans les implémentations de SecBus auxquelles nous avons contribué et que nous avons utilisées, l'algorithme de chiffrement par bloc choisi est DES-X (blocs de 64 bits, clé secrète de 184 bits) et les MAC sont calculés sur des groupes de quatre blocs de données consécutifs ($4 \times 64 = 256\, bits$, ce qui correspond à une ligne de cache du cœur ARM 32 bits). Donc, une page MS de taille $S \in \{4\, Kio, 64\, Kio, 1\, Mio, 16\, Mio\}$ contient quatre ensembles de MAC, chacune protège une page RO de taille S. Avec une structure quaternaire des arbres de MAC, une page MT de taille S contient trois arbres de MAC, chacun protégeant une page RW de taille S (figure 4.4).

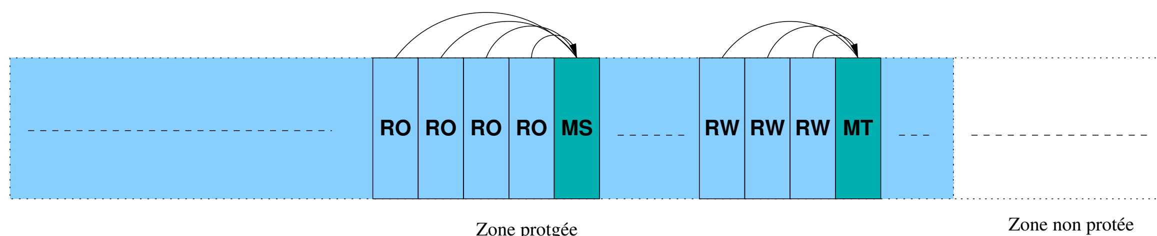


FIGURE 4.4 – Les différents types de pages mémoires

4.1.3 Architecture et structures de données

Les primitives cryptographiques utilisées pour garantir l'intégrité et la confidentialité sont calculées à l'intérieur du SoC (considéré comme zone sécurisée). Cela permet de garantir qu'aucune information sensible ne sort du SoC sans chiffrement et ne peut être altérée sans détection. SecBus est composé de deux parties : un module matériel (HSM, *Hardware Security Module*) qui prend en charge les opérations de la sécurité (chiffrement, déchiffrement et vérification d'intégrité) et d'un module logiciel permettant de configurer le HSM. L'ajout du HSM implique quelques modifications matérielles du SoC qui doivent rester les plus faibles et les plus transparentes possibles.

Tous les accès de lecture et d'écriture vers la mémoire externe passent par le HSM et, si nécessaire, des primitives cryptographiques sont appliquées. Le meilleur endroit possible pour le HSM est donc à l'intérieur de la puce entre le bus d'interconnexion et le contrôleur mémoire. La figure 4.5 présente un SoC avec le module HSM.

Les parties logicielle et matérielle de SecBus se complètent. L'assistance logicielle, assurée par le gestionnaire de sécurité logicielle (SSM, *Software Security Manager*), consiste à décider quelle politique de sécurité doit être appliquée à chaque page de mémoire. Dans l'architecture SecBus, le HSM est configuré et piloté par le SSM qui définit les politiques de sécurité (SP, *Security Policies*) et qui lie les pages mémoire avec les SP en utilisant des structures de données dédiées. Le HSM consulte ces structures de données de façon autonome et applique les politiques de sécurité spécifiées. Cette coopération entre le logiciel et le matériel est directement

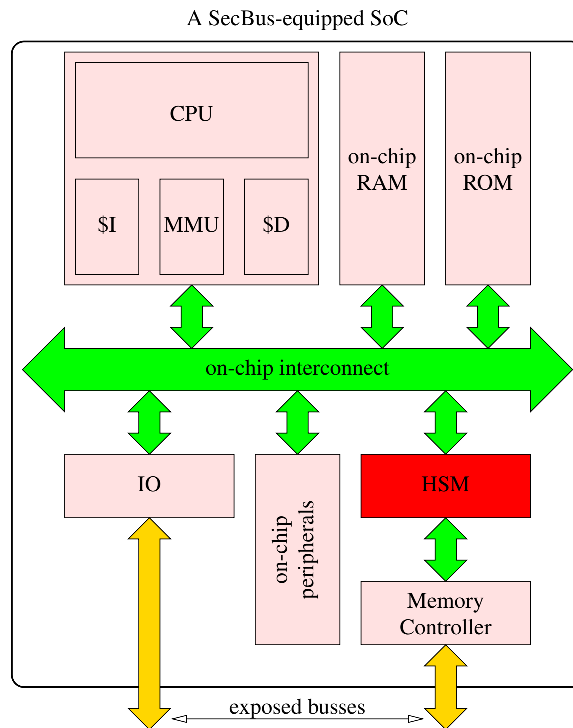


FIGURE 4.5 – Le HSM à l'intérieur du SoC

inspirée de celle qui existe entre le gestionnaire de mémoire d'un système d'exploitation et la MMU (Memory Management Unit) du processeur.

4.1.3.1 Politiques de sécurité

L'architecture SecBus permet la définition de sécurité avec le niveau de granularité des pages mémoire individuelles. Cela permet une plus grande flexibilité (exemple : traiter les pages mémoire RO et RW de manière différente pour atténuer la dégradation des performances, ne protéger que ce qui doit l'être...). Le SSM crée des politiques de sécurité (Security Policy ou SP). A chaque page de mémoire physique, il associe une SP et le HSM applique cette politique à tous les accès qui concernent cette page. Cette organisation simple facilite la conception du SSM et du HSM.

Une SP contient les informations suivantes :
— un bit de validité,
— un mode d'intégrité (NONE, MAC-Set ou MAC-Tree),
— un mode de confidentialité (NONE, CTR, CBC),
— des clés cryptographiques : une clé secrète pour les modes de confidentialité et une deuxième clé secrète pour les calculs des MAC utilisés pour l'intégrité.

4.1.3.2 Paramètres de sécurité de page

La seconde structure de données mise en place par le SSM fait le lien entre une page mémoire et la politique de sécurité à appliquer. Il s'agit de la PSPE (*Page Security Parameter Entry*). De la même manière qu'il existe une entrée par page dans une table de pages pour que la MMU puisse effectuer une traduction d'une adresse virtuelle en une adresse physique, il existe une PSPE par page mémoire pour que le HSM puisse appliquer aux accès à cette page les primitives cryptographiques nécessaires. Il existe deux formats de PSPE : maître et esclave.

Une PSPE maître contient les informations suivantes :
— un bit de validité,
— un indicateur de taille de page.
— un bit indiquant si la page est protégée en confidentialité ou en intégrité, redondant avec la SP associée et présent uniquement pour des raisons de performance,
— un index de SP,
— l'adresse d'un ensemble de IV utilisée uniquement si la page est protégée en mode CBC et que les IV sont activés,
— l'adresse d'un ensemble de MAC ou d'arbre de MAC utilisée uniquement si la page est protégée en intégrité.

L'indicateur de de taille de page permet d'associer une PSPE à une taille spécifique de page mémoire. Dans l'exemple d'un système à base d'ARM avec quatre tailles de pages, 4 Kio, 64 Kio, 1 Mio et 16 Mio, une PSPE correspondant à une page alignée sur une frontière de 16 Mio peut être celle d'une :
— page de 4 Kio (la page de 16 Mio est divisée en 4096 pages de 4 Kio),
— page de 64 Kio (la page de 16 Mio est divisée en 256 pages de 64 Kio),
— page de 1 Mio (la page de 16 Mio est divisée en 16 pages de 1 Mio),
— page de 16 Mio (la page de 16 Mio existe).

La recherche dans les tables de PSPE commence par trouver la première PSPE de la page possédant la taille la plus grande et dans laquelle l'adresse cible est inclue. Une fois cette PSPE récupérée, on vérifie si la taille de celle-ci couvre l'adresse cible. Si cette PSPE couvre bien l'adresse cible, la recherche est terminée. Sinon, on cherche la PSPE de la page possédant la prochaine taille la plus grande. Le processus de recherche continue jusqu'à trouver une PSPE valide et qui couvre l'adresse cible.

Dans le cas où l'index SP indique une protection d'intégrité, le champ adresse contient l'adresse d'un ensemble de MAC dans une page MS ou l'adresse d'un arbre de MAC dans une page MT.

Dans la version actuelle de SecBus, une PSPE esclave contient les informations suivantes :
— un condensé cryptographique (la racine d'un arbre de MAC),
— un indicateur de taille de page.

A chaque page MT correspond une PSPE esclave. Elle est utilisée pour stocker le condensé calculé sur les racines des arbres de la page MT. C'est, en quelque sorte, la racine des racines. Le processus de recherche de PSPE esclave est exactement le même que pour une PSPE maître.

4.1.3.3 Arbre de MAC Maître (MMT)

Les deux structures de données utilisées par SecBus, SP et PSPE, sont stockées dans une zone dédiée dans la mémoire externe, nommée *Master Block* (MB), et protégée en intégrité par un arbre de MAC Maître (MMT, *Master MAC Tree*).

La figure 4.6 présente l'organisation de la mémoire. Les pages RW sont allouées avec une protection d'intégrité et sont liées aux pages MT grâce aux PSPE maîtres. La racine des arbres de MAC de la même page MT est stockée dans la PSPE esclave associée, dans la zone de PSPE du MB. Les pages RO sont aussi allouées et protégées en intégrité par les ensembles de MAC qui sont stockées dans les pages MS.

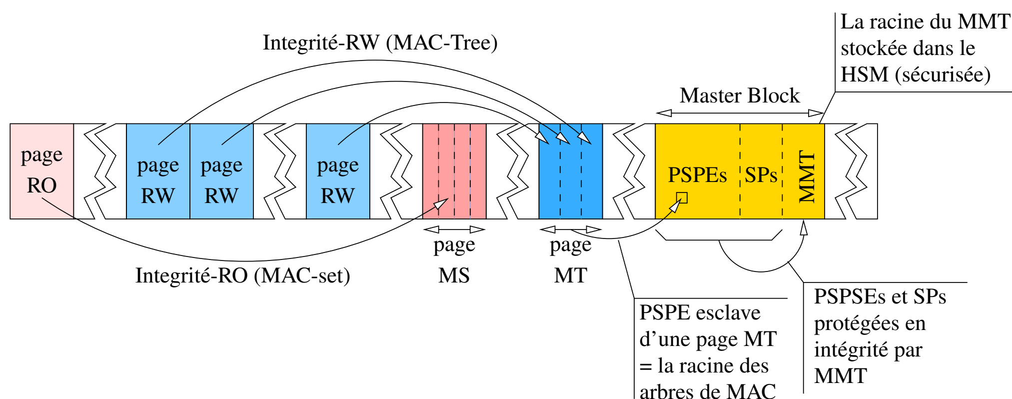


FIGURE 4.6 – Présentation de la mémoire externe avec les pages d'intégrité et le MB

4.2 Architecture logicielle

Dans cette section, nous introduirons l'architecture logicielle de SecBus qui contient deux composants importants : le bootloader et le SSM. Le rôle du premier composant est de garantir le chargement authentique et intègre du SSM qui interagit par la suite avec les applications et configure les paramètres de sécurité appliqués par le HSM.

4.2.1 Bootloader

Au démarrage du système, le bootloader est le premier composant logiciel à être lancé. Son rôle est critique pour garantir la sécurité du système. Il initialise le HSM et démarre le noyau (et le SSM) après l'avoir chargé et vérifié. Le bootloader s'assure de l'authenticité du SSM pour éviter qu'un adversaire ne le remplace par un autre composant logiciel qui pourrait, par exemple, configurer le HSM pour n'appliquer aucune protection et donc menacer la sécurité. Comme au démarrage le HSM n'est pas configuré et ne peut donc pas protéger l'exécution du bootloader, ce dernier doit être exécuté entièrement à l'intérieur du SoC (chargé depuis une mémoire non volatile interne et exécuté dans une RAM interne).

Le bootloader commence par initialiser le MB, en marquant toutes les SP et PSPE comme non valides et par ordonner au HSM de construire le MMT. Ensuite, le bootloader prépare les pages mémoire externes où le noyau doit être chargé et configure le HSM pour les protéger, au moins en intégrité.

Le bootloader peut dès lors charger le noyau et le SSM à partir d'un stockage de masse non volatile (flash, ROM, disque dur, réseau, etc.) vers la mémoire externe et calculer à la volée une signature du noyau. Le calcul de cette signature peut s'appuyer sur des accélérateurs cryptographiques exportées par le HSM ou être réalisé intégralement en logiciel (toujours en mémoire interne). La signature est utilisée pour vérifier l'authenticité du noyau et du SSM. Elle doit également permettre la mise à jour du noyau (par exemple pour corriger une faille dans le SSM) sans qu'un adversaire ne puisse utiliser l'ancienne version du noyau (*downgrade*). Pour cela, une signature de référence est stockée dans une mémoire non volatile [2] à l'intérieur du SoC pour permettre la comparaison. Cette signature est mise à jour à chaque fois le noyau est mis à jour. Une fois que le noyau est vérifié et chargé dans la zone protégée dans la mémoire externe, le bootloader lui passe le contrôle.

La procédure de démarrage assure que l'adversaire ne peut pas charger une version modifiée du SSM (grâce à la comparaison de la signature) ou la falsifier durant le démarrage (le noyau et le SSM sont stockés dans une zone mémoire protégée en intégrité par le HSM).

4.2.2 SSM

Le principal rôle du SSM est de configurer le HSM en fonction de la politique de sécurité demandée par le système d'exploitation et les applications. Le SSM s'intègre et interagit avec les autres composants du noyau du système d'exploitation et notamment le gestionnaire mémoire. La figure 4.7 présente la pile logicielle d'un système SecBus.

4.2.2.1 Interface d'applications

Grâce à la flexibilité de l'architecture SecBus, il existe de nombreuses possibilités d'intégration dans un système. En particulier, la définition des politiques de sécurité à appliquer aux applications logicielles peut prendre de très nombreuses formes. Dans le cas d'une application qui a été conçue sans prendre en compte l'architecture de SecBus, toutes ses pages mémoire peuvent, par exemple, être protégées par une SP par défaut. Cette SP est configurée par le SSM suivant les besoins spécifiques du système. On peut aussi définir plusieurs SP par défaut pour les différents types de sections (code, données constantes, données, pile, tas...)

Les applications dont la conception n'a pas tenu compte de l'architecture de SecBus peuvent aussi être spécialisées *a posteriori* lors de leur installation sur une plateforme équipée de SecBus. La plupart des formats binaires dans lesquels sont distribuées les applications logicielles permettent l'ajout de sections « utilisateur » dont la destination est laissée libre. C'est le cas du format ELF très largement utilisé dans le monde GNU/Linux et sur un grand nombre d'autres

2. D'autres solutions existent, comme des compteurs de mise à jour à base de e-Fuses ou le recours à un tiers de confiance externe (carte à puce, serveur distant...)

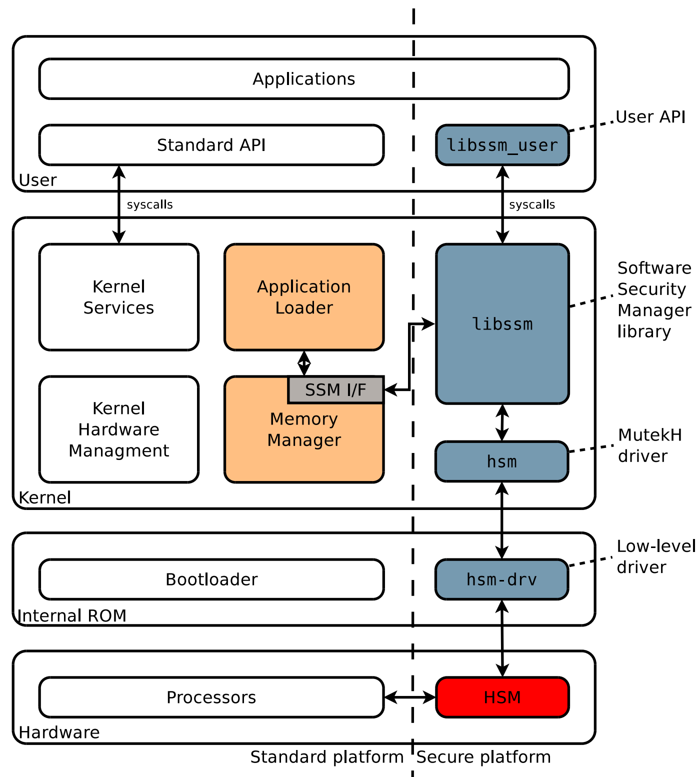


FIGURE 4.7 – Architecture logicielle de SecBus

familles de systèmes d'exploitation. Il est donc possible d'ajouter une section SecBus au format binaire ELF d'une application et d'y indiquer quelles sont les politiques de sécurité à appliquer aux autres sections.

Enfin, pour les applications conçues pour tirer parti d'une plateforme équipée de SecBus, le système d'exploitation peut offrir un appel système permettant à l'application de choisir une politique lors d'une allocation mémoire. À la réception de cet appel système, le SSM configure le HSM pour appliquer la SP reçue aux nouvelles pages allouées.

4.2.2.2 Chargeur d'applications

Au moment de chargement d'une application, ses différents segments de mémoire sont chargés dans la mémoire externe. Le concepteur d'application peut ajouter dans l'exécutable la politique de sécurité à appliquer pour chaque segment. Dans le contexte de SecBus, trois types d'applications sont distinguées :

— Applications ignorant l'existence de SecBus.
— Applications avec des conditions de sécurité statiques : le fournisseur connaît quelle politique de sécurité doit être appliquée à chaque segment de mémoire. Les conditions de sécurité sont ajoutées après la conception.
— Applications conscientes de l'existence de SecBus, avec des politiques de sécurité dynamiques.

4.2.2.3 Mémoire dynamique

Dans le contexte de SecBus, le concepteur logiciel peut choisir une différente SP pour chaque bloc mémoire alloué. Durant la phase d'exécution, un programme utilise une certaine quantité de la mémoire. Dans la plupart des cas, il s'appuie sur une pile (*stack*, par exemple utilisée pour les appels de fonctions ou quand le nombre de registres est dépassé) mais aussi sur un tas (*heap*), pour les allocations de mémoire dynamiques.

Comme la pile est créée avant l'exécution de l'application, elle doit être protégée durant le chargement de l'application. Chaque pile allouée pour une création de thread peut être protégée avec la SP sélectionnée par le concepteur logiciel.

Le concepteur logiciel peut sélectionner une politique de sécurité par défaut à appliquer pour chaque bloc mémoire alloué durant la vie d'exécution du programme. Comme chaque thread peut manipuler différents types de données, il faut aussi appliquer une politique de sécurité par défaut pour chaque thread entier. Pour certaines données, les besoins de sécurité sont différents, le concepteur logiciel doit être capable de sélectionner une politique de sécurité différente pour chaque bloc de mémoire alloué.

4.3 Architecture matérielle

Le HSM constitue la partie matérielle de l'architecture de SecBus. Ce dernier est localisé à l'intérieur du SoC entre le bus d'interconnexion et le contrôleur mémoire. Il intercepte de manière transparente les accès à la mémoire externe et, suivant les paramètres de sécurité (SP et PSPE) configurés par le SSM, il effectue les opérations cryptographiques nécessaires : chiffrement et déchiffrement des données, vérification d'intégrité et calcul des condensés cryptographiques. Ces opérations ajoutent une latence supplémentaire à chaque accès mémoire. Par exemple, pour une lecture d'une donnée chiffrée, un déchiffrement est nécessaire avant d'envoyer la donnée lue, en clair, au CPU. Cette dégradation de performance est limitée aux zones mémoire protégées.

La figure 4.8 présente les différents modules de l'architecture interne du HSM. Ces modules se divisent en trois catégories différentes :

- Modules d'interface : ils traitent les requêtes de lecture/écriture venant du bus d'interconnexion interne, vérifient si une protection est nécessaire et retournent les réponses des requêtes.
  - *Split Module* (SM)
  - *Merge Module* (MM)
  - *Input Controller* (IC)
  - *IO Registers Module* (IORM)
- Modules de protection : ils gèrent et effectuent les opérations cryptographiques.
  - *Security Context Controller* (SSC)
  - *Security Controller* (SC)
  - *Crypto Engine* (CE)
  - *MAC Set Controller* (MSC)

— *MAC Tree Controller* (MTC)

- Modules divers : caches internes, registres, FIFO, multiplexeurs...

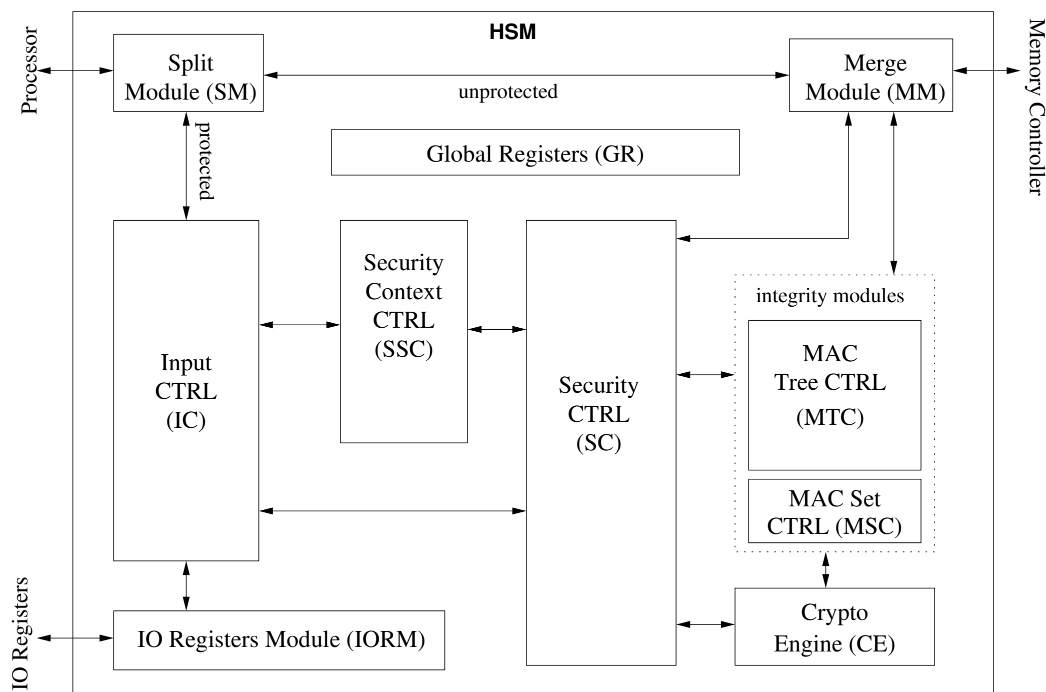


FIGURE 4.8 – Architecture interne du HSM

4.3.1 Modules d'interface

4.3.1.1 Split Module

Le *Split Module* joue le rôle d'un démultiplexeur. Il reçoit les requêtes de lecture/écriture à travers son interface cible et vérifie s'ils concernent la zone mémoire protégée. Dans ce cas, le *Split Module* achemine cette requête à l'*Input Controller* pour appliquer la protection requise. Autrement, il achemine la requête au module *Merge Module* pour des accès mémoire sans protection. La vérification est basée sur l'adresse de base et la taille de la zone protégée.

4.3.1.2 Merge Module

Le *Merge Module* joue le rôle d'un multiplexeur. Il reçoit les requêtes de lecture/écriture à partir du *Split Module* ou des modules de protection (*Security Controller* ou les modules d'intégrité) et achemine l'interface de sortie vers le contrôleur mémoire. Il achemine aussi les requêtes de réponse vers les modules correspondants.

4.3.1.3 Input Controller

L'*Input Controller* gère les requêtes de lecture/écriture venant directement (à travers le *Split Module*) ou indirectement (à travers le *IO Registers Module*) du processeur. Dans le cas où les requêtes sont venues directement du processeur (qui ne concernent jamais le MB), l'*Input Controller* demande au *Security Context Controller* de préparer le contexte de sécurité (SP et PSPE) et de le stocker dans un registre interne. Pour les commandes venues des registres d'IOs, l'*Input Controller* effectue une demande au *Security Context Controller* pour la création ou modification du PSPE et du SP ou pour l'initialisation d'arbre.

4.3.1.4 IO Registers Module

Ce module gère les registres d'IOs et envoie, aux modules correspondants, les commandes suivantes :

- Les adresses de base de la zone mémoire protégée (et sa taille) et du MB (PSPE, SP et MMT).
- La mise à jour du PSPE et du SP.
- L'initialisation de l'arbre.
- L'activation des cache (du *Security Context Controller*, *MAC Tree Controller* et *MAC Set Controller*).

4.3.2 Modules de protection

4.3.2.1 Security Context Controller

Le *Security Context Controller* prend en charge la gestion des contextes de sécurité (PSPE et SP) de chaque page mémoire. Lorsque le contexte de sécurité est demandé par l'*Input Controller* ou le *MAC Tree Controller*, le *Security Context Controller* parcourt (en se basant sur le champ d'adresse reçue) la table des PSPE afin de trouver la PSPE adéquate et la SP correspondante. Pour les accès à la mémoire externe (pour lire ou écrire une PSPE ou une SP), le *Security Controller* est responsable et les primitives cryptographiques nécessaires sont appliquées.

Le *Security Context Controller* utilise deux caches pour stocker quelques PSPE et SP et qui sont les premiers consultés avant d'aller voir dans la mémoire externe à travers le *Security Controller*. Les caches sont facultatifs et configurables.

4.3.2.2 Security Controller

Le *Security Controller* gère les accès de lecture/écriture dans la zone protégée de la mémoire externe, dont le MB. Le *Security Controller* reçoit les requêtes de lecture/écriture venant de l'*Input Controller* et du *Security Context Controller*. Il cherche la SP correspondante à partir du *Security Context Controller* ou des registres de configuration (suivant le type d'accès). À partir de la SP trouvée, les modes de protection d'intégrité et de confidentialité sont connus.

Le *Security Controller* demande au *Crypto Engine* d'effectuer les opérations de chiffrement et déchiffrement, au *MAC Tree Controller* ou au *MAC Set Controller* pour effectuer la vérification d'intégrité et la mise à jour.

4.3.2.3 Crypto Engine

Ce module contient deux sous-modules, un pour la confidentialité et l'autre pour l'intégrité. Le premier sous-module chiffre et déchiffre des blocs de données en utilisant l'algorithme de chiffrement par bloc choisi (par exemple DES-X) en mode compteur (pour les données en lecture seule) ou en mode CBC (pour les données en lecture et écriture). Le deuxième sous-module qui est responsable de l'intégrité est chargé de calculer les MAC des groupes de données en utilisant le mode CBC-MAC. Ces MAC, sont utilisés pour vérifier l'intégrité. Ils sont organisés en ensembles de MAC (pour les données en lecture seule) ou en arbres de Merkle de MAC (pour les données de lecture et écriture).

4.3.2.4 MAC Set Controller

Le *MAC Set Controller* gère les ensembles de MAC utilisés pour vérifier l'intégrité des pages mémoire RO. Il utilise un cache local pour stocker les MAC les plus fréquemment accédés, ce qui réduit le nombre d'accès à la mémoire externe. Les calculs de MAC pour la vérification d'intégrité ou pour la mise à jour sont effectués par le *Crypto Engine*. Les deux opérations faites par le *MAC Set Controller* sont demandées par le *Security Controller*. Dans le cas de la vérification d'intégrité, le *MAC Set Controller* compare le MAC généré à partir des blocs de données et le MAC de référence (que ce soit dans le cache ou dans la mémoire externe). Si la comparaison échoue, une interruption est levée. Dans le cas d'une mise à jour, après le calcul du MAC, le module envoie une requête d'écriture au *Merge Controller* pour écrire ce MAC dans la mémoire externe.

4.3.2.5 MAC Tree Controller

Le *MAC Tree Controller* gère les arbres de MAC utilisés pour vérifier l'intégrité des pages RW. Il utilise aussi un cache local pour stocker les nœuds les plus fréquemment utilisés, ce qui permet de réduire de manière significative le nombre d'accès à la mémoire externe et le nombre de calculs de MAC. Les calculs de MAC pour les différentes opérations de ce module sont effectués par le *Crypto Engine*. Le *MAC Tree Controller* réalise trois opérations : l'initialisation d'un arbre, la vérification d'intégrité lors de la lecture d'un bloc de données et la mise à jour lors de l'écriture d'un bloc de données.

L'opération d'initialisation est effectuée à chaque fois qu'une nouvelle page mémoire RW est réservée avec une protection en intégrité. Nous avons décrit en détail cette opération dans la section 2.3 (voir page 46) et, dans la section 3.1 (voir page 73) et la section 3.2 (voir page 76) nous avons proposé une nouvelle technique pour accélérer l'opération d'initialisation.

La vérification d'intégrité s'appuie sur l'algorithme ASAP : le processus de vérification

s'arrête une fois qu'il trouve le premier nœud intermédiaire dans le cache. Ce processus commence à la demande du *Security Controller* pour vérifier un bloc de donnée. Il lit le groupe de données qui contient le bloc, calcule son MAC et le compare avec le nœud parent. Si le nœud parent existe dans le cache, le processus termine à ce niveau. Sinon, il continue la vérification en montant dans la branche jusqu'à ce qu'il trouve un nœud sûr (dans le cache ou le nœud racine).

L'opération de mise à jour est plus compliquée que les autres opérations parce qu'elle traite le cas d'éviction d'un nœud victime du cache (quand le seuil d'un ensemble de cache est atteint) et sa mise à jour dans la mémoire externe, afin d'implémenter correctement l'algorithme ALAP et, donc de garder la cohérence de entre le cache et la mémoire externe. La section 3.3 (voir page 81) décrit de manière détaillée le processus d'éviction.

Le *MAC Tree Controller* accèdent aux fonctions du cache (`READ`, `WRITE`, `RESTORE`, `SYNCHRONIZE` et `XREAD`), qui sont utilisées dans les algorithmes d'ASAP et d'ALAP, par l'intermédiaire du *MAC Tree Cache Controller*.

4.4 Conclusion

Dans ce chapitre nous avons présenté le projet SecBus dont l'objectif est de protéger la confidentialité et l'intégrité du contenu de la mémoire externe contre des attaquants ayant tout contrôle sur le matériel, à l'exception du SoC lui même, et sur les logiciels. Nous avons commencé par décrire le modèle de menace et les objectifs de protection. Ensuite, nous avons présenté l'architecture logicielle et matérielle de SecBus. Dans le chapitre suivant, nous aborderons les différentes simulations effectuées autour de cette architecture en intégrant les différentes méthodes des arbres de Merkle.

Chapitre 5

Simulations et validations

Sommaire

5.1 Tests de simulation des Arbres de Merkle 120
5.1.1 Environnement de simulation 120
5.1.2 Résultats 123
5.1.3 Conclusion 126
5.2 Validation fonctionnelle 126
5.2.1 Introduction 126
5.2.2 Ce que l'on prouve 126
5.2.3 Quelques précisions sur la preuve de raffinement avec la méthode $\mathcal{B}$. 129
5.2.4 Quelques précisions sur la preuve d'équivalence avec *EasyCrypt* . . . 136
5.2.5 Conclusion 142
5.3 Évaluation des performance du module matériel HSM 143
5.3.1 Introduction 143
5.3.2 Coûts matériels 143
5.3.3 Augmentation de l'empreinte mémoire 144
5.3.4 Configurations et résultats 145
5.3.5 Conclusion 153

Dans ce chapitre, nous présenterons les expérimentations que nous avons effectuées pour tester l'architecture matérielle de SecBus, et particulièrement les arbres de Merkle. La première section introduit l'environnement de simulation et les différents outils utilisés et, montre les résultats obtenus de la simulation du module HSM au niveau CABA, afin d'évaluer les différentes variantes d'arbres de Merkle. Ensuite, dans la deuxième section, nous présenterons les différentes preuves fonctionnelles appliquées sur les algorithmes d'arbres de Merkle réguliers et creux. Enfin, la dernière section portera sur l'évaluation des performances du module HSM en utilisant une plate-forme de prototypage.

5.1 Tests de simulation des Arbres de Merkle

5.1.1 Environnement de simulation

L'évaluation des systèmes SoC au niveau RTL (*Register Transfer Level*) demande un temps de simulation considérable, ce qui ralentit la phase de conception. La modélisation au niveau CABA *(Cycle Accurate / Bit Accurate)* est une alternative qui permet de réduire le temps de simulation de ces systèmes. Comme au niveau RTL, elle émule le comportement du système à chaque cycle en se basant sur la méthode des machines à états finis (FSM, *Finite State Machine*) avec une communication synchrone. Dans le projet SecBus, nous avons travaillé sur la plate-forme virtuelle SoCLib [53] que nous introduirons par la suite. Le choix de cette plate-forme a été motivé par le fait que SoCLib soit "*open source*" et par sa flexibilité pour intégrer des nouveaux modules.

Par la suite, nous introduirons cette plate-forme virtuelle et ensuite, nous présenterons notre plate-forme matérielle de simulation et l'application utilisée.

5.1.1.1 Introduction de SoCLib

SoCLib est une plate-forme de modélisation et de simulation des composants matériels qui permet de réaliser un simulateur d'un système SoC (ou MPSoC, un SoC multiprocesseurs) au niveau CABA. L'environnement SoCLib est basé sur le langage et le moteur de simulation SystemC [1].

Dans SoCLib, les différents modules d'une architecture matérielle communiquent entre eux en utilisant le protocole VCI (*Virtual Component Interface*). Différents modules sont disponibles de base dans SoCLib : des processeurs (Sparc, Mips, ARM, PPC, etc.), des mémoires cache, des interconnexions VCI, des mémoires RAM et d'autres modules divers (FIFO, TTY, GPIO, etc.). La norme VCI définit une communication point à point asymétrique entre un initiateur et une cible [54]. L'initiateur émet des requêtes (de lecture ou d'écriture) et la cible y répond (figure 5.1). L'interface VCI est constituée de deux paquets de signaux : les signaux de commande et les signaux pour acheminer la réponse. Ces deux paquets fonctionnent de façon asynchrone.

Le fonctionnement du protocole VCI peut être décrit comme suit [54] : un initiateur informe une cible que des données valides sont prêtes à être émises (`CMD`) en envoyant un signal `CMDVAL`. Les données à envoyer peuvent être mises dans un ou plusieurs paquets et chaque paquet peut contenir un ou plusieurs mots. La cible, à son tour, envoie le signal `CMDACK` à l'initiateur pour l'informer de son état (libre ou occupée). Le cycle de transfert commence si la cible est libre. Le nombre de cycles nécessaires pour terminer la transaction dépend du nombre de mots dans le paquet de données. Lorsqu'une requête de réponse est prête, la cible envoie un signal `RSPVAL` à l'initiateur qui répond par le signal `RSPACK` pour indiquer son état. Si l'initiateur est libre, la cible commence à transférer les données de réponse (`RSP`).

1. SystemC est un langage de modélisation de système qui a été conçu pour résoudre le problème de la modélisation de systèmes mixtes matériel et logiciel, tout en offrant les avantages du langage C++.

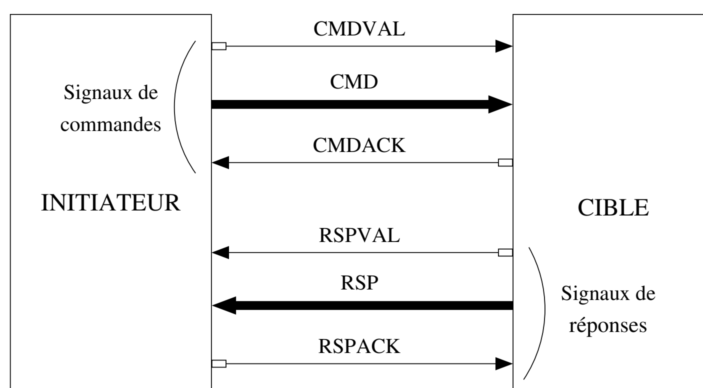


FIGURE 5.1 – Principe d'un protocole VCI

Pour ce qui concerne le logiciel embarqué, si on part d'une application utilisateur écrite en langage C, il faut utiliser un cross compilateur spécifique pour le processeur choisi. Le résultat de l'exécution est un fichier binaire au format ELF (*Executable and Linkable Format*). Le code binaire correspondant doit ensuite être chargé dans les mémoires embarquées du SoC.

5.1.1.2 Plate-forme et application

La plate-forme matérielle simulée est constituée d'un SoC et d'une mémoire externe, qui communiquent entre eux à travers le bus mémoire externe (figure 5.2). Le SoC est basé sur un processeur MIPS. La plate-forme SoCLib dispose d'un simulateur de jeu d'instructions (ISS, *Instruction Set Simulator*) qui simule l'exécution des jeux d'instructions sur le MIPS32. Le processeur est instancié avec le bus système interne comme un paramètre template d'un wrapper (XcacheWrapper). Le module matérielle HSM est localisé à l'intérieur du SoC entre le bus système et le contrôleur mémoire. Les interconnexions utilisées sont de type VGMN (*VCI Generic Micro-Network*).

5.1.1.2.1 Initialisation des arbres de Merkle Dans la gestion des arbres de Merkle, l'opération d'initialisation est réalisée de manière différente pour chaque variante d'arbre. Pour les AMR, cette opération est effectuée par une machine à états finis, à l'intérieur du module HSM. Les AMC-I sont initialisés à l'aide d'un module DMA (*Direct Memory Access*) localisé à l'extérieur du SoC et connecté au bus mémoire. Enfin l'initialisation des AMC-NI est triviale. Elle consiste simplement à initialiser la racine par la valeur `NULL`.

5.1.1.2.2 Configuration des paramètres de sécurité La plate-forme matérielle exécute une petite pile logicielle (qui joue le rôle du SSM) qui décrit l'application utilisateur et configure les paramètres de sécurité nécessaires. Trois paramètres sont initialisés lors de la phase de configuration : l'adresse et la taille de la zone mémoire à protéger, et l'adresse du MB. Ensuite, les SP sont initialisées (clés, primitives cryptographiques, etc.). Dans notre exemple de tests des arbres, nous choisirons les arbres de Merkle pour la protection d'intégrité (MAC-Tree) et aucune primitive pour la confidentialité (NONE). Enfin, les PSPE maîtres sont initialisés pour

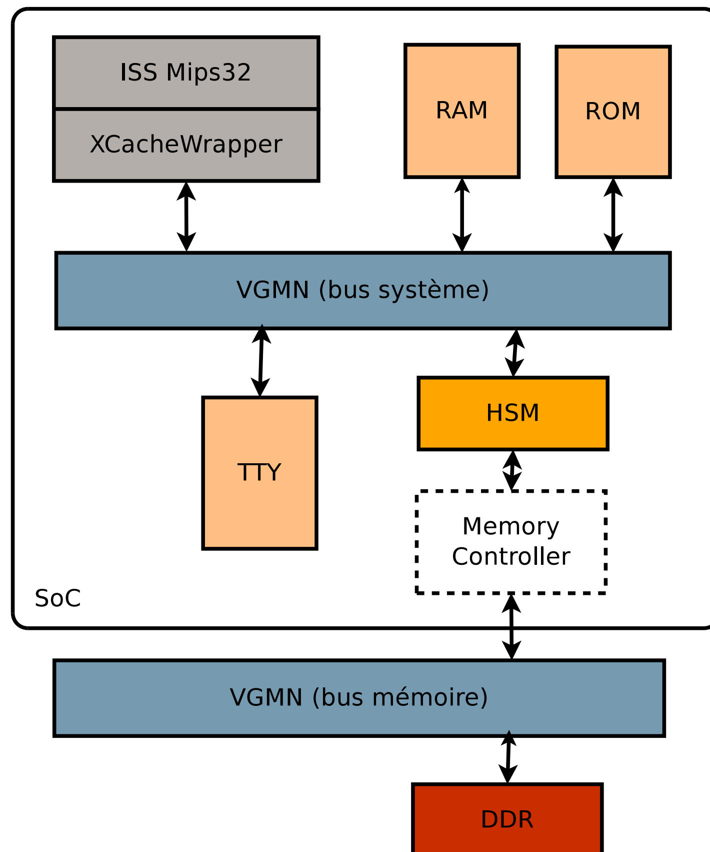


FIGURE 5.2 – Plate-Forme matérielle de SoCLib avec le HSM

lier chaque page mémoire créée avec une SP, et les PSPE esclaves pour contenir les nœuds racines des pages MT dans le cas des arbres de Merkle.

5.1.1.2.3 Application L'application utilisateur choisie pour les estimations de performances est simple. Elle écrit des données de manière aléatoire dans 12 pages mémoires différentes. Le but est de peupler au maximum le cache des arbres de Merkle afin de vérifier son impact sur la gestion des arbres de Merkle. L'arité de l'arbre de Merkle est fixée à 4. Donc, pour protéger les 12 pages mémoires, nous avons besoin de 4 pages MT additionnelles pour stocker les nœuds des arbres (trois arbres peuvent être stockés dans une page MT).

5.1.1.2.4 Paramètres de la simulation Les tests de simulation utilisent les paramètres suivants :

- latence du bus système : 1 cycle d'horloge CPU
- latence du bus mémoire : 3 ou 100 cycles d'horloge CPU
- fonction de compression à sens unique : CBC-MAC basé sur DES-X
- latence de l'algorithme DES-X : 4 cycles d'horloges CPU
- taille d'une page mémoire à protéger : 4 kio
- taille d'une page MT : 4 kio
- nombre des écritures aléatoires : 12.000
- cache MT : *set-associative*, 64 ensembles, 8 blocs, 8 octets par bloc, LRU, write-back

5.1.2 Résultats

Les résultats des simulations sont séparés en deux parties : l'étape d'initialisation et les écritures aléatoires. Le but est de montrer la différence entre les trois variantes (AMR, AMC-I et AMC-NI) dans ces deux phases. De plus, les simulations sont effectuées avec et sans cache pour montrer l'influence de ce dernier sur la gestion des arbres de Merkle.

Les tableaux 5.1 et 5.3 présentent les résultats des simulations, respectivement sans et avec utilisation d'un cache. Pour les trois variantes d'arbres de Merkle, nous avons mesuré trois paramètres :

- cc : le nombre de cycles d'horloge CPU (pour une latence mémoire de 3 et 100 cycles),
- Acc : le nombre d'accès à la mémoire externe,
- MAC : le nombre de calculs du MAC.

Avant de montrer les résultats des simulations avec utilisation d'un cache, nous présentons dans le tableau 5.2 les mesures effectuées sur l'impact du seuil sur les performances du cache à partir des résultats obtenus, nous choisirons le seuil de 70 %, vu qu'il possède le nombre d'accès au cache le plus petit, et avec un taux de `miss` raisonnable.

À partir de ces résultats, nous remarquons que l'utilisation du cache n'a pas d'impact durant l'opération d'initialisation. Son influence est très significative après l'initialisation des arbres de Merkle, durant les opérations de vérification d'intégrité et de mise à jour (au moment des écritures aléatoires). Par exemple, dans le cas des AMR, les simulations sont 6,5 fois plus rapide avec l'utilisation du cache.

TABLE 5.1 – Résultats de la simulation sans cache

Variante	Sans cache			
	cc(3)	cc(100)	Acc.	MAC
Initialisation des 12 arbres de Merkle				
AMR	613.150	4.231.843	29.311	10.885
AMC-I	50.189	377.748	2.106	378
AMC-NI	19.822	119.726	875	393
12.000 écritures aléatoires				
AMR	14.771.897	102.515.709	743.999	432.000
AMC-I	14.771.897	102.515.709	743.999	432.000
AMC-NI	14.819.747	103.006.599	748.559	432.354

TABLE 5.2 – Impactes du seuil sur les performances du cache

	Total		WRITE		READ		RESTORE		SYNC	
	Acc	miss	Acc	miss	Acc	miss	Acc	miss	Acc	miss
10 %	829.619	0,028	130.440	0,014	564.771	0,017	14.228	0,825	120.180	0
30 %	172.551	0,111	39.139	0,008	97.344	0,079	14.683	0,758	21.385	0
50 %	107.265	0,211	27.130	0,016	47.251	0,213	47.251	0,574	11.718	0
70 %	106.958	0,231	26.308	0,028	44.672	0,240	25.093	0,529	10.885	0
90 %	112.008	0,287	25.568	0,054	43.511	0,285	32.978	0,557	9.951	0

TABLE 5.3 – Résultats de la simulation avec cache

Variante	Avec cache			
	cc(3)	cc(100)	Acc.	MAC
Initialisation des 12 arbres de Merkle				
AMR	597.550	4.110.631	28.580	10.332
AMC-I	41.307	308.644	1.523	47
AMC-NI	8.244	22.086	199	72
12.000 écritures aléatoires				
AMR	2.992.027	16.000.585	162.815	37.169
AMC-I	2.984.689	15.941.367	162.620	37.151
AMC-NI	3.018.228	16.261.442	165.766	38.393

TABLE 5.4 – Résultats de la simulation sans cache

	AMR	AMC-I	AMC-NI
Nombre d'accès au cache			
Total	106.958	88.870	89.126
WRITE	26.308	16.030	18.191
READ	44.672	36.932	24.300
RESTORE	25.093	24.999	23.079
SYNCHRONIZE	10.885	10.909	10.313
XREAD	-	-	13.243
Nombre de cache miss			
Total	24.753	24.802	28.403
WRITE	735	799	5.079
miss READ	10.743	10.754	6.815
RESTORE	13.275	13.249	11.404
XREAD	-	-	5.105
Taux de miss			
Total	0.231	0.279	0.319
WRITE	0.027	0.050	0.279
READ	0.240	0.291	0.280
RESTORE	0.529	0.530	0.494
XREAD	-	-	0.385

Les AMR nécessitent beaucoup de temps pour être initialisés. Par contre, l'initialisation des AMC est beaucoup plus rapide. L'initialisation des AMC-I est 13,5 fois plus rapide et celle des AMC-NI est 186 fois plus rapide que celle des AMR.

Les calculs de MAC effectués pendant l'étape d'initialisation dans les variantes AMC-I et AMC-NI sont dus à l'initialisation des paramètres de sécurité du *master block* (1 SP, 12 PSPE maîtres et 4 PSPE esclaves) qui entraîne une mise à jour du *master MAC tree*.

Après les 12.000 écritures aléatoires, les tests de simulation montrent que les AMR et AMC-I ont presque les mêmes résultats, que ce soit avec ou sans utilisation du cache. Par contre, les AMC-NI dégradent un peu les performances. En effet, pour cette variante, les algorithmes de lecture et d'écriture vérifiées sont un peu plus complexes car une branche accédées pour la première fois contient des valeurs aléatoires (non initialisées) et il faut les distinguer de valeurs modifiées volontairement par l'adversaire (voir section 3.2).

Le tableau 5.4 montre les statistiques d'utilisation du cache MT, pour les trois variantes d'arbres de Merkle, obtenues à partir des simulations. Pour les différentes opérations utilisées par le cache, nous avons mesuré le nombre d'accès au cache, le nombre de cache `miss` et le taux de `miss`.

Les résultats obtenus dans ce tableau montrent que le nombre d'accès au cache est plus élevé pour les AMR par rapport AMC. Cette augmentation est due à l'utilisation du cache pendant l'opération d'initialisation de l'arbre, et particulièrement les opérations `READ` et `WRITE` (les fonctions `RESTORE` et `SYNCHRONIZE` ne sont pas utilisées pendant cette opération). Par

contre, le nombre de cache de `miss` des AMR est, plus au moins, égale à celui de AMC-I et inférieur à celui de AMC-NI. Ce paramètre est obtenu pendant les écritures aléatoires dans les pages mémoires (c'est-à-dire après l'opération d'initialisation). Enfin, le taux de `miss` du AMR est le plus petit par rapport aux autres méthodes car son nombre d'accès au cache est supérieur et son nombre de cache `miss` est inférieur aux autres méthodes.

5.1.3 Conclusion

Comme ce que nous avons déjà vu précédemment dans l'évaluation théorique des performances des différentes variantes des arbres de Merkle, les résultats de simulations effectuées dans cette section montrent que l'utilisation des arbres de Merkle creux accélèrent nettement la phase d'initialisation par rapport aux arbres de Merkle réguliers. La variante AMC-I permet d'accélérer la phase d'initialisation sans effets indésirables ultérieurs sur les opérations de vérification ou de mise à jour. Pour la variante AMC-NI, la phase d'initialisation est beaucoup plus rapide (presque instantanée) par rapport aux AMR ou aux AMC-I. Par contre, les accès de lecture et d'écriture dans la mémoire externe, après l'initialisation, prennent plus de temps et nécessitent plus de calculs de MAC, le temps que l'arbre soit totalement initialisé.

5.2 Validation fonctionnelle

5.2.1 Introduction

SecBus est une architecture qui permet de protéger la communication entre un SoC et la mémoire au moyen d'algorithmes appliqués au niveau de la puce lors de chaque échange de données : du chiffrement pour la confidentialité et du hachage pour l'intégrité. Valider solidement cette technologie serait bien sûr bienvenu pour renforcer la garantie de sécurité qu'elle apporte. Cela implique de valider un certain nombre d'aspects complémentaires. Par ailleurs, il existe plusieurs façons de valider. La garantie la plus forte est celle de la preuve mais la contrepartie de cette force est que la preuve est généralement chère : rarement automatique, souvent longue, difficile, ou même inabordable en raison de sa complexité.

L'approche par preuve a été appliquée à SecBus sur certains aspects complémentaires en utilisant deux outils différents (l'atelier $\mathcal{B}$, prouveur de la méthode $\mathcal{B}$, et *EasyCrypt*). Dans cette section nous présentons ce travail en situant tout d'abord ce qui a été prouvé par rapport à SecBus, puis en donnant un rapide aperçu des outils et des preuves elle-même.

5.2.2 Ce que l'on prouve

Ce que l'on prouve concerne l'intégrité des données. En résumé (les détails sont exposés dans la suite), pour garantir l'intégrité d'une zone, SecBus maintient un `hash` de cette zone sous forme d'arbre de Merkle qui est mis à jour quand on écrit dans cette zone et lu pour vérification chaque fois que l'on lit dans cette zone. C'est ce qu'illustre la figure 5.3.

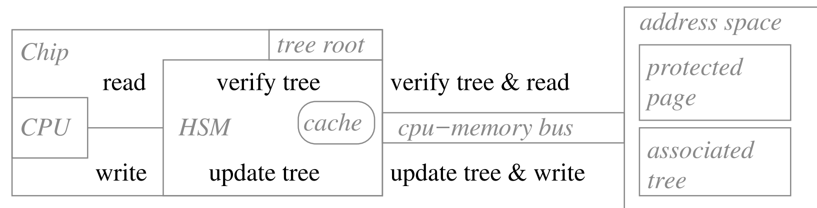


FIGURE 5.3 – Protection d'intégrité par des arbres de Merkle

Seule la racine de l'arbre est sauvegardée à l'intérieur de la puce. Les autres nœuds sont en mémoire. Ceci est coûteux : vérifier et mettre à jour les arbres engendre beaucoup de communications sur le bus. Aussi SecBus est doté d'un cache sur la puce pour les nœuds, avec une gestion assez subtile. Une autre optimisation consiste à ne parcourir que les nœuds utiles des arbres lors des diverses opérations (les ascendants et "oncles" des cellules mémoire accédées). Enfin on évite l'initialisation des arbres en utilisant des arbres creux. De tout ceci il découle qu'une preuve complète devrait traiter les divers points suivants :

1. *Preuve du principe de protection par les arbres de Merkle*. C'est une approche classique et il est généralement admis que cela fonctionne (à l'incertitude des collisions près). Pour une preuve complète, il faudrait le prouver.
2. *Preuve de l'optimisation par un parcours partiel* pour garantir qu'il suffit de considérer les ascendants et oncles des cellules accédées pour garantir le respect de la structure d'arbre de Merkle.
3. *Preuve des algorithmes réels de SecBus* garantissant qu'ils se comportent comme les algorithmes théoriques sur lesquels on prouve les points précédents.
4. *Preuve de la correction comportementale des arbres creux* pour assurer que l'utilisation des arbres creux est similaire à celle des arbres classiques. Il s'agit de montrer que l'on obtient les mêmes résultats sur une mémoire non corrompue (donc pas de fausse alerte) et que la seule différence porte sur l'accès à des cellules non corrompues dans une mémoire corrompue : dans ce cas, l'accès est sûr pour la donnée concernée. La corruption sur d'autres cellules que celle accédée est détectée différemment par les deux algorithmes.
5. *Preuve de la sécurité des algorithmes réels* garantissant que leur exécution peut effectivement être considérée comme atomique, c'est-à-dire qu'elle n'introduit pas d'accès multiples dangereux à la mémoire.
6. *Preuve de la transparence des caches* pour garantir que l'utilisation des caches ne compromet pas les résultats des points précédents.

Mener les preuves des points 1 et 2 sur les algorithmes finaux avec le détail des choix d'implémentation serait extrêmement complexe. Les preuves formelles nécessitent l'explicitation de tous les détails et l'abstraction est indispensable pour les rendre abordables. Il est donc opportun de mener ces preuves sur des algorithmes mathématiques abstraits, épurés. Il faut ensuite montrer que les algorithmes concrets implémentent bien les algorithmes abstraits, ce qui est l'approche usuelle par raffinement. La preuve de raffinement est l'objet du point 3.

Le point 4 peut être abordé au niveau abstrait comme concret. Une preuve manuelle serait plus simple au niveau abstrait mais en utilisant un outil permettant de vérifier automatiquement l'équivalence de programme, le niveau concret semble faisable (comme nous le verrons plus loin avec une preuve partielle). Les points 5 et 6 concernent intrinsèquement le niveau concret et doivent donc être prouvés à ce niveau. Enfin, le dernier point à ne pas oublier est que toute preuves porte sur des objets mathématiques et la correspondance entre le physique et ses modèles mathématiques doit être assurée par d'autres moyens.

Parmi toutes ces étapes d'une preuve "idéale", deux ont été abordées :

— En utilisant la méthode $\mathcal{B}$ et l'outil de preuve Atelier $\mathcal{B}$ associé, le point 3 a été traité : les arbres de Merkle ont été modélisés selon une représentation mathématique abstraite standard. Sur cette représentation ont été spécifiés des algorithmes fonctionnels pour la lecture et l'écriture dans la mémoire effectuant les mises à jour et vérifications associées sur les arbres. Il a ensuite été prouvé que la représentation concrète des arbres et les algorithmes finaux sur cette dernière constituaient bien un raffinement (c'est-à-dire une implémentation conforme) de la version abstraite.
— En utilisant l'outil *EasyCrypt*, le point 4 a été partiellement traité au niveau concret : il a été prouvé que l'algorithme sur les arbres creux avait le même comportement que celui des arbres standard sur une mémoire non corrompue.

La figure 5.4 suivante résume ces différents points. La fiabilité des caches et l'atomicité de l'exécution des algorithmes réels n'ont pas été traités.

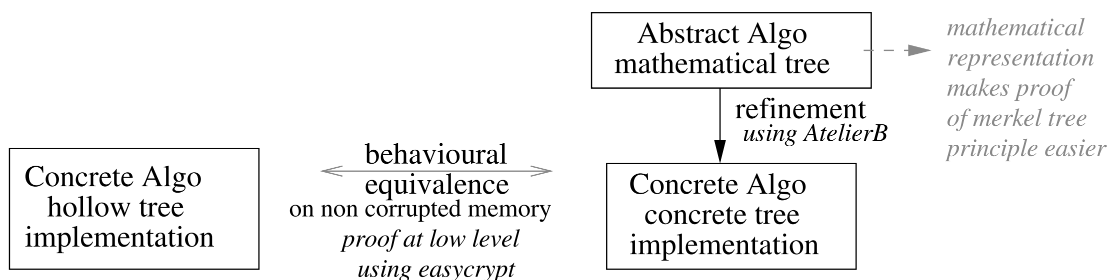


FIGURE 5.4 – Les deux étapes d'une preuve

Garantir la correspondance entre les deux niveaux d'abstraction ne suffit pas à prouver leur correction car ils pourraient être tous les deux erronés. Mais cette preuve de la relation d'implémentation entre deux représentations différentes renforce la confiance que l'on peut avoir en chacune d'entre elles. De plus l'abstraction facilite la compréhension et la vérification humaine. Elle rend abordable la preuve du point 1 (Principe de sécurité de Merkle) qui renforcerait considérablement cette confiance. Enfin, ce raffinement montre que l'implémentation est sûre (par exemple qu'il n'y a pas d'erreur de calculs d'indices) et que les preuves au niveau abstrait offrent les bonnes garanties sur le niveau concret.

Le travail avec *EasyCrypt* constituait une étude de faisabilité. La garantie apportée sur le cas "sain" est satisfaisante. Le cas où la mémoire est corrompue, quoique plus complexe, semble abordable.

5.2.3 Quelques précisions sur la preuve de raffinement avec la méthode $\mathcal{B}$

Dans cette présentation rapide, nous omettons ici les détails techniques de la méthode $\mathcal{B}$ pour donner une vue d'ensemble simplifiée de ce qui a été fait. Nous simplifions également d'autres aspects, par exemple en considérant des arbres seuls alors que le cas où il y a plusieurs racines (donc une forêt de plusieurs arbres) a été traité.

Le modèle a la forme générale suivante (figure 5.5) :

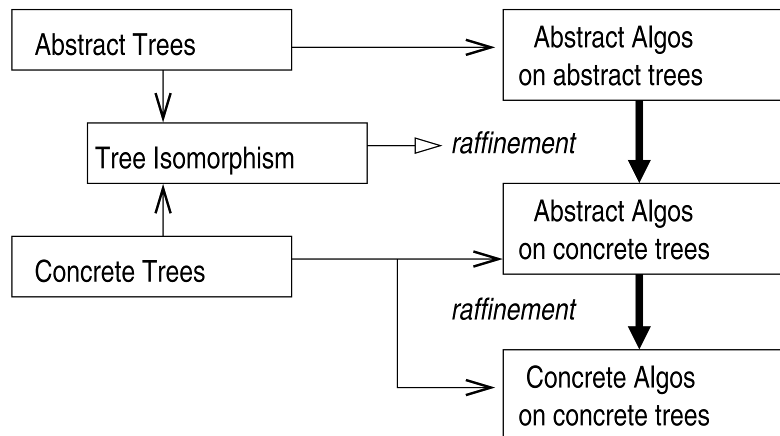


FIGURE 5.5 – Modèle d'arbres

5.2.3.1 Des arbres abstraits aux arbres concrets

Un arbre est un mapping (fonction totale) de ses nœuds vers des valeurs. La différence entre arbres abstraits et concrets est que pour premiers, les nœuds abstraits sont représentés par leur chemin depuis la racine tandis que les seconds sont représentés par un couple (h, i) où h est la hauteur du nœud tandis que i est son index dans la liste (ordonnée) des nœuds de même hauteur. Dans un cas on a une représentation mathématique classique (un chemin est une liste) et dans l'autre une représentation pratique pour un calcul plus optimisé.

L'isomorphisme montre l'équivalence des deux représentations : il consiste essentiellement en une bijection entre les nœuds des deux représentation, et on a prouvé formellement que cette bijection préserve toutes les opérations que l'on a défini sur les arbres. Les opérations définies sont pour la plupart des opérations usuelles comme "père", "fils", etc. Et elles sont toutes définies sur les deux représentations. La préservation de ces opération par l'isomorphisme assure, par exemple, qu'au père d'un nœud on associe le père de son image.

L'aperçu ci-dessous du contenu des spécifications (simplifié), transcrit en langage "usuel" montre comment cela se traduit concrètement sans rentrer dans les détails syntaxiques et techniques. On y considère des arbres N-aires (N fils par nœud interne) de hauteur H. La spécification des arbres et de quelques fonctionnalités associées est spécifique pour chaque représentation. Ensuite, la définition des autres fonctionnalités est identique pour les deux représentations.

Arbres abstraits
// spécifique à la représentation
$A_Nodes =$ *suites sur* $[1, N]$*de taille* $\leq H$
$dom(A_father) = A_Nodes$ *non vides*
$\forall ux \in dom(A_father), A_father(ux) = u$
$\ldots$
// indépendant la représentation
$A_Trees = A_Nodes \rightarrow Values$
$A_read : A_Nodes \times A_Trees \rightarrow Values$
$\forall (n, t) \in A_Nodes \times A_Trees, A_read(n, t) = t(n)$
$\ldots$

Arbres concrets
// spécifique à la représentation
$C_Nodes = \{(h, i) | h \in [0, H]\ \&\ i \in [0, N^{H-h} - 1]\}$
$dom(C_father) = \{(h, i) \in C_Nodes \mid (h + 1, i/N) \in C_Nodes\}$
$\forall (h, i) \in dom(C_father),\ C_father(h, i) = (h + 1, i/N)$
$\ldots$
// indépendant la représentation
$C_Trees = C_Nodes \rightarrow Values$
$C_read : C_Nodes \times C_Trees \rightarrow Values$
$\forall (n, t) \in C_Nodes \times C_Trees, C_read(n, t) = t(n)$
$\ldots$

L'isomorphisme entre les deux représentations est une mise en relation des éléments homonymes des deux spécifications. Et pour constituer un isomorphisme, ces relations doivent être bijective et préserver les opérations homonymes. Ainsi la relation entre objets (arbres, nœuds...) est étendue aux fonctions. Concrètement, ces relations sont définies dans les spécifications, et les propriétés d'isomorphismes y apparaissent comme des assertions à prouver. Voici un aperçu de ce à quoi ressemble la spécification de l'isomorphisme.

Isomorphismes
// caractérisation de la relation entre les arbres des deux représentations
$AC_NodeIso$: $A_Nodes \leftrightarrow C_Nodes$
$AC_TreeIso$: $A_Trees \leftrightarrow C_Trees$
$\ldots$ *spécification technique précise de* $AC_TreeIso$ $\ldots$
$\ldots$
$CA_NodeIso$: *relation inverse de* $AC_NodeIso$
$CA_TreeIso$: *relation inverse de* $AC_TreeIso$
$\forall t \in A_Tree,\ AC_TreeIso(t) = CA_TreeIso \circ t$
(rappel : $\circ$ désigne la composition de fonctions)
$\ldots$
// propriétés prouvées : symétries, bijections
$AC_NodeIso$ *est une fonction totale* et $CA_NodeIso$ *est une fonction totale*
$AC_NodeIso$ *est une bijection* et $CA_NodeIso$ *est une bijection*
$AC_TreeIso$ *est une fonction totale* et $CA_TreeIso$ *est une fonction totale*
$AC_TreeIso$ *est une bijection* et $CA_TreeIso$ *est une bijection*
$\forall t \in C_Tree,\ CA_TreeIso(t) = AC_TreeIso \circ t$
$\ldots$
// propriétés prouvées : préservations
$\forall n \in A_Nodes,\ AC_NodeIso(A_father(n)) = C_father(AC_NodeIso(n)))$
$\forall n \in A_Nodes,\ \forall t \in A_Trees,\ (A_read(n,t) = C_read(AC_NodeIso(n), AC_TreeIso(t)))$
$\forall n \in C_Nodes,\ CA_NodeIso(C_father(n)) = A_father(CA_NodeIso(n)))$
$\forall n \in C_Nodes,\ \forall t \in C_Trees,\ (C_read(n,t) = A_read(CA_NodeIso(n), CA_TreeIso(t)))$
$\ldots$

L'isomorphisme a été prouvé avec l'atelier $\mathcal{B}$ (preuve du premier ordre). La partie "difficile" est celle qui traite des définitions spécifiques aux représentations (caractère bijectif et correspondance des fonctionnalités de base). Pour toutes les autres fonctions, les preuves ont été faciles car complètement méthodiques du fait que ces fonctionnalités sont définies de la même manière dans les cas concret et abstrait. Comme l'atelier $\mathcal{B}$ ne propose pas de facilité particulière pour les isomorphismes, ces preuves méthodiques ont été un peu fastidieuses car manuelles, mais on peut les voir comme automatiques même si la mise en œuvre a été humaine.

La preuve de cet isomorphisme permet de prouver très simplement le premier raffinement. En effet, remplacer les arbres abstraits dans l'algorithme abstrait par les arbres concrets consiste simplement à remplacer les opérations sur les arbres abstraits par les opérations homonymes sur les arbres concrets (A_father devient C_father, A_sons devient C_sons,etc). Comme ces opérations se comportent de la même manière ("préservées" par l'isomorphisme), les algorithmes qui les utilisent aussi. La preuve de l'équivalence des deux algorithmes a été quasi-automatique comme l'avait été celle des opérations définies identiquement dans les deux représentations.

Ci-dessous un aperçu pour l'opération de lecture dans la mémoire (pour l'écriture, c'est très similaire). Techniquement, l'état comprend la mémoire, une variable pour recevoir l'argument de l'opération avant l'appel (le nœud dont on veut la valeur) et une variable pour y trouver le résultat après l'appel (la valeur, en cas de succès). Une fonction "$secure$" vérifie que la mémoire n'est pas corrompue pour le nœud à lire (condition pour exiger un résultat précis de la part de l'opération de lecture). Lors de son appel, l'opération retourne dans "$test$" une

indication sur la validité du résultat (non corruption).

Algorithme sur les arbre abstraits
// Variables d'état : (initialisées par des valeurs quelconques)
A_Memory : A_Trees
$A_Node_argument$: A_Nodes
A_Result_value : $Values$

// Opération de lecture
Read (return $test$) =
IF $A_secure(A_Node_argument, A_Memory)$
THEN $test := SUCCESS$ & $A_Result_value := A_read(A_Node_argument, A_Memory)$
ELSE $test := ERROR$ & $A_Result_value :=$ non spécifié, i.e n'importe quelle valeur
END ;

Algorithme sur les arbre concrets, RAFFINE "Algorithme sur les arbre abstraits"
// Variables d'état : (initialisées par des valeurs quelconques)
C_Memory : C_Trees
$C_Node_argument$: C_Nodes
C_Result_value : $Values$
// Opération de lecture
Read (return $test$) =
IF $C_secure(C_Node_argument, C_Memory)$
THEN $test := SUCCESS$ & $C_Result_value := C_read(C_Node_argument, C_Memory)$
ELSE $test := ERROR$ & $C_Result_value :=$ non spécifié, i.e n'importe quelle valeur
END
// Invariant de liaison (exprime la relation entre l'état abstrait et l'état concret)
$A_Memory = CA_TreeIso(C_Memory)$
$A_Node_argument = CA_NodeIso(C_Node_argument)$
$A_Result_value = C_Result_value$

La preuve de raffinement assure que, "modulo" la relation entre les deux états,
— Tout état résultant de l'initialisation de la machine concrète correspond à un état possible après l'initialisation de la machine abstraite.
— Toute exécution de l'opération concrète donne un résultat correspondant à un résultat possible de la machine abstraite, à partir de deux situations de départ correspondantes (par l'invariant de liaison).

Le raffinement assure donc que le comportement concret correspond à un comportement abstrait possible donc acceptable. C'est une réduction du non-déterminisme qui traduit un choix d'implémentation parmi ceux autorisés par la spécification, et certains comportement abstraits possibles peuvent donc être éliminés au niveau concret. Dans le cas du passage des arbres abstraits aux concrets, seule la représentation de l'état change et l'algorithmique est conservée sans modification si bien que aucun comportement n'est en fait éliminé (même si ça n'est pas plus prouvé que requis par la logique du raffinement). Lors de la deuxième étape de raffinement, il en va différemment.

5.2.3.2 De l'algorithme abstrait (fonctionnel récursif) au concret (impératif itératif)

Lors de la deuxième étape de raffinement, c'est l'algorithmique qui est modifiée. La représentation ne change pas et donc les variables d'état sont les mêmes. Les opérations ont le même profile : elles ont les mêmes arguments et renvoie le même type de valeur de retour. Ce qui change est le calcul et, là encore, pour avoir un raffinement, le résultat de ce calcul (valeur de retour et effet sur les variables d'état) doit faire partie des résultats possibles de l'algorithme abstrait.

L'initialisation concrète est évidemment un raffinement de l'initialisation abstraite :

— au niveau abstrait aucune contrainte n'est imposée si bien que l'initialisation affecte des valeurs quelconques aux variables
— au niveau concret, l'initialisation est déterministe (requis pour les "implémentations" en B) et on affecte donc des constantes (leur valeur importe peu, mais elles sont fixées a priori)
— ces constantes sont clairement un cas possible de valeurs quelconques

La lecture concrète raffine l'abstraite en fixant ce qui ne l'est pas au niveau abstrait, c'est-à-dire la valeur retournée dans la variable "C_Result_value" quand une corruption de la mémoire est détectée. Dans l'algorithme abstrait ci-dessous (écriture re-simplifiée pour être plus lisible ; C_Result_value devient $Result$, par exemple), nous mettons la notation B utilisée pour l'affectation du résultat en cas d'erreur, i.e. ":∈".

Read = IF $secure(Node_argument, Memory)$	
THEN $Result := read(Node_argument, Memory),$	$return(SUCCESS)$
ELSE $Result :\in Values,$	$return(ERROR)$

Plus haut, nous avions donné l'intuition ":= non spécifié, i.e n'importe quelle valeur". La sémantique précise est que l'affectation prend n'importe quelle valeur dans l'ensemble $Values$, de façon non-déterministe. L'algorithme concret (non transcrit car assez complexe) utilise l'affectation déterministe ":=" quelque part dans son code et pour un état mémoire donné renvoie une valeur déterminée. L'autre différence fondamentale est qu'au niveau abstrait rien n'est dit quant à l'ordre des calculs et seul leur effet compte, tandis qu'au niveau concret on est dans un séquencement bien identifié.

L'algorithme concret est la transcription en $\mathcal{B}$ de l'algorithme 2 (voir page 49) des arbres de Merkle réguliers. Quelques petites modifications ont été nécessaires pour le faire rentrer dans les contraintes sur la forme des implémentations en $\mathcal{B}$ (essentiellement de la syntaxe) mais elles sont mineures et on peut considérer les deux versions comme identiques. Le seul point un petit peu significatif est qu'il a fallu expliciter l'abréviation "..." du parcours des fils d'un nœud par une boucle "for" développée en $\mathcal{B}$. En effet une implémentation B doit être complète : aucun des détails concrets de la réalisation ne peut être omis et tout raccourci abstrait doit disparaître.

Nous ne rentrons pas ici dans les détails de l'algorithme et de sa preuve, mais essayons simplement d'en donner les grandes idées, pour beaucoup assez générales. Il s'agit d'un algorithme impératif usuel avec une initialisation de variables locales internes puis une boucle (parcours sur la mémoire) à la fin de laquelle ce qui doit être retourné (succès/échec et valeur lue) est déterminé. Sur ce schéma, le prouveur $\mathcal{B}$ supporte la mise en œuvre de preuve de Hoare avec invariant de boucle usuel. Dans notre cas, ce principe de preuve a suivi la forme suivante :

Considérant un programme `INIT; i:=0; WHILE(TEST_i) iteration_i; i++; END_BOUCLE`,
Considérant une propriété ϕ que l'on veut vraie à la fin de la boucle,

— Montrer que la boucle termine. La technique consiste à trouver un variant, c'est-à-dire un entier V, positif au départ, dont on montre :
  1. qu'il décroît (de façon entière) à chaque itération
  2. que sa valeur reste positive dans la boucle

  Par [1], l'entier V atteindra Zéro (ou moins) en un nombre fini d'itérations et par [2], à ce moment on ne pourra rester dans la boucle. Donc on sort de la boucle au bout d'un nombre fini d'itérations.

— Trouver un invariant de boucle, c'est-à-dire une propriété Ω portant sur les variables locales. $\Omega(i)$ dénote le fait que Ω est vrai des variables locales au début de l'itération i. Et montrer
  — que l'initialisation établit Ω, i.e. $\Omega(0)$
  — que les itérations intermédiaires préservent Ω, i.e $\Omega(i)$ & $TEST_i \Rightarrow \Omega(i+1)$
  — que Ω à la dernière itération assure le résultat recherché, i.e $\Omega(i)$ & $not(TEST_i) \Rightarrow \phi$

On peut alors en déduire que ϕ est vrai à la fin du programme.

Voici ce à quoi cela ressemble plus précisément pour l'opération de lecture implémentée concrètement par une boucle (raffinement sans invariant de liaison puisque les états concrets et abstraits sont les mêmes). En résumé, la valeur du nœud est stockée lors de la première itération et les itérations suivantes servent à remonter le long des ascendants du nœud afin de vérifier qu'ils vérifient bien la propriété de Merkle (ils sont un `hash` de leurs fils). Si on arrive à la racine sans problème, cela assure que la mémoire n'est pas corrompue (du moins pour le nœud consulté) d'où un test en succès et une valeur significative. Sinon il y a corruption et le test est mis en échec, arrêtant la boucle. Voici donc ce "cœur" d'algorithme, présentant l'essentiel en "oubliant" tous les aspects techniques (forme précise des variables, variables auxiliaires...).

Algorithme concret, RAFFINE "Algorithme abstrait sur les arbre concrets"
// Variables d'état : $Memory, Node, Result$
// Lecture
Read (return $Test$) =
VAR (initialisation des variables locales)
$Test := unknown, Current := Node, \ldots$
WHILE ($Test = unknown$)
si 1ère itération, $Result := Memory(Node),\ Current := calcul_du_pere_de(Current)$.
si autre itération, vérification de la propriété de Merkle, et
$-$ si la vérification échoue, $Test := echec$
$-$ sinon, $*$ si on est à la racine, $Test := success$
$*$ sinon $Current := calcul_du_pere_de(Current)$
END_WHILE

La propriété qui doit être vraie en fin de boucle pour assurer que le résultat de l'algorithme est le même que celui de l'algorithme abstrait, est la propriété ϕ suivante :

$not(secure(Node, Memory))$ & $Test = echec$
OR $secure(Node, Memory)$ & $Test = success$ & $Result := read(Node, Memory)$

Sachant que $secure(n,t)$ est défini par $\forall x \in ascendants(n),\ t(x) = merkle(x,t)))$, où

les ascendants d'un nœuds sont ses père, grand-père... et $merkle(x, t)$ désigne la valeur espérée pour le nœud x dans l'arbre t (`hash` des valeurs de ses fils).

La preuve de Hoare avec invariant de boucle a été réalisée comme suit :

— Pour le variant V qui permet de prouver la terminaison de la boucle, on a considéré la profondeur du nœud courant ($Current$). Clairement, elle est positive au départ, décroît lors des itérations et est positive dans la boucle (son arrivée à 0, profondeur de la racine, provoque la sortie de boucle.

— Pour l'invariant de boucle Ω on a considère (invariant essentiel, détails techniques omis) :
Soit ψ = "$\forall x \in ascendants(Node), depth(x) > depth(Current) \Rightarrow Memory(x) = merkle(x, Memory)$"
Ω = "Si $Test = echec$ alors $not(\psi)$ sinon (ψ & $Result = read(Node, Memory)$)"

Ω qui est vrai dans la boucle, où $Test$ vaut $unknown$, sauf à la première itération (on n'a pas encore $Result = read(Node, Memory)$ car la valeur de $Node$ en mémoire n'a pas encore été lue). La vraie formule est plus complexe et résout ce problème, mais notre objectif ici est de donner l'intuition simplement et non la preuve exacte, d'où cet abus.
Ω assure ϕ en sortie, où $Test$ vaut $success$ ou $echec$.

Cette version simplifiée masque une preuve plus complexe. Pour montrer que $\Omega \Rightarrow \phi$ en sortie de boucle, des lemmes ϕ_{Lem} sont nécessaires qui demandent à leur tour des invariants de boucle Ω_{lem} pour être établis. Par exemple il faut montrer que "tous les ascendants de profondeur comprise entre celle de la racine et celle de l'arbre" recouvre bien "tous les ascendants".

Enfin, le dernier aspect non strictement technique de la preuve d'implémentation est que dans l'algorithme concret les calculs modifiant les variables d'état sont souvent codés directement à l'endroit de leurs affectations. Il n'y a pas d'invocation des fonctions définies sur les arbres. Il faut donc montrer tout au long qu'invoquer ces calculs et invoquer les fonctions utilisées pour les descriptions abstraites revient au même. Ceci est facile car la représentation concrète des arbres a été écrite pour que ce soit direct/trivial. Par exemple la définition de C_father est décalquée sur $calcul_du_pere_de(\ldots)$.

La preuve de l'isomorphisme entre les deux représentations des arbres présentée dans la section précédente permet d'étendre cette équivalence entre calcul direct et utilisation de fonction sur les arbres concrets à une équivalence entre calcul direct et utilisation de fonction sur les arbres abstraits. En fait, cela correspond à la transitivité de la relation d'implémentation entre machines plus abstraites et machines plus concrètes, et ainsi on a montré que l'algorithme concret est bien une réalisation correcte de celui qui a été décrit au niveau le plus abstrait. Bien que plus complexe puisqu'il y a modification de l'état mémoire, les preuves concernant l'opération d'écriture sont du même type.

5.2.3.3 Preuve formelle

La preuve présentée ci-dessus a été menée avec l'atelier $\mathcal{B}$ dans le cadre du projet TRESCCA. Il s'agit donc d'une preuve basée sur la logique du premier ordre. Il s'agit aussi d'une preuve formelle, donc beaucoup plus détaillée que ce dont nous avons rendu compte ici. En effet, le preuve formelle exige l'identification de tous les pas de preuve, y compris par exemple la mise en œuvre de la commutativité du "ou" et du "et" logiques. Donc bien que non fondamentalement difficile, cette preuve est relativement longue. La contrepartie est qu'elle est sûre (une version antérieure de l'atelier $\mathcal{B}$ a été certifiée et les suivantes reposent sur le même cœur) et offre une garantie forte. L'autre intérêt de ce travail est méthodologique : en offrant un niveau abstrait lié de façon sûre au concret, il permet d'y travailler et d'oublier les détails d'implémentation. C'est au niveau abstrait par exemple que la preuve que le principe des arbres de Merkle fonctionne est envisageable.

5.2.4 Quelques précisions sur la preuve d'équivalence avec *EasyCrypt*

Nous avons tenté dans ce qui suit de montrer que les deux versions des procédures de lecture/écriture vérifiées des arbres de Merkle sont équivalentes respectivement. Équivalentes dans le sens, où les deux procédures se comportent de même façon vis à vis d'un observateur : elles effectuent les mêmes calculs observables.

Pour mener cette preuve, nous avons utilisé l'outil *EasyCrypt* [55] développé conjointement entre l'IMDEA Software Institute et Inria [56]. *EasyCrypt* est un outil formel d'aide à la vérification de cryptosystèmes. Son approche de preuve repose sur la formalisation de constructions cryptographiques à partir de code concret, dans laquelle la sécurité et les hypothèses de sécurité sont modélisées à partir de programmes. La méthode de raisonnement utilisée est basée sur la logique de Hoare étendue aux probabilités.

EasyCrypt permet d'écrire des jugements de preuve de la forme :

$$\vdash p_1 \sim p_2 : \varphi \Rightarrow \psi$$

où p_1, p_2 sont deux programmes, φ et ψ (respectivement pré-condition et post-condition) sont des relations sur des états déterministes de la mémoire, c'est-à-dire, des formules logiques qui lient les états de la mémoire. Un exemple de formules, $=\{x\} \Rightarrow x\{1\} = x\{2\}$ énonce que si avant leur exécution les deux programmes p_1 et p_2 prennent comme entrée une même valeur pour la variable x, alors ils retournent la même valeur pour x après l'exécution. Notons, la notation $x\{1\}$ et $x\{2\}$ qui représente la variable x correspondant au premier et au deuxième programme respectivement.

EasyCrypt permet d'établir la validité des jugements grâce aux prouveurs SMT auxquels il fait appel (`alt-ergo` et `Z3`).

Dans un même fichier (portant l'extension `.ec`) l'utilisateur peut décrire le modèle du système, écrit sous forme de code concret ; ainsi que les hypothèses à prouver écrites sous forme de lemmes ou de théorèmes. Un fichier *EasyCrypt* comporte trois parties :

— une partie déclaration, fiche descriptive des types de données, des constantes et des opérations qui serviront dans la construction du modèle et l'expression des jugements.
— une partie code décrite dans des entités appelées *modules* regroupant un ensemble de définitions apparentées ; Il comporte des variables globales typées et des procédures. Pour coder les procédures *EasyCrypt* offre un langage impératif dont la syntaxe est aussi riche qu'un langage de programmation, permettant l'utilisation des boucles, des appels de procédures, etc. À l'instar d'un langage de programmation, les procédures peuvent avoir des résultats de retour d'un type donné ou de type "vide" ou "nul".
— une partie jugement contenant la définition des prédicats et la définition des hypothèses (lemmes ou théorèmes) ainsi que les preuves. Les jugements sont des formules logiques exprimées en utilisant les opérateurs logiques et les quantificateurs.

5.2.4.1 Modèle

Dans cette partie nous formalisons le modèle de sécurité de la mémoire à partir des programmes définis dans la section 2.3. Nous formalisons également des propriétés comportementales visant à garantir des propriétés de sécurité à vérifier, notamment les propriétés d'équivalence de programmes. Nous donnerons aussi leurs preuves dans l'outil *EasyCrypt*.

5.2.4.1.1 Formalisation des constructions Nous modélisons la mémoire par la structure de données de type *map* destinée à ranger les données en fonction des adresses, une mémoire étant une association d'adresses et de données. Nous représentons une adresse par un couple d'entiers l'abscisse et l'ordonnée ; une données par la type abstrait *data* ; et une clé par un type abstrait *key*.

$$\begin{array}{lcl} \textbf{\textit{type}}\ key. & & \\ \textbf{\textit{type}}\ data. & & \\ \textbf{\textit{type}}\ addr & & int * int. \\ \textbf{\textit{type}}\ memory & = & (addr, data)\ map. \end{array}$$

Dans la représentation du modèle de sécurité d'une mémoire par un arbre de hachage, la zone mémoire à protéger est divisée en blocs. Nous définissons alors le type *block* comme étant un vecteur de mémoire (*array memory*).

$$\textbf{\textit{type}}\ block \quad = \quad memory\ array.$$

Un arbre de Merkle régulier est défini par son arité *a* et son niveau *s* que nous définissons comme des constantes.

$$\begin{array}{lcl} \textbf{\textit{const}}\ s & : & int. \\ \textbf{\textit{const}}\ a & : & int. \end{array}$$

Nous définissons aussi deux fonctions d'accès à la mémoire : la fonction *make* qui retourne le bloc associé à une adresse donnée ; la fonction *readb* qui calcule la donnée associée à un bloc.

$$\begin{array}{lll} \textbf{\textit{op}}\ \ make & : & memory \rightarrow addr \rightarrow block. \\ \textbf{\textit{op}}\ \ readb & : & block \rightarrow data. \end{array}$$

Dans le modèle d'arbre de Merkle, les feuilles de l'arbre étant les données à protéger et les nœuds sont les MAC qui protègent les données de manière hiérarchique, chaque nœud (y compris la racine) contient le condensé des blocs appartenant à un groupe : ses fils. Nous définissons alors un opérateur MAC qui calcule le condensé d'une donnée.

$$\textbf{\textit{op}}\ \ MAC \ : \ key * addr * data \rightarrow data.$$

Nous définissons également la fonction *father* qui calcule d'adresse du père d'un nœud donné et sa spécification par l'axiome *Prop_father*.

$$\begin{array}{lll} \textbf{\textit{op}}\ father & : & addr \rightarrow addr. \\ \textbf{\textit{axiom}}\ Prop_father & : & \forall (i : int, j : int), (0 \leq i < s) \wedge (0 \leq j) \wedge father(i,j) = (i+1, j\%a). \end{array}$$

Pour manipuler mémoire comme une entité à part, nous la définissons dans un module à part que appelons *M*, nous définissons également dans ce module une clé *k*.

module $M = \{$
var m : *memory*
var k : *key*
$\}.$

5.2.4.1.2 Modèle calculatoire

Une fois les types et les structures de données formalisés, nous construisons le modèle calculatoire de la mémoire sécurisée à partir du code source. Nous définissons deux modules, dans lesquels nous reprenons les codes sources de toutes les procédures définies dans la version de gestion des arbres réguliers (voir section 2.3, page 46) et la version des arbres creux initialisés (voir section 3.1, page 73) appelés respectivement par *M1* et *M2*.

Le module *M1* comporte les procédures d'initialisation, de lecture et d'écriture (*initialisation*, *verifyread*, et *verifywrite*).

module $M1 = \{$

proc *initialisation*$()$: ***unit*** $= \{\ldots\}$

proc *verifyread*$(i$: *int*$, j0$: *int*$, j1$: *int*$)$: *data* ***option*** $= \{\ldots\}$

proc *verifywrite*$(i$: *int*$, j0$: *int*$, j1$: *int*$, v$: *data*$))$: ***unit*** $= \{\ldots\}$
}

Les procédures *initialisation* et *verifywrite* retournent le type vide, ce type est désigné en *EasyCrypt* par ***unit***.

EasyCrypt introduit le mot clé ***option*** afin de permettre d'étendre un type donné à une valeur inconnue ou non définie (sans avoir à le spécifier). Comme le type retourné par la procédure *verifyread* est une donnée de type *data* ou possiblement un message d'erreur (ou une exception), nous utilisons alors *data* ***option*** comme type de retour pour cette procédure. Ces procédures manipulent et accèdent à la mémoire définie dans le module M en utilisant la notation $M.m$.

module $M2 = \{$

proc *initialisation*$()$: ***unit*** $= \{\ldots\}$

proc *verifyread*$(i$: *int*$, j0$: *int*$, j1$: *int*$)$: *data* ***option*** $= \{\ldots\}$

proc *verifywrite*$(i$: *int*$, j0$: *int*$, j1$: *int*$, v$: *data*$)$: *unit* $= \{\ldots\}$
}

Dans le module *M2* nous retrouvons les trois procédures d'initialisation, de lecture et d'écriture (*initialisation*, *verifyread*, et *verifywrite*).

Une fois le modèle calculatoire spécifié, nous procédons alors à la formalisation des jugements et des propriétés et à la réalisation des preuves.

5.2.4.2 Preuve

Dans *EasyCrypt* une preuve est structurée par une suite de lemmes et la théorie d'*EasyCrypt* permet l'utilisation de plusieurs notions : les types, les prédicats, les opérateurs, les modules, les axiomes et les lemmes.

Commençons par définir une notion que nous utiliserons dans la suite, la notion de mémoire équivalente dans la représentation sous forme d'arbre de Merkle. Dans cette structure la mémoire est stockée dans les feuilles. Nous définissons le prédicat *equiv_mem* qui énonce que deux mémoires sont équivalentes si les données contenues dans les nœuds feuilles sont respectivement égales.

pred *equiv_mem*$(m1$: *memory*$)(m2$: *memory*$)$
$$= \forall (j : \textit{int}), 0 \leq j \Longrightarrow m1.[(0, j)] = m2.[(0, j)].$$

Nous montrons le lemme nommé *init* qui énonce que les procédures d'initialisation d'une mémoire décrites dans les modules $M1$ et $M2$ (des deux versions) sont équivalentes (c'est-à-dire **equiv**$[M1.$*initialisation* $\sim$ $M2.$*initialisation*$])$ dans le sens où partant de la même mémoire après exécution de ces deux procédures le contenu des résultats est équivalents. Ce lemme garantie que les procédures d'initialisation ne touchent pas à la mémoire.

$$\textbf{\textit{lemma}}\ init:\ \textbf{\textit{equiv}}[M1.\textit{initialisation} \sim M2.\textit{initialisation} : \\ =\{M.m\} \Longrightarrow \textit{equiv_mem}(M.m\{1\})(M.m\{2\})].$$

Proof. La preuve est donnée dans 5.3.5.

Nous donnons un autre prédicat *encrypt_correct* qui définit ce qu'un arbre correctement calculé.

$$\textbf{\textit{pred}}\ \textit{encrypt_correct}(m : \textit{memory}, k : \textit{key}, i : \textit{int}, j : \textit{int}) = \\ \textit{mem}(\textit{dom}\ m)(i,j) \Rightarrow m.[\textit{father}(i,j)] = \textit{Some}(\textit{MAC}(k,(i,j),\textit{readb}(\textit{make}\ m(i,j)))).$$

Ce prédicat énonce qu'étant données une mémoire, une adresse et une clé, si l'adresse appartient au domaine de la mémoire ($\textit{mem}(\textit{dom}\ m)(i,j)$) alors la donnée contenue à l'adresse du nœud père est égale à la valeur du condensé cryptographique de l'ensemble de ses fils $\textit{Some}(\textit{MAC}(k,(i,j),\textit{readb}(\textit{make}\ m(i,j))))$.

Notons l'utilisation de l'opérateur *Some* ; *EasyCrypt* gère le type de données *map* à la manière du type d'Haskell *Maybe*, pour un type de donnée de type $(\alpha,\beta)\,map$, l'accès à une donnée quelconque α retourne une donnée de type *Some* $\beta\,|\,$*None* selon que la valeur de de la donnée d'entrée est liée ou non.

Nous montrons que l'appel de la procédure d'initialisation du module $M1$ sur une mémoire garantie après exécution de la procédure que la mémoire résultante est correctement chiffrée. La preuve d'un tel lemme requiert l'accès au résultat de la procédure d'initialisation.

Remarquons, qu'à la manière dont la procédure *initialisation* est codée dans le module $M1$ nous n'avons pas accès à la mémoire, la procédure ne retourne aucun résultat (le résultat de type *unit*). Pour pallier ce problème et sans modifier le comportement de celle-ci nous ajoutons dans le module une procédure englobante nommée *initial* qui ne fait rien d'autre qu'appeler procédure d'initialisation et retourne la mémoire.

module $M1 = \{$

...

proc *initial*() : *memory* $= \{$

initialisation() ;
return M.m;
}
}

Nous définissons alors le lemme ***lemma*** *init_correct1* qui énonce que la procédure *initialisation* préserve le prédicat *encrypt_correct*.

lemma *init_correct1* : ***hoare***[*M1.initial* : ***true*** $\Longrightarrow$
$\forall(i : int, j : int), 0 \leq j \wedge 0 \leq i < s \Rightarrow$ *encrypt_correct* ***res*** *M.k* $i\ j]$.

Les valeurs de toutes les adresses de la mémoire résultant de l'exécution de la procédure *initial* (introduite par la variable spécifique ***res***) sont correctement chiffrées.

Proof. La preuve est donnée dans 5.3.5.

Montrons également que la procédure d'écriture *verifywrite* (dans les deux versions) préserve le prédicat *encrypt_correct*, i.e, qu'après chaque écriture la mémoire demeure correctement condensée. Pour ce faire, nous procédons de la même manière que pour la procédure *initialisation*, nous ajoutons dans les deux modules $M1$ et $M2$ une procédure englobante appelée *write* qui retourne l'état de la mémoire à la suite de l'appel de *verifywrite*.

module $M1 = \{$

...

proc *write*$(i : int, j : int, v : data)$: *memory* $= \{$
verifywrite$(i, j\%a, j - (a * (j\%a)), v)$;
returnM.m;
}
}

Nous définissons alors le lemme *correct_write* qui énonce que le résultat d'une écriture vérifiée satisfait le prédicat *encrypt_correct*, préserve la notion de mémoire correctement condensée.

lemma *correct_write* :
hoare[$M1$.*write* : ***true*** $\Longrightarrow$

$\forall(i\colon int, j\colon int), 0 \leq j \;\wedge\; 0 \leq i < s \Rightarrow$ *encrypt_correct* ***res*** $M.k\, i\, j]$.

Proof. La preuve est donnée dans 5.3.5.

Nous montrons le même lemme pour la procédure *write* du module $M2$.

Nous montrons enfin un lemme, appelé *Read*, sur l'équivalence des procédures de lecture vérifiée *verifyread*.

equiv *Read* : $M1$.*verifyread* $\sim$ $M2$.*verifyread* : $=\{M.k, M.m\} \wedge$
equiv_mem $M.m\{1\}$ $M.m\{2\} \wedge \forall(i : int, j : int), (0 \leq j) \wedge (0 \leq i \leq s) \wedge$
encrypt_correct ***M.m***$\{1\}$ ***M.k***$\{1\}$ $i\, j$ $\wedge$ *encrypt_correct* ***M.m***$\{2\}$ ***M.k***$\{2\}$ $i\, j \Longrightarrow =\{$***res***$\}$.

Proof. La preuve est donnée dans 5.3.5.

Le lemme énonce que partant d'un même espace mémoire (ou équivalent) et d'une même clé, et que si les deux espaces mémoire sont correctement condensés alors le résultat de la lecture *verifyread* du module *M1* et le résultat de lecture *verifyread* du module *M2* sont égaux ($=\{$***res***$\}$).

Le lemme *Read* garantie que dans le cas où il n'y a pas de corruption de la mémoire les deux procédures *verifyread* sont équivalentes dans le sens où elles retournent le même résultat.

5.2.5 Conclusion

Le travail présenté dans cette section rend compte de ce qui a été fait au sein de l'équipe du LabSoC afin de mieux cerner les garanties que l'on peut espérer apporter au système SecBus avec une approche de type preuve. Il en résulte une clarification de ce que pourrait être une démarche complète pour la partie matérielle "théorique". Cette démarche "modularise" la preuve et permet de traiter séparément différents aspects. Elle ne s'applique qu'au théorique, au sens ou aucune preuve mathématique ne peut garantir l'état physique du matériel. Qui plus est, cette étude à eu lieu sur des "proto-algorithmes", avant que les implémentations matérielles réelles soient fixées. Enfin, cette étude ne traite que du matériel SecBus, en ce sens qu'elle ne considère ni les logiciels (bootloader, applications...) utilisant les service offerts et l'intelligence de cette utilisation, ni l'utilisabilité elle-même de ces services pour protéger des données sensibles.

5.3 Évaluation des performance du module matériel HSM

5.3.1 Introduction

Le module de sécurité matériel HSM (introduit dans la section 4.3, voir page 114) est responsable du chiffrement et de déchiffrement des données transférées entre le SoC et ses mémoires externes et de la gestion et la vérification de leur intégrité. Il est situé à l'intérieur du SoC, entre les interconnexions internes et le contrôleur mémoire.

Nous présentons ici l'implémentation du HSM sur une carte à base de FPGA de la famille Zynq de Xilinx [57]. Ces FPGA sont constitués de deux parties :

— La partie *Processing System* (PS) qui contient notamment deux cœurs de processeur ARM, un contrôleur mémoire et quelques périphériques

— La partie *Programmable Logic* (PL) qui contient une matrice FPGA programmable classique

Ces deux zones communiquent entre elles grâce à plusieurs bus AXI.

L'espace d'adressage 32 bits des cœurs ARM est divisé en quatre zones de chacune 1 Gio. La première permet d'adresser directement la mémoire physique via le contrôleur mémoire (lien AS0 dans la figure 5.6). Les deux zones suivantes permettent aux processeurs d'envoyer des requêtes vers la partie PL (via les deux bus AXI, AS1 et AS2 sur la figure). La dernière zone permet aux processeurs d'adresser les périphériques situés dans la partie PS et notamment la *On-Chip Memory*.

La figure 5.6 montre comment le HSM est inséré dans une plate-forme de prototypage à base de Zynq (comme, par exemple, le ZedBoard [58]). Le logiciel s'exécutant sur la partie PS a été modifié de manière à utiliser la plage d'adresse AS2 au lieu d'AS0 pour accéder à la mémoire externe. Ainsi, toutes les transactions mémoires ont été déroutées via la partie PL où est implémenté le HSM. Cette configuration a permis de tester le HSM comme s'il était physiquement présent entre le processeur et son contrôleur mémoire.

Cette section présente les résultats de l'évaluation de performance du HSM sur la plateforme de démonstration.

5.3.2 Coûts matériels

Le tableau 5.5 résume l'utilisation des ressources occupées par le HSM sur le FPGA Xilinx Zynq-7020 de Xilinx de la carte ZedBoard.

Le Zynq-7020 étant le deuxième plus petit membre de la famille et un FPGA étant beaucoup moins dense qu'un autre circuit intégré à technologie équivalente, ces résultats peuvent être considérés comme très raisonnables. L'utilisation (généralement optimiste) du rapport de conversion de Xilinx peut être estimé à environ 750 k portes ou 3 M transistors. La fabrication du HSM dans un processus de $14\ nm$ occuperait une surface de silicium d'environ $0,3\ mm^2$.

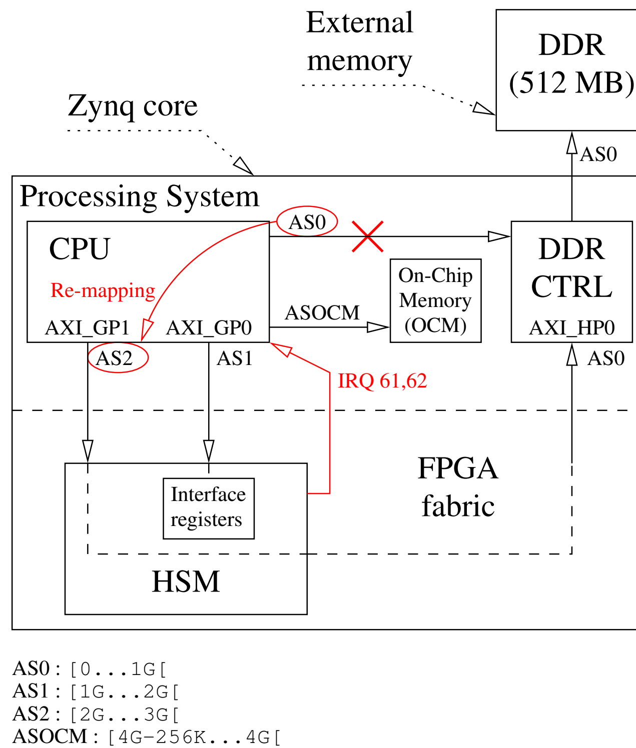


FIGURE 5.6 – Exemple d'un prototype HSM sur une carte ZedBoard à base de FPGA Zynq.

Type de ressource	Utilisation
Logique (LUTs)	59,69 %
Registres	3,67 %
Mémoire (Blocs RAMs)	6,29 %
DSPs	0 %

TABLE 5.5 – Résumé de l'utilisation des ressources matérielles.

5.3.3 Augmentation de l'empreinte mémoire

La flexibilité de protection fournit par le HSM, la capacité d'affiner et de le contrôler par le logiciel a un prix : les structures de données utilisées par le contrôle sont stockées dans la mémoire externe. De plus, les MAC utilisés pour protéger l'intégrité des pages mémoires sont également stockés dans la mémoire externe.

Le *Master Block* occupe $1/256$ (0,39 %) de l'espace total de mémoire à protéger. Par exemple, la protection d'un espace de 4 Gio de mémoire nécessite 16 Mio pour stocker le *Master Block* (qui contient les SP, les PSPE et le MMT).

Avec les hypothèses présentées précédemment, protéger l'intégrité d'une page mémoire en lecture seule (via des MAC) nécessite un quart de page supplémentaire (25 %) pour y stocker les MAC calculés. De même, protéger une page en lecture-écriture (via un arbre) nécessite un

tiers de page supplémentaire (33 %).

À titre d'exemple, un système disposant d'une mémoire externe de 4 Gio avec 256 Mio de mémoire en lecture-écriture protégée en intégrité et 1 Gio de mémoire en lecture seule protégée en intégrité, consacrerait $16 + 256/3 + 1024/4 = 357$ *Mio* (8,7 %) de sa mémoire externe aux structures de données nécessaires pour la protection (*Master Block*, MAC et MT).

5.3.4 Configurations et résultats

5.3.4.1 Configurations

Dans le but d'estimer l'impact sur les performances introduit par le module HSM, plusieurs expériences ont été menées dans diverses configurations. Une de ces configurations est utilisée comme référence et est nommée HD (*HSM Disabled*). Dans cette configuration le HSM est désactivé : tous les accès de la mémoire externe passent par le HSM mais aucune politique de sécurité n'est appliquée, le HSM est transparent. Cette référence est utilisée pour tenir compte de la dégradation de performances introduite par le routage de toutes les transactions par la partie FPGA qui tourne à une fréquence beaucoup plus faible que la partie PS.

Les autres configurations diffèrent par la politique de sécurité appliquée de façon identique à tout l'espace mémoire de la pile logicielle (le noyau Linux et des applications). Elles se distinguent aussi par la granularité de la protection : le HSM fonctionne sur les pages mémoire et prend en charge plusieurs tailles de page (4 kio, 64 kio, 1 Mio et 16 Mio). Ces configurations sont notées Gs_ci où :

- $s \in \{4K, 64K, 1M, 16M\}$ indique la granularité de la protection.
- $c \in \{N, O, C\}$ représente la protection de confidentialité appliquée :
  - N pour *None* : la page mémoire n'est pas chiffrée.
  - O pour *One-time pad* (OTP) : la page mémoire en lecture seule est chiffrée en utilisant le mode compteur du chiffrement par bloc.
  - C pour *Cipher Block Chaining* (CBC) : la page mémoire en lecture-écriture est chiffrée en utilisant le mode CBC du chiffrement par bloc.
- $i \in \{N, M, T\}$. s représente la protection d'intégrité appliquée :
  - N pour *None* : la page mémoire n'est pas protégée en intégrité.
  - M pour ensemble de MAC (*Mac set*) : la page mémoire est protégée par des ensembles de MAC classiques, calculés sur des blocs de données en utilisant le mode CBC-MAC du chiffrement par bloc. Ce mode de protection est appliqué sur des pages en lecture seule.
  - T pour arbres de MAC (*MAC Tree*) : la page mémoire en lecture-écriture est protégée par un arbre de Merkle dont les nœuds sont des MAC calculés en utilisant le mode CBC-MAC du chiffrement par bloc.

Par exemple, la configuration *G1M_CT* signifie que l'espace mémoire de la pile logicielle est divisé en pages de 1 Mio et chaque page est chiffrée par le mode CBC et protégée en intégrité par des arbres de MAC. Un deuxième exemple d'une configuration est *G4K_NN* pour lequel l'espace mémoire est divisé en pages de 4 Kio et sans protection.

Par contre, quelques configurations n'ont pas de sens et donc ne seront pas prises en compte dans nos tests. Ce sont :

— *Gs_OT* et *Gs_CM* : la combinaison entre un mode appliqué sur des pages en lecture seule avec un mode appliqué sur des pages lecture-écriture.
— *Gs_cM* : les configurations utilisant le mode d'intégrité M ne peuvent pas être appliquées sur un espace mémoire de la pile logicielle qui contient des pages en lecture-écriture. Ces configurations pourraient constamment détecter des violations d'intégrité, ce qui pourrait arrêter le système. Le même problème ne se pose pas avec le mode de confidentialité O car, si on l'applique sur des pages en lecture-écriture, il fonctionne, même si dans un point de vue de sécurité n'a pas de sens. Nous utilisons ce mode uniquement pour l'évaluation des performances.

Dans l'ensemble, nous évaluons 21 configurations : *HD*, *Gs_NN*, *Gs_ON*, *Gs_CN*, *Gs_NT* et *Gs_CT* où $s \in \{4K, 64K, 1M, 16M\}$.

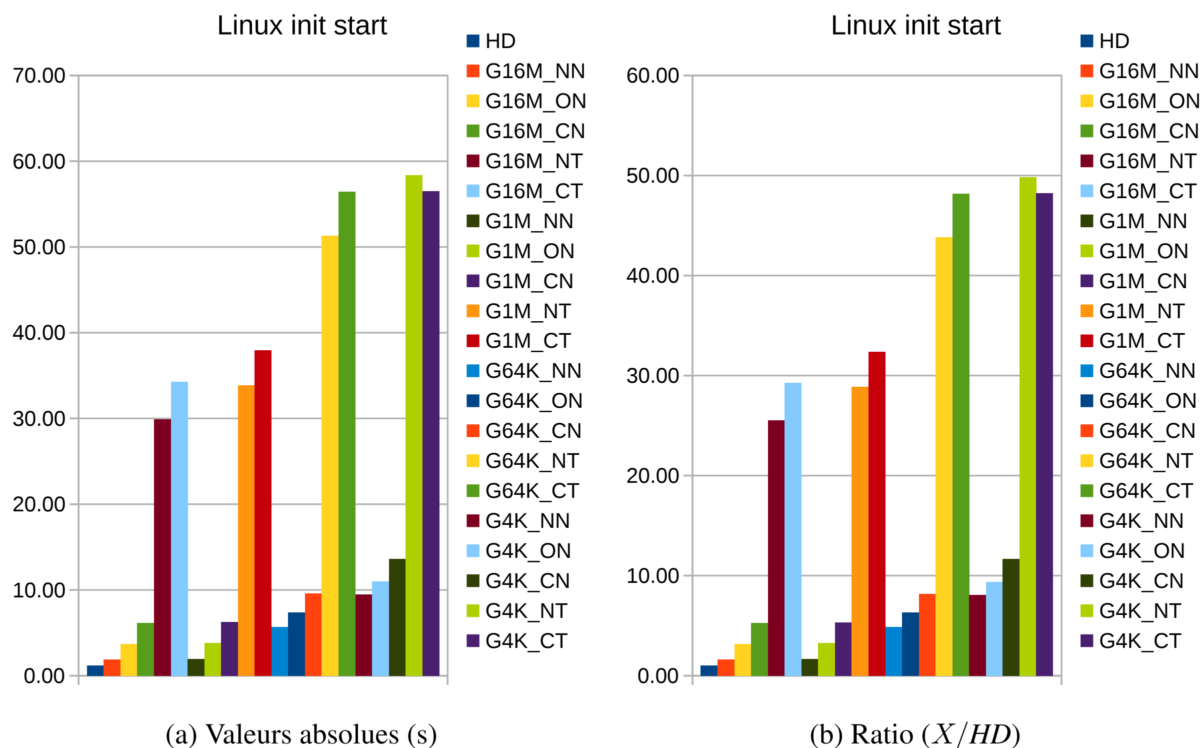
Dans toutes ces configurations une pile logicielle complète (Linux, BusyBox, applications) a été exécutée. De plus, plusieurs benchmarks classiques ont été utilisés :

— *Dhrystone* qui mesure les performances de calcul sur des entiers
— *Whetstone* qui mesure les performances de calcul flottant
— *RAMSpeed* et *RAMSMP* qui mesurent les performances des accès mémoires

Enfin, le temps d'initialisation du noyau Linux a également été mesuré en utilisant l'horodatage des messages du noyau.

Dans l'ensemble, 12 mesures différentes ont été effectuées dans chacune des 21 configurations :

— `Linux init start` (en secondes) : temps entre l'initialisation du timer matériel et le premier message du processus `init`
— `Linux init duration` (en secondes) : durée de démarrage du processus `init`.
— `dhrystone 10000000` (en `dhrystones` par seconde) : benchmark de la puissance de calcul entier (sortie de `dhrystone 10000000`)
— `whetstone 10000` (en `MIPS`) : benchmark de la puissance de calcul en virgule flottante (sortie de `whetstone 10000`)
— RAM bandwidth (en Mio/s) mesurée avec RAMSpeed et RAMSMP. Un Gio de données est lu ou écrit par un seul CPU (RAMSpeed) ou deux CPU, chacun exécutant un processus de lecture ou d'écriture (RAMSMP). La zone de 1 Gio est divisée en blocs de 1 Kio ou 32 Mio. Chaque bloc est lu ou écrit à plusieurs reprises et la durée moyenne est mesurée. Comme les blocs de 1 kio peuvent tenir dans les caches du processeur, l'utilisation de cette taille de bloc mesure principalement la bande passante entre le processeur et les caches. Les blocs de 32 Mio ne rentrent pas dans les caches et sont utilisés pour tenir compte de la latence de la mémoire externe. En total, huit tests ont été réalisés :
  — `RAM IW-2P-1GB-1kB` : RAMSMP, Écriture, 2 processus, 1 Gio de données, blocs de 1 kio
  — `RAM IW-2P-1GB-32MB` : RAMSMP, Écriture, 2 processus, 1 Gio de données, blocs de 32 Mio
  — `RAM IR-2P-1GB-1kB` : RAMSMP, Lecture, 2 processus, 1 Gio de données, blocs de 1 kio



(a) Valeurs absolues (s)

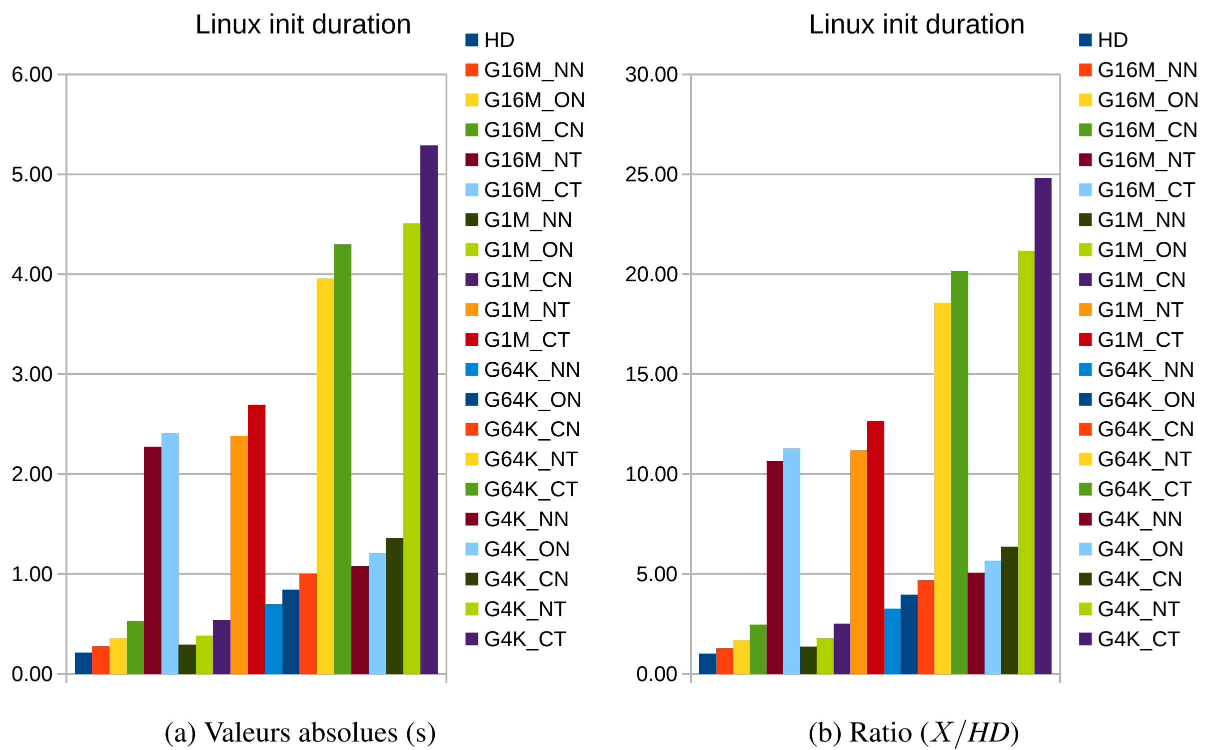
(b) Ratio (X/HD)

FIGURE 5.7 – Linux init start

— `RAM IR-2P-1GB-32MB` : RAMSMP, Lecture, 2 processus, 1 Gio de données, blocs de 32 Mio
— `RAM IW-1P-1GB-1kB` : RAMSpeed, Écriture, 1 processus, 1 Gio de données, blocs de 1 kio
— `RAM IW-1P-1GB-32MB` : RAMSpeed, Écriture, 1 processus, 1 Gio de données, blocs de 32 Mio
— `RAM IR-1P-1GB-1kB` : RAMSpeed, Lecture, 1 processus, 1 Gio de données, blocs de 1 kio
— `RAM IR-1P-1GB-32MB` : RAMSpeed, Lecture, 1 processus, 1 Gio de données, blocs de 32 Mio

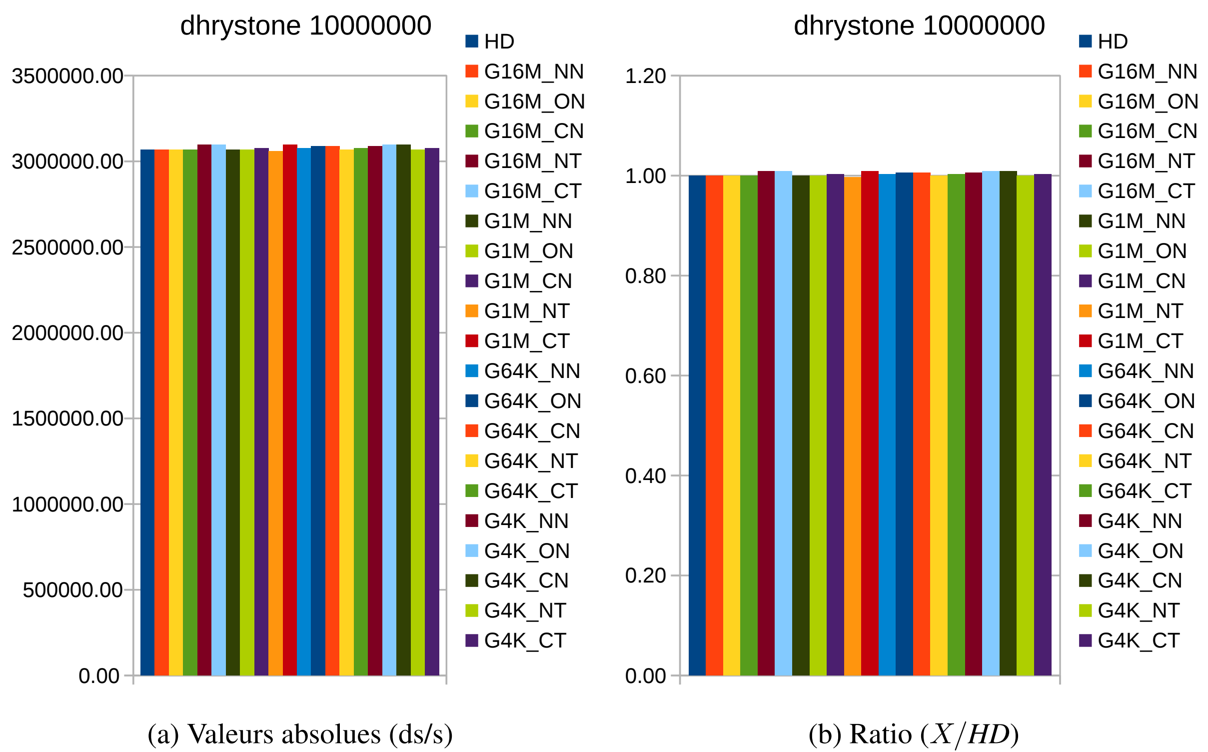
5.3.4.2 Résultats

Les figures 5.7 à 5.18 présentent les résultats des évaluations des performances. Chaque figure présente une des 12 métriques pour les 21 configurations et est proposée en deux versions : l'une avec les valeurs absolues, noté $M(Gs_ci)$, de la métrique considérée M pour les configurations considérées *Gs_ci*, l'autre, pour faciliter la comparaison, avec les ratios par rapport la configuration de référence *HD*. Pour les diagrammes des bandes passantes du RAM, le rapport est $M(HD)/M(Gs_ci)$, et $M(Gs_ci)/M(HD)$ pour les autres métriques.



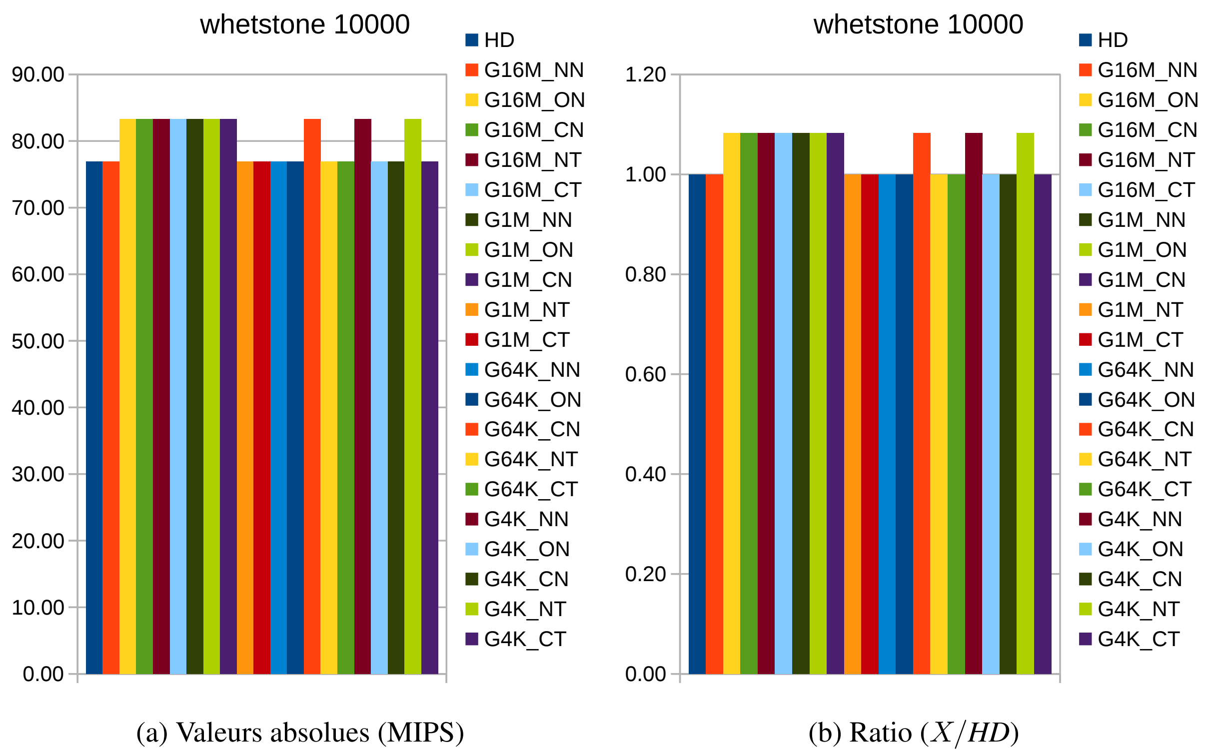
(a) Valeurs absolues (s) (b) Ratio (X/HD)

FIGURE 5.8 – Linux init duration



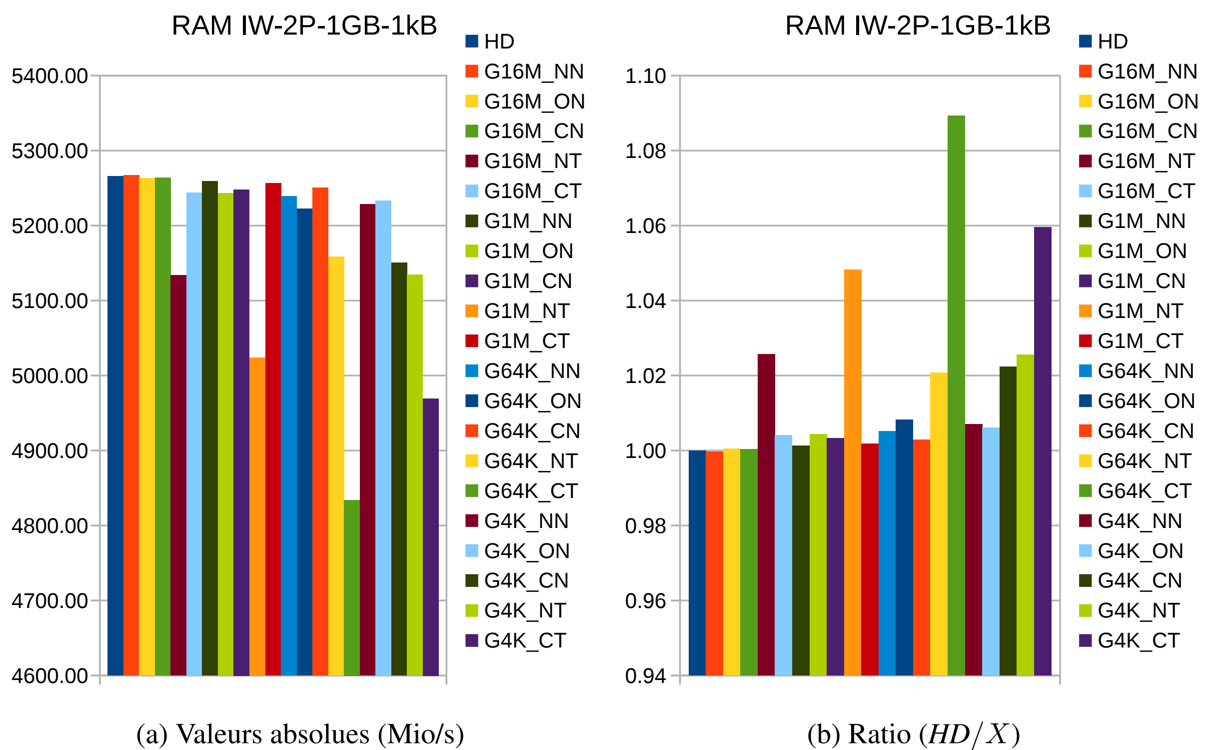
(a) Valeurs absolues (ds/s) (b) Ratio (X/HD)

FIGURE 5.9 – Dhrystone



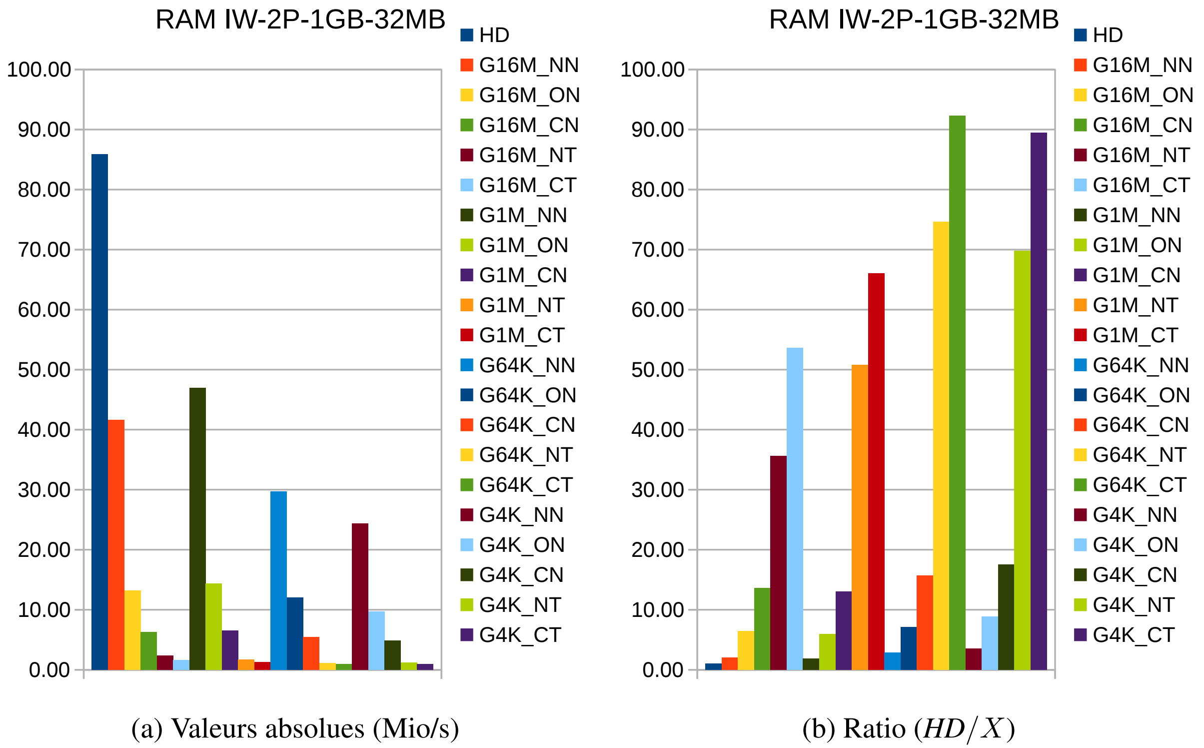
(a) Valeurs absolues (MIPS) (b) Ratio (X/HD)

FIGURE 5.10 – Whetstone



(a) Valeurs absolues (Mio/s) (b) Ratio (HD/X)

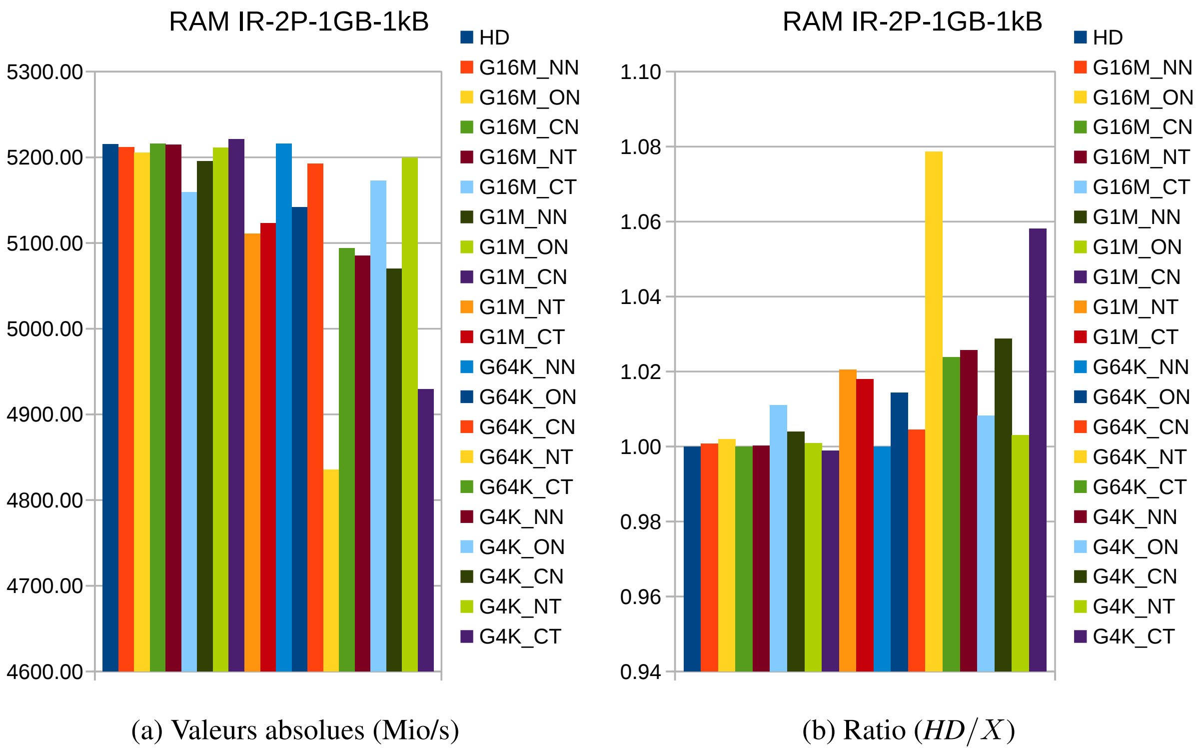
FIGURE 5.11 – RAMSMP 1 Gio, 2 processus, écriture, blocs de 1 kio



(a) Valeurs absolues (Mio/s)

(b) Ratio (HD/X)

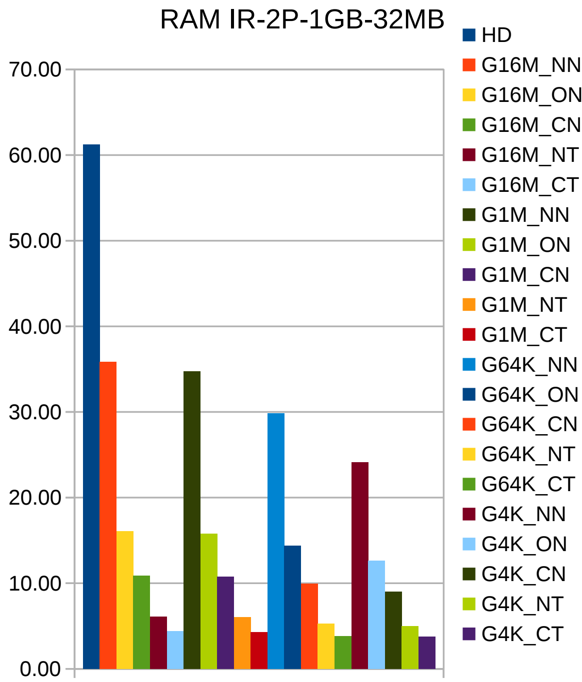
FIGURE 5.12 – RAMSMP 1 Gio, 2 processus, écriture, blocs de 32 Mio



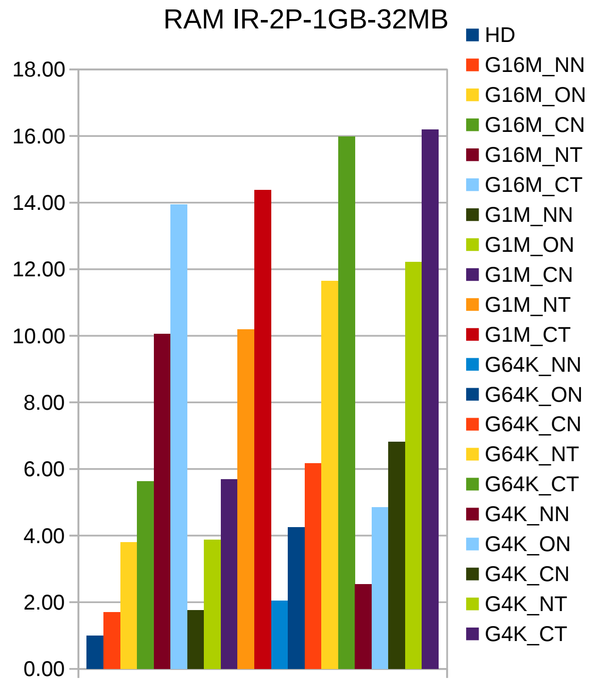
(a) Valeurs absolues (Mio/s)

(b) Ratio (HD/X)

FIGURE 5.13 – RAMSMP 1 Gio, 2 processus, lecture, blocs de 1 kio

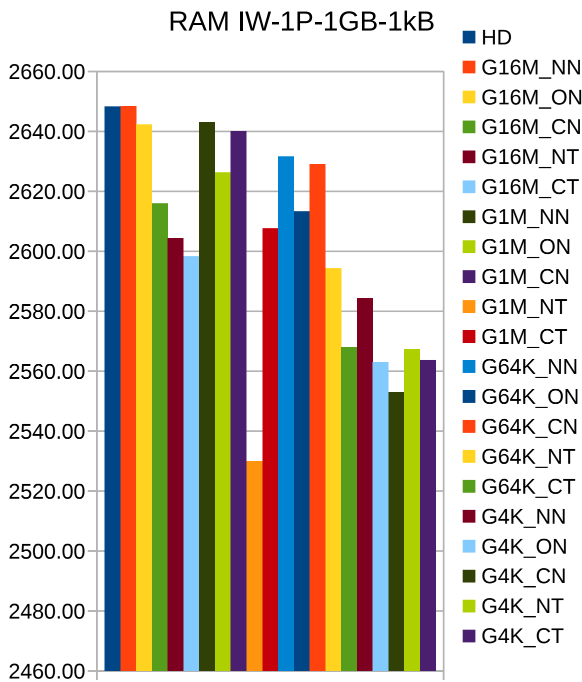


(a) Valeurs absolues (Mio/s)

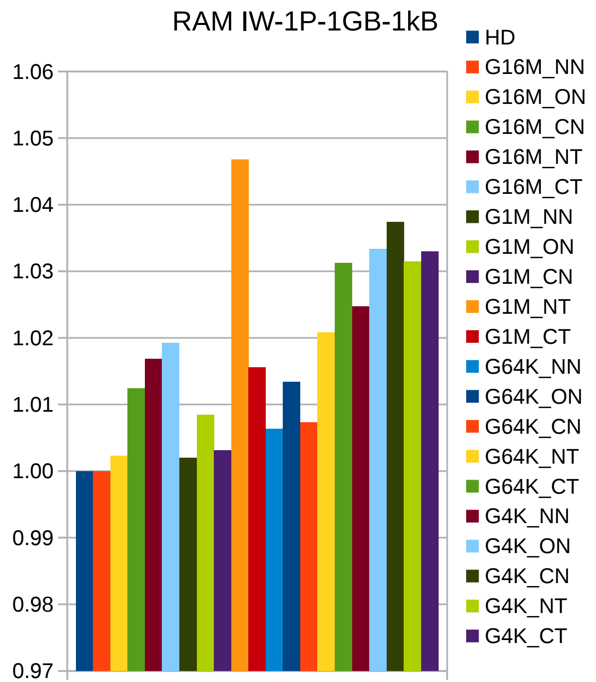


(b) Ratio (HD/X)

FIGURE 5.14 – RAMSMP 1 Gio, 2 processus, lecture, blocs de 32 Mio

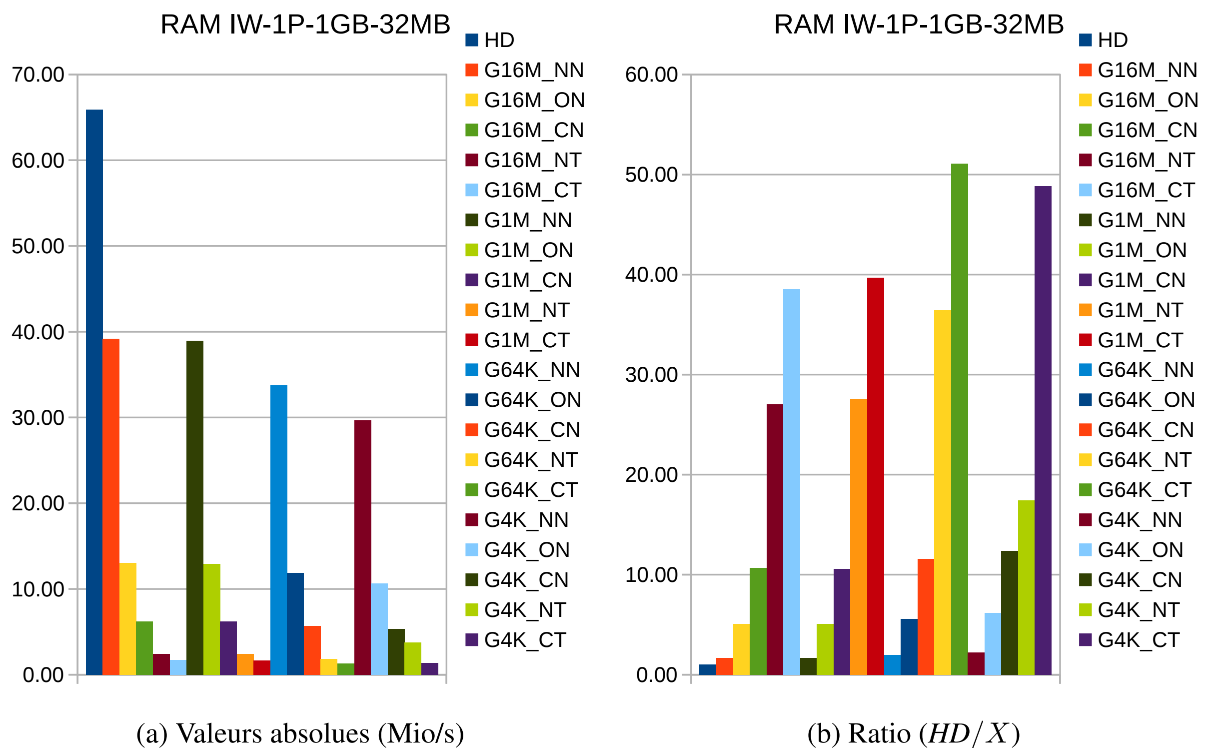


(a) Valeurs absolues (Mio/s)



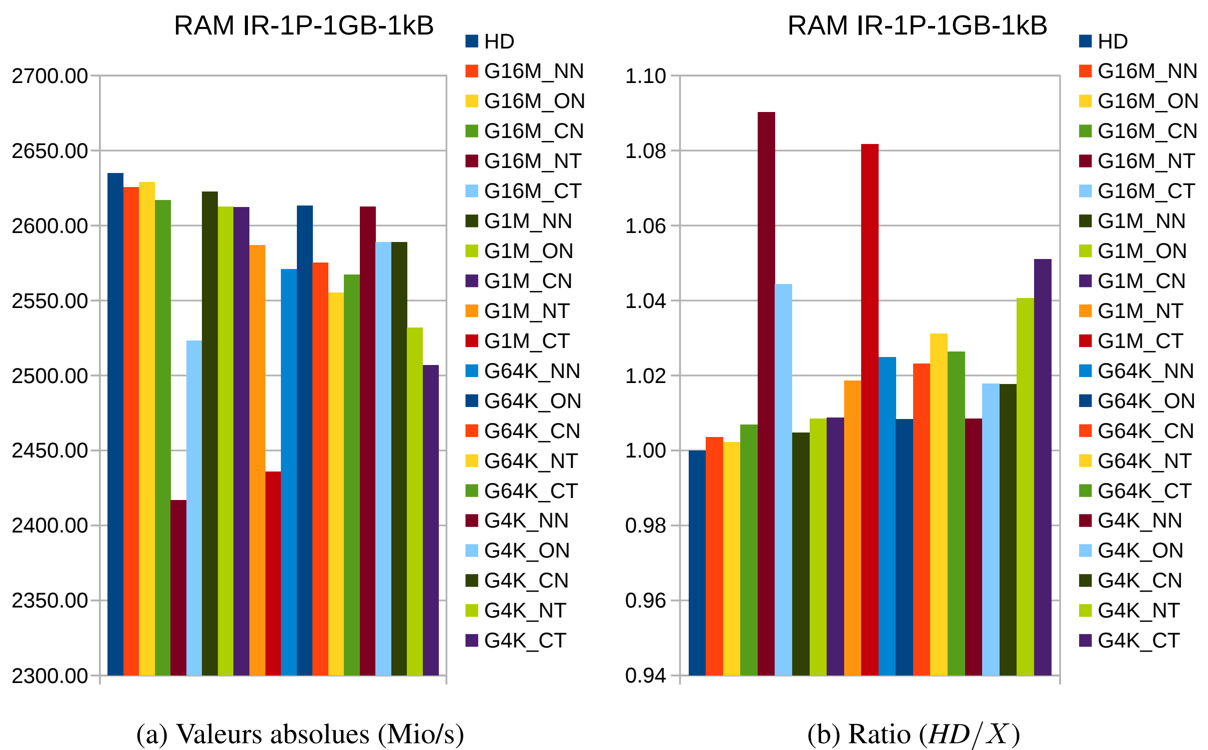
(b) Ratio (HD/X)

FIGURE 5.15 – RAMSpeed 1 Gio, 1 process, écriture, blocs de 1 kio



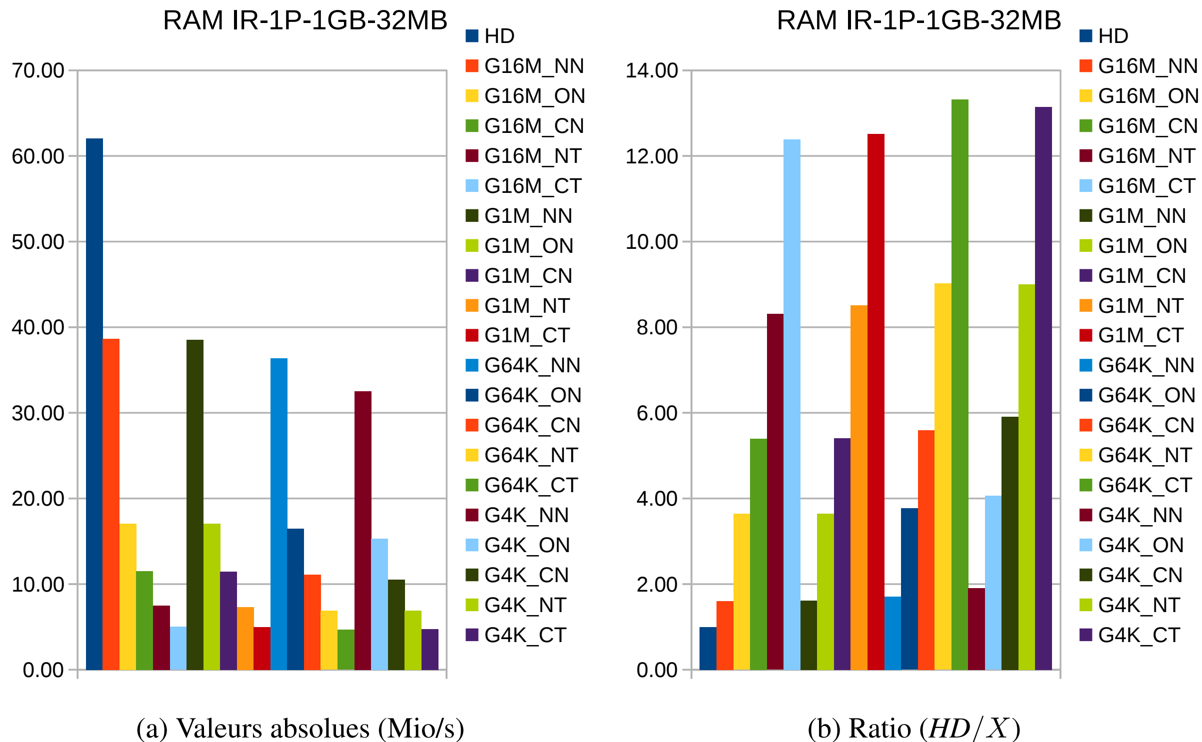
(a) Valeurs absolues (Mio/s) (b) Ratio (HD/X)

FIGURE 5.16 – RAMSpeed 1 Gio, 1 processus, écriture, blocs de 32 Mio



(a) Valeurs absolues (Mio/s) (b) Ratio (HD/X)

FIGURE 5.17 – RAMSpeed 1 Gio, 1 processus, lecture, blocs de 1 kio



(a) Valeurs absolues (Mio/s) (b) Ratio (HD/X)

FIGURE 5.18 – RAMSpeed 1 Gio, 1 processus, lecture, blocs de 32 Mio

5.3.5 Conclusion

Comme on peut le voir, les benchmarks Dhrystone et Whetstone montrent des différences négligeables entre les différentes configurations testées. Ceci est dû au fait qu'ils visent la mesure des performances d'une architecture en calcul sur des entiers et des flottants et donc qu'ils génèrent peu de transferts vers la mémoire. La plupart du temps, les données qu'ils utilisent rentrent dans les caches du CPU et / ou les caches du HSM.

Les temps de démarrage de Linux et du processus *init* sont plus intéressants car ils sont typiques d'une application qui récupère beaucoup de données et de code non cachés depuis la mémoire externe. Les performances sont ici fortement impactées car le processeur réalise de très nombreux accès mémoire qui transitent via le HSM. Les facteurs de ralentissement vont de $1,3$ à 50 en fonction des configuration. Logiquement, le pire cas correspond aux configurations *Gs_CT* (protection de la mémoire en confidentialité et en intégrité à l'aide d'arbres de MAC). Pour une politique donnée, la dégradation des performances est plus importante pour des petites pages que pour des grandes pages. En effet, pour ces dernières, le nombre de défauts de cache PSPE est logiquement plus faible.

Les benchmarks de bande passante mémoire montrent des résultats similaires : pour les petits blocs de données (1 kio) le coût est très limité (moins de $10\,\%$ dans le pire des cas) car la plupart des accès sont servis par les caches du CPU (et ne vont donc pas vers le HSM) et ceux qui ne le sont pas bénéficient fortement des caches locales du HSM. Pour les grands blocs (32 Mio), les caches du CPU et du HSM sont souvent mis en défaut. En fonction des configu-

rations, la bande passante mémoire est réduite d'un facteur $1,6$ à 90. Comme précédemment, la pire configuration est *Gs_CT* et pour une configuration donnée, la pénalité diminue avec la taille des pages protégées.

L'impact des mécanismes de protection de l'architecture SecBus sur les performances et sur l'empreinte mémoire peut sembler très important à la vue de ces résultats, voir dans certaines configuration, irréaliste. Néanmoins, il faut garder à l'esprit que la plupart des tests et simulations réalisés concernent un pire cas. Dans des situations réalistes où seules les ressources les plus sensibles sont protégés et avec des primitives cryptographiques les plus appropriés, ces coûts seraient probablement acceptables.

Conclusions et perspectives

Dans cette thèse, nous avons tout d'abord introduit un modèle de menace dans lequel un adversaire peut avoir accès aux données contenues dans un espace de stockage non fiable ou non digne de confiance (problème de confidentialité) et peut également les modifier (problème d'intégrité), et présenté les méthodes classiques de protection de la confidentialité et de détection de violations d'intégrité proposées dans la littérature dans le cadre de ce modèle, ainsi que plusieurs plates-formes de calcul sécurisé.

Par la suite, nous nous sommes concentrés sur les arbres de Merkle, une primitive qui offre une protection contre toute attaque qui menace l'intégrité des données et particulièrement les attaques par rejeu. Par contre, ils introduisent une dégradation significative des performances du système. Comme le montre l'évaluation des performances que nous avons présentée, toutes les opérations des arbres de Merkle sont concernées par cette dégradation. Afin d'atténuer cette pénalité nous avons présenté l'utilisation d'une mémoire cache dédiée qui est située dans la zone sécurisée (proche du composant de calcul) et qui contient les nœuds intermédiaires les plus fréquemment et ou récemment utilisés, ce qui réduit le nombre d'accès à la mémoire externe et le nombre de calculs de condensés cryptographiques.

Nous avons également proposé une solution supplémentaire, pour améliorer les performances des arbres de Merkle, basée sur des arbres de Merkle creux. L'utilisation des arbres de Merkle creux, avec l'une ou l'autre des deux variantes d'initialisation proposées, permet d'accélérer la phase d'initialisation de l'arbre par rapport aux arbres de Merkle réguliers, comme le montre l'évaluation des performances réalisée. La variante des arbres de Merkle creux initialisés permet d'accélérer la phase d'initialisation sans effet indésirable ultérieur sur les opérations de vérification ou de mise à jour. Elle utilise les mêmes opérations de cache que les arbres de Merkle réguliers. Pour la variante des arbres de Merkle creux non-initialisés, la phase d'initialisation est beaucoup plus rapide (presque instantanée) par rapport aux arbres de Merkle réguliers ou aux arbres de Merkle creux initialisés. Par contre, les accès de lecture et d'écriture dans la mémoire externe, après l'initialisation, prennent plus de temps et nécessitent plus de calculs de MAC.

Ensuite, nous avons présenté l'architecture SecBus dont l'objectif est de protéger la confidentialité et l'intégrité du contenu de la mémoire externe contre des attaquants ayant tout contrôle sur le matériel (à l'exception du SoC lui même) et sur les logiciels. Nous avons détaillé l'architecture logicielle et matérielle de SecBus, et plus particulièrement le module de sécurité matériel (HSM) qui est le responsable du chiffrement et de déchiffrement des données transférées avec les mémoires externes et de la gestion et la vérification de leur intégrité.

Puis, nous avons présenté les résultats des différentes simulations qui ont été menées pour tester les différentes variantes d'arbres de Merkle et pour obtenir des indications sur les performances des arbres de Merkle creux et de l'utilisation d'un cache dédié. Nous avons aussi appliqué une approche par preuve à SecBus sur certains aspects complémentaires en utilisant deux outils différents.

Enfin, nous avons testé le module HSM en utilisant une plate-forme de prototypage matérielle à base de FPGA Zynq de Xilinx. Nous avons exécuté une pile logicielle complète dans diverses configurations et utilisé plusieurs programmes de benchmark classiques. Les configurations évaluées diffèrent par la politique de sécurité monolithique appliquée à l'espace mémoire de la pile logicielle et par la granularité de la protection.

En résumé des contributions effectuées durant cette thèse :

- Étude poussée sur les arbres de Merkle en présentant les opérations d'initialisation, de vérification d'intégrité et de mise à jour, et évaluant leurs performances.
- Proposition de nouvelle variante d'arbres de Merkle : les arbres de Merkle creux.
- Analyse détaillée sur l'utilisation des mémoires caches avec les arbres de Merkle, en étudiant les différents scénarios possibles afin de préserver la cohérence des arbres de Merkle.
- Dans le cadre de SecBus, une participation au développement de plusieurs modules matériels de protection.
- Intégration des solutions proposées dans l'architecture de SecBus.
- Présentation Des différentes preuves fonctionnelles appliquées sur les algorithmes d'arbres de Merkle réguliers et creux.

Durant cette thèse, nous avons effectué des experimentations sur les différents méthodes d'arbres de Merkle creux en utilisant une plate-forme logicielle de modélisation et de simulation (SoCLib), une implémentation matérielle des arbres de Merkle creux dans une plate-forme matérielle de prototypage reste nécessaire, afin de compléter l'évaluation de manière plus réaliste. Aussi, un seul type de cache mémoire a été évalué dans nos expérimentations et avec une seule politique de remplacement. Il se peut d'être comme réference, mais il est recommandé d'évaluer d'autres types de caches avec d'autres politiques de remplacement.

Dans la validation fonctionnelle, une fois bien précisée la portée de l'approche, le travail réalisé montre la faisabilité de certaines des étapes exhibées lors de l'analyse et la décomposition du problème. Les preuves de principes, à un niveau abstrait sont sans doute faisables (que la protection par arbres de Merkle fonctionne, qu'il suffise de considérer les ascendants quand on accède à une cellule), bien qu'elles n'aient pas été menées pour une raison de temps. De même d'ailleurs, le passage des pseudos-algorithmes que nous avons traités à la réalisation hardware réelle pourrait poser quelques difficultés et demander quelque décomposition supplémentaires des problèmes à prouver. Si tout ceci a déjà fait l'objet de réflexions approfondies, c'est surtout l'objet d'éventuels travaux futurs complémentaires dont nous pensons qu'ils pourraient être pertinents et raisonnablement abordables au vu de l'expérience présentée de façon résumée ici.

Appendices

Code source de la preuve EasyCrypt

```
require import FSet.
require import FMap.
require import Int.
require import List.
require import Array.
require import Option.

type key.
type addr = int*int.
type data.
type memory = (addr, data) map.
type block = memory array.

const s : int.
const a : int.
const data_init: data.
const null: data.

op father:  addr -> addr.
axiom father_prop : forall(i:int, j:int), 0<=i<s /\ 0<=j /\
      father(i,j)=(i+1,j /% a).

op readb: block -> data.
op make : memory -> addr  -> block.
op MAC: key*addr*data  -> data.
op tree_val: data option -> data.

module M = {
  var m : memory
  var k : key
}.

(************* Procedures ************)

module M1 = {
```

```
  proc initialisation () : unit = {
  var stop1 ,stop2 ,i ,j ,xijb , mc;
    i=1;
    stop1 = false;
    while (!stop1) {
    j=0;
    stop2 = false;
      while (!stop2) {
      xijb = readb(make M.m (i,j));
      mc = MAC(M.k,(i,j), xijb);
      M.m.[(i, j)] = mc;
      if (j = (a^(s-i))-1)  stop2 = true;
      else j=j+1;
       }
     if (i = s)  stop1 = true;
     else i=i+1;
  }
}

proc initial() : memory = {
initialisation();
return M.m;
}

proc verifyread (i:int,j0:int,j1:int) : data  option  = {
    var u,v1,v0,xu0,xuv1,xuvb,mc,result , stop1 , stop2;
    u=i;
    v0=j0;
    v1=j1;
    mc=data_init;
    stop1 = false;
    stop2 = false;
    xuvb = readb(make M.m (u,(a*v0+v1)));
    result = M.m.[(u,(a*v0+v1))];
    mc =  MAC(M.k,(u+1,v0),xuvb);
    (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
    while (!stop1 && !stop2)
    {
      xuvb = readb(make M.m (u,(a*v0+v1)));
      xuv1 = M.m.[(u,v1)];
      if (xuv1 <> Some mc)  stop2 = true;
      mc =  MAC(M.k,(u+1,v0),xuvb);
      (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
      if (u = s) stop1 = true;
     }
     xu0 = M.m.[(s,0)];
```

```
      return (!(xu0 <> Some mc) && !stop2) ? (omap tree_val (Some result))
}

proc verifywrite (i:int,j0:int,j1:int,v: data) :unit = {
var u,v0,v1,xu0,xuvb,xuv1,mcold,mcnew,stop1,stop2;
    u=i;
    v0=j0;
    v1=j1;
    stop1 = false;
    stop2 = false;
    xuvb = readb(make M.m (u,(a*v0+v1)));
    mcold = MAC(M.k,(u+1,v0),xuvb);
    M.m.[(u,v1)] = v;
    M.m.[(u, (a*v0+v1))] = v;
    mcnew = MAC(M.k,(u+1,v0),xuvb);
    (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
    while (!stop1 && !stop2)
    {
      xuvb = readb(make M.m (u,(a*v0+v1)));
      xuv1 = M.m.[(u,v1)];
      if (xuv1 <> Some mcold)  stop2 = true;
      mcold = MAC(M.k,(u+1,v0),xuvb);
      M.m.[(u,v1)] = mcnew;
      M.m.[(u, (a*v0+v1))] = mcnew;
      mcnew = MAC(M.k,(u+1,v0),xuvb);
      (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
      if (u = s) stop1 = true;
    }
    xu0 = M.m.[(s,0)];
    if (xu0 <> Some mcold) M.m.[(s,0)]= mcnew;
}

proc write(i:int,j:int,v:data) : memory = {
verifywrite(i,j/% a, j-(a*(j /% a)),v);
return M.m;
}
}.

module M2 = {
 proc initialisation () : unit = {
  var stop1,stop2,i,j;
    i=1;
    stop1 = false;
    while (!stop1) {
    j=0;
    stop2 = false;
```

```
      while (!stop2) {
      M.m.[(i, j)] = null;
      if (j = (a^(s-i)-1))  stop2 = true;
      else j=j+1;
       }
     if (i = s)  stop1 = true;
     else i=i+1;
  }
}

proc initial() : memory = {
initialisation();
return M.m;
}

proc verifyread (i:int,j0:int,j1:int) : data  option  = {
    var u,v1,v0,xu0,xuv1,xuvb,mc,result, stop1, stop2;
    u=i;
    v0=j0;
    v1=j1;
    mc=data_init;
    stop1 = false;
    stop2 = false;
    xuvb = readb(make M.m (u,(a*v0+v1)));
    result = M.m.[(u,(a*v0+v1))];
    mc =  MAC(M.k,(u+1,v0),xuvb);
    (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
    while (!stop1 && !stop2)
    {
      xuvb = readb(make M.m (u,(a*v0+v1)));
      xuv1 = M.m.[(u,v1)];
      if ((xuv1 <> Some null) && (xuv1 <> Some mc))  stop2 = true;
      mc =  MAC(M.k,(u+1,v0),xuvb);
      (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
      if (u = s) stop1 = true;
     }
     xu0 = M.m.[(s,0)];
     return (!(xu0 <> Some mc) && !stop2) ? (omap tree_val (Some result))
}

proc verifywrite (i:int,j0:int,j1:int,v: data) :unit = {
var u,v0,v1,xu0,xuvb,xuv1,mcold,mcnew,stop1,stop2;
    u=i;
    v0=j0;
    v1=j1;
    stop1 = false;
```

```
    stop2 = false;
    xuvb = readb(make M.m (u,(a*v0+v1)));
    mcold =  MAC(M.k,(u+1,v0),xuvb);
    M.m.[(u,v1)] = v;
    M.m.[(u, (a*v0+v1))] = v;
    mcnew =  MAC(M.k,(u+1,v0),xuvb);
    (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
    while (!stop1 && !stop2)
    {
      xuvb = readb(make M.m (u,(a*v0+v1)));
      xuv1 = M.m.[(u,v1)];
      if ((xuv1 <> Some null) && (xuv1 <> Some mcold))  stop2 = true;
      mcold =  MAC(M.k,(u+1,v0),xuvb);
      M.m.[(u,v1)] =  mcnew;
      M.m.[(u, (a*v0+v1))] = mcnew;
      mcnew =  MAC(M.k,(u+1,v0),xuvb);
      (u,v0,v1) = (u+1, v0/%a, a*(v0 /% a));
      if (u = s) stop1 = true;
    }
    xu0 = M.m.[(s,0)];
    if (xu0 <> Some mcold) M.m.[(s,0)]= mcnew;
}


proc write(i:int,j:int,v:data) : memory = {
verifywrite(i,j/% a, j-(a*(j /% a)),v);
return M.m;
}
}.

(************ Proof Part ***********)

pred equiv_mem (m1 : memory) (m2:memory) =
  forall (j:int), 0<=j =>  m1.[(0,j)] = m2.[(0,j)].


lemma init : equiv[ M1.initialisation ~ M2.initialisation : ={M.m}
==>  equiv_mem(M.m{1})(M.m{2})].

proof.
proc.
while (={i} /\ equiv_mem M.m{1} M.m{2}).
auto.
while (={j} /\ equiv_mem M.m{1} M.m{2}).
auto; progress; smt.
auto.
```

```
auto.
qed.

pred encrypt_correct (m : memory, k: key, i:int, j:int) = mem (dom m) (i
   => m.[father (i,j)] = Some(MAC(k, (i,j), readb(make m (i,j)))).

lemma init_correct1: hoare [M1.initial: true ==> forall(i:int,j:int),
                                  0<=j /\ 0<=i<s => encrypt_correct res
M.k i j].

proof.
proc.
sp.
inline M1.initialisation.
sp.
while (stop1).
sp.
auto.
auto.
auto.
while (stop2).
auto;smt.
auto;smt.
auto;smt.
qed.

lemma init_correct2: hoare [M2.initial: true ==> forall(i:int,j:int),
                                  0<=j /\ 0<=i<s => encrypt_correct res
M.k i j].

proof.
proc.
sp.
inline M2.initialisation.
sp.
while (stop1).
sp.
auto.
auto.
while (stop2).
auto;smt.
auto;smt.
auto;smt.
qed.

equiv Read : M1.verifyread ~ M2.verifyread :
```

```
    = {M.k, M.m} /\ equiv_mem M.m{1} M.m{2}  /\ (forall(i:int,j:int),
      0<=j /\ 0<=i<=s /\ encrypt_correct M.m{1} M.k{1} i j /\ encrypt_corr
={res}.

proof.
proc.
sp.
wp.
while (={u}/\ u{1}<=s).
auto; progress; smt.
auto;smt.
qed.

lemma write1: hoare [M1.write:  true ==> forall(i:int,j:int),
                     0<=j /\ 0<=i<s =>  encrypt_correct res  M.k i j].

proof.
proc.
inline*.
sp.
seq 1 :(s<=u).
while (u<=s).
auto;smt.
auto;smt.
auto;smt.
qed.



lemma write2: hoare [M2.write:  true ==> forall(i:int,j:int),
                     0<=j /\ 0<=i<s =>  encrypt_correct res  M.k i j].

proof.
proc.
inline*.
sp.
seq 1 :(s<=u).
while (u<=s).
auto;smt.
auto;smt.
auto;smt.
qed.
```

Acronymes

AES Advanced Encryption Standard
NIST National Institute of Standard and Technology
DES Data Encryption Standard
OTP One Time Pad
ECB Electronic CodeBook
CBC Cipher-Block Chaining
CFB Cipher FeedBack
OFB Output FeedBack
IV Initialization Vector
RSA Rivest Shamir Adleman
PKI Public Key Infrastructure
MDC Modification Detection Codes
MAC Message Authentication Codes
CRHF Collision Resistant Hash Function
SHA Secure Hash Algorithm
SSL Secure Sockets Layer
TLS Transport Layer Security
IPSec Internet Protocol Security
CBC-MAC Cipher Block Chaining Message Authentication Code
SCA Side Channel Attacks
SoC System-on-Chip
Nonce Number used once
AMR Arbre de Merkle Régulier
AREA Added Redundancy Explicit Authentication
PAT Parallelizable Authentication Tree
TEC-Tree Tamper-Evident Counter Tree
XOM eXecute-Only Memory
MESA Memory-Centric Security Architecture

PE-ICE Parallelized Encryption and Integrity Checking Engine
RO Read Only
RW Read/Write
AMC Arbres de Merkle Creux
AMC-I Arbres de Merkle Creux Initialisés
AMC-NI Arbres de Merkle Creux Non Initialisés
ASAP As-Soon-As-Possible
ALAP As-Late-As-Possible
CAM Contrôleur d'Arbres de Merkle
CCAM Contrôleur de Cache d'Arbre de Merkle
CIP Cohérence Interne Partielle
DMA Direct Memory Access
GPU Graphics Processing Unit
HDD Hard Disk Drive
HSM Hardware Security Module
SSM Software Security Manager
SP Security Policies
PSPE Page Security Parameter Entry
MMU Memory Management Unit
MMT Master MAC Tree
ROM Read Only Memory
RTL Register Transfer Level
CABA Cycle Accurate / Bit Accurate
FSM Finite State Machine
VCI Virtual Component Interface
ELF Executable and Linkable Format
ISS Instruction Set Simulator
VGMN VCI Generic Micro-Network
FPGA Field Programmable Gate Array
AXI Advanced eXtensible Interface
LUT Look Up Table
RAM Random Access Memory
DSP Digital Signal Processor

Bibliographie

[1] “Bank customer data sold on ebay.” [Online]. Available : http://news.bbc.co.uk/2/hi/uk_news/7581540.stm

[2] M. G. Kuhn, “Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP,” in *IEEE Transaction on Computers*, vol. 47, no. 10. IEEE Computer Society, 1998, pp. 1153–1157.

[3] J. Stern, *La Science du secret*, O. Jacob, Ed., 1998.

[4] A. J. Menezes, P. C. v. O. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, C. Press, Ed., 1996.

[5] A. Kerckhoffs, “La cryptographie militaire,” in *Journal des sciences militaires*, 1883, pp. 5–38.

[6] ——, “La cryptographie militaire,” in *Journal des sciences militaires*, 1883, pp. 161–191.

[7] G. S. Vernam, “Secret signaling system,” in *United States Patent, US1310719A*, 1919.

[8] C. E. Shannon, “Communication theory of secrecy systems,” in *Bell system technical journal*, vol. 28. Blackwell Publishing Ltd, 1949, pp. 656–715.

[9] NIST, “Advanced encryption standard (aes),” in *Processing Standards Publication 197*, 2001.

[10] ——, “Data encryption standard (des),” in *Processing Standards Publication FIPS PUB 46-2*, 1993.

[11] E. F. Foundation, *Cracking DES : Secrets of Encryption Research, Wiretap Politics and Chip Design*. ACM, 1998.

[12] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede, “A 3.84 gbits/s aes crypto coprocessor with modes of operation in a 0.18 μm cmos technology,” in *Proceedings of the 15th ACM Great Lakes symposium on VLSI (GLSVLSI’05)*. ACM, 2005, pp. 60–63.

[13] NIST, “Recommendation for block cipher modes of operation,” in *Special Publication 800-38A*, 2001.

[14] M. E. Diffi, Whitfield et Hellman, “New directions in cryptography,” in *Transactions on Information Theory*. IEEE, 1976, pp. 644–654.

[15] A. method for obtaining digital signatures and public-key cryptosystems, “Rivest, r. l. et shamir, a. et adleman, l.” in *Communications of the ACM*. ACM, 1978, pp. 120–126.

[16] J. McLoone, M. et McCanny, “Fast montgomery modular multiplication and rsa cryptographic processor architectures,” in *Proceedings of the 37th IEEE Computer Society Asilomar Conference on Signals, Systems and Computers*. IEEE, 2003, pp. 379–384.

[17] NIST, “Secure hash standard (shs),” in *Federal Information Processing Standards Publication 180-2*, 2002.

[18] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Advances in Cryptology — CRYPTO’96*, vol. 1109. Stanford : Springer, 1996, pp. 104–113.

[19] P. C. Kocher, J. Joshua, and J. Benjamin, “Differential power analysis,” in *Advances in Cryptology — CRYPTO’99*, vol. 1666. San Francisco : Springer, 1999, pp. 388–397.

[20] E. Biham and A. Shamir, “Differential fault analysis of secret key cryptosystems,” vol. 1294. Santa Barbara : Springer, 1997, pp. 513–525.

[21] R. Elbaz, D. Champagne, and R. B. Lee, “The reduced address space (ras) for application memory authentication,” in *Information Security*. Taipei, Taiwan : Springer, 2008, pp. 47–63.

[22] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, “A parallelized way to provide data encryption and integrity checking on a processor-memory bus,” in *Proceeding of the 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 506–509.

[23] C. Fruhwirth, “New methods in hard disk encryption,” 2005.

[24] R. Elbaz, D. Champagne, C. Gebotys, R. Lee B., N. Potlapally, and L. Torres, “Hardware mechanisms for memory authentication : A survey of existing techniques and engines,” in *Transactions on Computational Science IV*, vol. 5430. Springer, 2009, pp. 1–22.

[25] R. Merkle, “Method of providing digital signatures,” 1982, uS Patent 4,309,569. [Online]. Available : http://www.google.com/patents/US4309569

[26] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, “Checking the correctness of memories,” in *Proceedings. 32nd Annual Symposium in Foundations of Computer Science*. San Juan : IEEE, 1991, pp. 90–99.

[27] W. E. Hall and C. S. Jutla, “Parallelizable authentication trees,” in *Selected Areas in Cryptography*. Springer, 2006, pp. 95–109.

[28] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, “Tec-tree : A low-cost, parallelizable tree for efficient defense against memory replay attacks,” in *Cryptographic Hardware and Embedded Systems - CHES’07*. Vienna, Austria : Springer, 2007, pp. 289–302.

[29] R. M. Best, “Microprocessor for executing enciphered programs,” United States Patent, Tech. Rep. US4168396, septembre 1979.

[30] ——, “Preventing software piracy with crypto-microprocessors.” IEEE Computer Society, février 1980, pp. 466–469.

[31] ——, “Crypto microprocessor for executing enciphered programs,” United States Patent, Tech. Rep. US4278837, juillet 1981.

[32] ——, “Crypto microprocessor that executes enciphered programs,” United States Patent, Tech. Rep. US4465901, août 1984.

[33] *DS5002FP Secure Microprocessor Chip*, Dallas Semiconductor, 2006, http://datasheets.maxim-ic.com/en/ds/DS5002FP.pdf.

[34] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, 2000, pp. 168–177.

[35] D. Lie, J. Mitchell, C. A. Trekkath, and M. Horowitz, "Specifying and verifying hardware for tamper-resistant software," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03)*, 2003, pp. 166–177.

[36] D. Lie, C. A. Trekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, pp. 178–192.

[37] J. Yang and Y. Gao, Lan et Zhang, "Fast secure processor for inhibiting software piracy and tampering," in *IEE Transactions on Computers*. IEEE, 2003, pp. 351–360.

[38] ——, "Improving memory encryption performance in secure processors," in *IEE Transactions on Computers*. IEEE, 2005, pp. 630–640.

[39] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis : Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th International Conference on Supercomputing (ICS'03)*, 2003, pp. 160–171.

[40] ——, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[41] D. Clarke, S. Devadas, M. van Dijk, and B. Gassend, "Incremental multiset hash functions and their application to memory integrity checking," in *Advances in Cryptology - AsiaCrypt 2003 : 9th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2003, pp. 188–207.

[42] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, 2003, pp. 295–306.

[43] G. Duc and R. Keryell, "CryptoPage : an efficient secure architecture with memory encryption, integrity and information leakage protection," in *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*. IEEE Computer Society, 2006, pp. 483–492.

[44] G. Duc, "Support matériel, logiciel et cryptographique pour une exécution sécurisée de processus," Ph.D. dissertation, École Nationale Supérieure des Télécommunications de Bretagne, 2007, http://enstb.org/~gduc/these/these.pdf.

[45] W. Shi, H.-H. S. Lee, C. Lu, and M. Ghosh, "High speed memory centric protection on software execution using one-time-pad prediction." Georgia Institute of Technology, juillet 2004.

[46] ——, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems." IEEE, septembre 2004, pp. 123–134.

[47] ——, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Special issue : Workshop on architectural support for security and anti-virus (WASSA'05)*. ACM, mars 2005, pp. 6–15.

[48] W. Shi, C. Lu, and H.-H. S. Lee, "Transactions on high-performance embedded architectures and compilers i," P. Stenström, Ed. Berlin, Heidelberg : Springer-Verlag, 2007, ch. Memory-Centric Security Architecture, pp. 95–115. [Online]. Available : http://dx.doi.org/10.1007/978-3-540-71528-3_7

[49] R. Elbaz, "Hardware mechanisms for secured processor-memory transactions in embedded systems," Ph.D. dissertation, Université de Montpellier II, 2006.

[50] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, and M. Bardouillet, "PE-ICE : Parallelized encryption and integrity checking engine," in *Proceedings of the 9th IEEE workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2006, pp. 141–142.

[51] L. Su, "Confidentialité et intégrité du bus mémoire," Ph.D. dissertation, Télécom ParisTech, 2010.

[52] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz, and R. Pacalet, "Secbus : Operating system controlled hierarchical page-based memory bus protection," in *Design Automation & Test in Europe (DATE 2009)*, 2009, pp. 570–573.

[53] "Soclib website : http://www.soclib.fr." [Online]. Available : http://www.soclib.fr

[54] "Vsi alliance virtual component interface standard version 2 (ocb 2 2.0)."

[55] "EasyCrypt," https ://www.easycrypt.info.

[56] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, vol. 6841. Springer, 2011, pp. 71–90.

[57] "Xilinx all programmable socs : http://www.xilinx.com/products/silicon-devices/soc.html." [Online]. Available : http://www.xilinx.com/products/silicon-devices/soc.html

[58] "Zedboard community-based web site : http://zedboard.org/." [Online]. Available : https://www.altera.com/products/soc/overview.html