République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieure et de la Recherche Scientifique
Université Djillali Liabès de Sidi-Bel-Abbès

Faculté des Sciences Exactes
Département d'Informatique

Thèse

Pour l'obtention du Diplôme de Doctorat en Sciences en Informatique
Option : Systèmes d'Information et de Connaissances (SIC)

Présenté par :
BELFEDHAL Alaa Eddine

Etude et Implémentation des Fonctions de Hachage Cryptographiques Basées sur les Automates Cellulaires

---

Directeur de thèse :
Pr. FARAOUN Kamel Mohamed
Professeur à l'université Djillali Liabès

Soutenu en 2015

Devant le jury composé de :

Dr. BOUKLI HACENE Sofiane	MCA, UDL de Sidi Bel Abbès	(Président)
Dr. AMINE Abdelmalek	MCA, Université de Saïda	(Examinateur)
Dr. KADRI Benamar	MCA, Université de Tlemcen	(Examinateur)
Dr. DEBBAT Fatima	MCA, Université de Mascara	(Examinateur)
Dr. KESKES Nabil	MCA, UDL de Sidi Bel Abbès	(Examinateur)
Pr. FARAOUN Kamel Mohammed	Professeur, UDL de Sidi Bel Abbès	(Directeur de Thèse)

Etude et Implémentation des Fonctions de Hachage Cryptographiques Basées sur les Automates Cellulaires

Alaa Eddine **B**ELFEDHAL

2015

Remerciements

Je voudrais tout d'abord exprimer mes plus profonds remerciements à mon encadreur Pr. FARAOUN Kamel Mohammed pour son accord d'être mon directeur de thèse et de sa disponibilité et son aide pendant toute la préparation de cette thèse.

Je remercie ainsi Dr. BOUKLI HACENE Sofiane le chef de département d'informatique de l'UDL de Sidi Bel Abbès pour avoir accepter de présider le jury, et pour ses conseils et son encouragement.

Je tiens aussi à remercier tous les membres de jury : Dr. AMINE Abdelmalek, Dr. KADRI Benamar, Dr. DEBBAT Fatima et Dr. KESKES Nabil, pour leur disponibilité et acceptation d'examiner et de rapporter mon travail.

Je remercie tout mes collegues de l'université de Sidi Bel Abbès et de l'université de Mascara, pour leurs encouragement et précieuses orientations pendant toute la période de l'élaboration de ce travail.

Je ne saurais oublier de remercier mes parents ainsi que mes frères et mes sœurs pour leur soutien moral, leurs encouragements et leur patience durant les étapes de réalisation de ce travail.

Enfin, Que tous ceux qui directement ou indirectement m'ont apporté leur aide, trouvent ici l'expression de mes sincères remerciements.

Résumé

Les fonctions de hachage cryptographiques sont des primitives importantes pour construire de nombreux systèmes de sécurité tels que les protocoles d'authentification et d'intégrité. Ils ont attiré un intérêt de recherche exceptionnel au cours des dernières années, en particulier après l'augmentation de nombre d'attaques contre les fonctions largement utilisés comme MD5, SHA-1 et RIPEMD. À la lumière de ces attaques, il est nécessaire d'envisager de nouvelles stratégies de désigne et de conception de fonctions de hachage. Parmi les stratégies prometteuses sont celles basées sur les systèmes dynamiques et en particulier sur les automates cellulaires, qui constituent une bonne approche pour construire des fonctions de hachage rapides et sécurisées en raison de leur comportement chaotique et complexe dérivé à partir de règles simples. Dans cette thèse, nous étudions l'utilisation des automates cellulaires comme base de construction de fonctions de hachage et nous proposons deux fonctions de hachage simples et efficaces basées sur les automates cellulaires programmables. La première est une fonction sans clé basée sur les automates cellulaires élémentaires et la deuxième est une fonction avec clé secrète basée sur les automates cellulaires avec mémoire et les automates 2D. Les résultats des tests statistiques ont montré que les fonctions proposées possèdent de bonnes propriétés cryptographiques, telles que la confusion, la diffusion et une grande sensitivité vers les changements d'entrée. En outre, elles peuvent être facilement implémentées via le logiciel ou le matériel, et elles fournissent des performances fonctionnelles très compétitives.

Mots clés : Fonctions de Hachage Cryptographiques, Systèmes chaotiques, Automates Cellulaires, Règles d'Automates Cellulaires, Tests Statistiques, Effet d'avalanche.

Abstract

Cryptographic hash functions are important blocks to build many cryptographic systems such as authentication and integrity verification protocols. They have recently brought an exceptional research interest, especially after the increasing number of attacks against the widely used functions as MD5, SHA-1 and RIPEMD. Hence, the need to consider new hash functions design and conception strategies is imposed. Among promising strategies are those based on dynamical systems and particularly cellular automata, which presents a good approach to build fast and secure hash functions, due to their chaotic and complex behavior derived from simple rules interaction. In this thesis, we study the use of cellular automata as built base for cryptographic hash functions, and we present two simple and efficient hash functions based on programmable cellular automata. The first proposed hash function is a keyless function built by using elementary cellular automata . The second one is a keyed function with a secrete key, based on cellular automat with memory and 2D cellular automata. The functions are evaluated using several statistical tests, while obtained results demonstrate very admissible cryptographic proprieties such as confusion, diffusion capability and high sensitivity to input changes. Furthermore, the hashing schemes can be easily implemented through software or hardware, and provide very competitive running performances.

Keywords: Cryptographic Hash Functions, Dynamical Systems, Cellular Automata, Cellular Automata Rules, Statistical Tests, Avalanche Effect.

Table des matières

Introduction **1**

I Etat de l'art **6**

1 Les fonctions de hachage cryptographiques **7**

1.1 Définitions 8

1.1.1 Fonction de hachage 8

1.1.2 Fonction de hachage cryptographique 9

1.2 Notions de sécurité des fonctions de hachage cryptographiques 11

1.2.1 Résistance aux attaques 11

1.2.2 Propriétés des fonctions de hachage cryptographiques 14

1.3 Conception des fonctions de hachage cryptographiques 16

1.3.1 Algorithme d'extension du domaine (Mode opératoire) 16

1.3.2 Fonction de compression 23

1.4 Applications des fonctions de hachage 31

1.4.1 Code d'authentification de message 31

1.4.2 Signature électronique (Hash-and-Sign) 32

1.4.3 Génération de nombres pseudo-aléatoires 33

1.4.4 Stockage de mots de passe 34

1.4.5 Schémas d'engagement 35

1.4.6 Protection des fichiers 36

1.4.7 Authentification par défi/réponse 36

1.5 Conclusion 37

2 Les fonctions de hachages MD-SHA **38**

2.1 Les fonctions "Messages Digest" 41

2.1.1 MD2 41

2.1.2 MD4 41
2.1.3 MD5 43
2.1.4 HAVAL 46
2.2 Les fonctions "Standard Hash Algorithm" 47
2.2.1 SHA-0 47
2.2.2 SHA-1 49
2.2.3 SHA-256 52
2.2.4 SHA-512 54
2.3 Les fonctions RIPEMD 55
2.3.1 RIPEMD-0 55
2.3.2 RIPEMD-128 58
2.4 Conclusion 60

3 Automates Cellulaires **61**
3.1 Définition 62
3.2 Les automates cellulaires élémentaires 63
3.2.1 Diagramme espace-temps d'un automate cellulaire 65
3.2.2 Classes d'automates cellulaires élémentaires 66
3.2.3 Automates cellulaires programmables 69
3.3 Automates cellulaires avec mémoire 69
3.4 Automates cellulaires à deux dimensions 2D 70
3.5 Automates cellulaires et cryptographie 73
3.6 Conclusion 75

4 Fonctions de hachage basées sur les Automates Cellulaires **76**
4.1 Proposition de Damgard 77
4.1.1 Description de l'approche 77
4.1.2 Attaque de Meier et Staffelbach 78
4.2 Proposition de Daemen et al. (CellHash) 80
4.3 Proposition de Mihaljevic, et al. 82
4.4 Proposition de Dasgupta et al. 84
4.5 Proposition de Cheol Jeon (CAH-256) 85
4.6 Proposition de Jamil et al. 86
4.7 Conclusion 88

II Contribution **89**

5 Proposition de fonctions de hacahges cryptographiques basées sur les ACs Programmables **90**

5.1 Proposition 1: Une fonction de hachage sans clé . . . 91

5.1.1 Schéma général . . . 91

5.1.2 La fonction de compression . . . 92

5.1.3 La fonction de transformation . . . 95

5.2 Proposition 2: Une fonction de hachage avec clé secrète . . . 97

5.2.1 Mode opératoire . . . 97

5.2.2 Automate cellulaire avec mémoire . . . 98

5.2.3 La fonction f . . . 99

5.2.4 La fonction g . . . 100

5.3 Conclusion . . . 101

6 Tests et évaluation de sécurité **102**

6.1 La première fonction . . . 103

6.1.1 Analyse de sécurité . . . 103

6.1.2 Test d'effet d'avalanche . . . 104

6.1.3 Test du critère d'avalanche strict . . . 104

6.1.4 Tests du comportement pseudo-aléatoire . . . 107

6.1.5 Résistance aux collisions (Test de pseudo-collision) . . . 109

6.1.6 Test de vitesse d'exécution . . . 110

6.1.7 Comparaison avec d'autres fonctions de hachage basées sur les ACs . . . 111

6.2 La deuxième fonction . . . 113

6.2.1 Sensitivité par rapport aux changements du message . . . 113

6.2.2 Distribution uniforme dans l'espace des hachés . . . 116

6.2.3 Sensitivité par rapport aux changements de la clé . . . 118

6.2.4 Test de collision . . . 118

6.2.5 Test du comportement pseudo-aléatoire . . . 120

6.3 Conclusion . . . 123

Conclusion **124**

Bibliographie **138**

Table des figures

1.1 Construction de Merkle-Damgard. 17
1.2 Multi-collision sur une fonction de hachage utilisant la construction de Merkle-Damgård. 19
1.3 Construction Wide-Pipe. 20
1.4 La construction éponge. 22
1.5 Une construction à base d'arbre. 23
1.6 Schémas de constructions basés sur des chiffrements par blocs. 26
1.7 Schéma de signature électronique. 33
1.8 Stockage sécurisé de mots de passe. 35

2.1 Une étape de la fonction de compression de MD4. 43
2.2 Une étape de la fonction de compression de MD5. 45
2.3 Une étape de la fonction de compression de SHA-0. 48
2.4 Une étape de la fonction de compression de SHA-1. 51
2.5 Une étape de la fonction de compression de SHA-2. 53
2.6 Une étape de la fonction de compression de SHA-512. 56
2.7 Une étape de la fonction de compression de RIPMED-0. 58
2.8 Une étape de la fonction de compression de RIPMED-128. 60

3.1 Diagramme espace-temps d'un automate à une dimension et à 2 états. 65
3.2 Diagramme espace-temps de l'AC élémentaire 90. 65
3.3 Diagramme espace-temps de l'AC 36 à partir d'une configuration aléatoire. 67
3.4 Diagramme espace-temps de l'AC 40 à partir d'une configuration aléatoire. 67
3.5 Diagramme espace-temps de la règle 30. 68

3.6 Diagramme espace-temps de l'AC 18 à partir d'une configuration aléatoire. 68
3.7 Diagramme espace-temps de l'AC 20 à partir d'une configuration aléatoire. 68
3.8 Un Automate Cellulaire à 2 dimensions. 71
3.9 Voisinage de Von Neumann et Voisinage de Moore. 71
3.10 Règles d'AC à 2 dimensions. 72
3.11 Exemple d'application de la règle 8. 72

4.1 Construction de Merkle-Damgard. 77
4.2 Générateur de nombres pseudo-aléatoires de Wolfram utilisé par Damgard. 78
4.3 Triangle déterminé par la configuration initiale. 79
4.4 Un exemple simple de cryptanalyse de la règle 30. 81
4.5 La fonction de compression de CAH-256. 86
4.6 Réseau Omega-Flip. 87

5.1 Schéma général de la fonction proposée. 92
5.2 Diagrammes espace-temps des 4 règles 30, 90, 105 et 150. 93
5.3 Diagramme espace-temps de l'AC Programmable. 94
5.4 La fonction de transformation T. 96
5.5 Mode opératoire de la fonction proposée. 98
5.6 Voisinage de l'AC avec mémoire. 98
5.7 Représentation visuelle de la fonction f. 100
5.8 Représentation visuelle de la fonction g. 101

6.1 Distribution des distances de Hamming. 105
6.2 La distribution expérimentale et la distribution théorique des distances de Hamming. 106
6.3 Comparaison en termes de distances de Hamming expérémentales avec d'autres fonctions de hachage basées sur les ACs. 113
6.4 Distances de Hamming. 116
6.5 Distribution expérimentale des distances de Hamming. 117
6.6 Distribution des hachés dans l'espace de hachage. 117
6.7 Distances de Hamming. 118
6.8 Nombre de bits inversés 119
6.9 Distribution de ω 120

Liste des tableaux

1.1 Complexité des attaques génériques sur les fonctions de hachage . . . 13
1.2 Approches pour la construction des fonctions de compression. 24

3.1 Les règles 30 et 2 d'ACs élémentaires. 64
3.2 Représentation de la règle 34680. 70
3.3 La forme FNA de quelques règles d'AC avec mémoire. 70

5.1 Logique du control de l'ACP. 94
5.2 La règle 65534 de l'AC avec mémoire. 99
5.3 Configuration de l'ACM programmable. 99

6.1 Résultats des tests "DIEHARD". 108
6.2 Résultats des tests "ENT". 108
6.3 Rangs et probabilités théoriques de collisions pour 12 et 14 bits (une fonction aléatoire)[ADS10]. 110
6.4 Rangs et fréquences de collisions pour 12 et 14 bits (la fonction proposée). 110
6.5 Comparaison en terme de résistance aux collisions. 111
6.6 Comparaison de vitesse contre les fonctions de hachage largement utilisées. 111
6.7 Comparaison en termes de performences de sécurité avec d'autres fonctions de hachage basées sur les ACs. 112
6.8 Distribution des distances de Hamming. 115
6.9 Comparaison en terme sensitivité par rapport aux changements du message. 116
6.10 Résultats des tests "DIEHARD". 121
6.11 Résultats des tests 800-22 de NIST. 122

Introduction

Une fonction de hachage est un algorithme déterministe, rapide et efficace acceptant un message de longueur arbitraire comme entrée et produisant un haché de taille fixe en sortie. Ces fonctions sont très utiles en informatique, notamment dans les domaines des bases de données et des réseaux. Elles permettent de manipuler de très grandes quantités de données tout en gardant un temps d'execution acceptable en ce qui concerne la recherche, le tri, et le contrôle d'intégrité de ces données.

Cependant, dans cette thèse, nous nous intéressons uniquement à une sous-catégorie de ces fonctions : les fonctions de hachage dites "cryptographiques". Celles-ci, ont la capacité d'assurer certaines propriétés de sécurité, ce qui favorise leur utilisation dans le domaine de la cryptographie et de la sécurité informatique. En particulier, on veut éviter à tout prix la situation où deux entrées distinctes produisent la même valeur de sortie par la fonction de hachage (situation appelée collision). Informellement, cela implique que toute modification, même très petite, du message d'entrée, doit engendrer des changements importants et imprévisibls dans l'empreinte en sortie (Effet d'avalanche).

La deuxième différence entre les fonctions de hachage ordinaires et les fonctions de hachage cryptographiques est que ces dernières doivent êtres non-inversibles (à sens unique). Une fonction de hachage ordinaire n'a pas l'obligation de vérifier cette propriété de sécurité. La même chose en ce qui concerne les collisions : pour une fonction de hachage ordinaire, on demande simplement qu'un grand nombre de messages arbitraires ne génère que très peu de collisions. Pour une fonction de hachage cryptographique, on demande qu'il soit impossible pour un attaquant de trouver une collision, cet attaquant "intelligent" pouvant choisir délibérément les messages et donc essayer de se placer dans un cas qui lui est favorable. Donc, l'idée de base des fonctions de hachage cryptographiques est qu'un haché sert d'une image compacte représentative d'une chaîne d'entrée, comme si la chaine d'entrée est identifiable de façon unique avec son haché (comme une empreinte digitale).

Ces fonctions de hachage cryptographiques sont utilisés avec d'autres algorithmes pour construire de nombreux systèmes et protocoles de sécurité, tels que les systèmes de protection des mots de passe, les codes d'authentification de message (MAC), les générateurs de nombres pseudo-aléatoires et de clés secrètes etc. Aussi, depuis l'apparition des algorithmes de chiffrement asymétriques et des schémas de signature numériques, l'utilisation des fonctions de hachage est devenue très importante. Avant sa transmission, un message haché puis signé permet d'assurer en réception, l'intégrité du message, l'authentification de sa source et la non répudiation de l'expéditeur, tout en gardant un temps d'éxecution raisonnable.

D'un point de vue structurel, les fonctions de hachage cryptographiques peuvent être classées en trois grandes catégories selon la nature des opérations utilisées dans leurs fonctions de compression internes. La première catégorie représente les fonctions de hachage basées sur le chiffrement symétrique par blocs et par flux, comme celles proposées dans [PGV93] et dans [BRS02]. La construction basée sur le chiffrement par blocs la plus utilisée est celle attribué à Davies et Meyer [MMO85]. La plupart des fonctions de la famille MD-SHA utilisent cette construction dans leurs fonctions de compression. La sécurité de telles constructions est basée sur la sécurité de l'algorithme de chiffrement sous-jacent.

La deuxième catégorie représente les fonctions basées sur des problèmes mathématiques difficiles, comme : le problème de logarithme discret [BRP07], le problème de la factorisation des grands nombres [CLS06] et le problème de la recherche de cycles dans les graphes expanseurs [CLG07]. Malgré que la sécurité de ces constructions soit prouvée mathématiquement (tant que $P \neq NP$), la plus part d'entre elles ne sont pas utilisées dans la pratique, a cause de leur forte consommation des ressources et des performances d'exécution très réduites.

Enfin, il y a la catégorie des fonctions de hachage dédiées, qui sont conçus spécialement pour le hachage. Parmi les fonctions dédiées, il y a ceux basées sur l'utilisation de cartes chaotiques, qui sont des fonctions qui présentent certains comportements chaotiques et imprévisibles ; des exemples de ces cartes comprennent : la carte logistique et la carte Chebyshev [DSL05].

Les automates cellulaires (AC) sont également utilisées pour concevoir des fonctions de hachage dédiées. Les ACs sont très utiles pour concevoir des fonctions de hachage sûres avec une basse complexité matérielle et logicielle en raison des attributs de leurs opérations logiques. Leur comportement chaotique, complexe et imprévisible (pour quelques règles) permet de concevoir des fonctions de hachage sécurisées et fiables. Les AC fournissent également un haut niveau de parallélisme

et par conséquent, ils sont en mesure d'atteindre des vitesses d'exécution très élevées.

Dans ce travail nous étudions les différentes approches de conception de fonctions de hachage cryptographiques, et nous proposons deux fonctions de hachage basées sur les automates cellulaires programmables : la première est une fonction sans clé, alors que la deuxième est avec clé secrète. Ensuite, nous testons leur sécurité, leur fiabilité et leurs performances.

Après cette introduction, cette thèse est divisée en deux grandes parties, qui sont l'état de l'art et la contribution.

L'**état de l'art** est présenté en trois chapitres :

Le **Chapitre 1** est une introduction générale au domaine des fonctions de hachage, il présente les fonctions de hachage cryptographiques, en commençant par une définition de leurs concepts de base et une description de leurs principes de fonctionnement. Ensuite le chapitre introduit les différentes notions, attaques et propriétés de sécurité des fonctions de hachages. Le chapitre détaille par la suite les différentes approches de construction de fonction de hachages présentes dans la littérature. A la fin il présente leurs multitudes applications dans le domaine de la sécurité informatique.

Le **Chapitre 2** représente une étude des fonctions de hachage de la famille MD-SHA (Messages Digest - Standard Hash Algorithm), qui constituent les standards en terme de fonctions de hachage cryptographiques. Dans ce chapitre, nous présentons leurs principes de fonctionnement et nous montrons les différentes attaques des cryptanalystes contre eux.

Dans le **Chapitre 3** nous présentons succinctement les principes et le fonctionnement des Automates Cellulaires (ACs). Ensuite nous définissons la notion d'ACs élémentaires, et nous montrons leurs différentes classes et les propriétés de chacune d'elles. Nous présentons aussi les ACs avec mémoires et les ACs à deux dimensions (2D). La dernière partie de ce chapitre est consacré aux applications des ACs dans la cryptographie.

Le **Chapitre 4** est un état de l'art sur la conception des fonctions de hachage à partir d'automates cellulaires. Ce chapitre expose les différentes approches et techniques de conception utilisées dans la littérature.

La **deuxième partie** concerne notre contribution. Elle est divisée en deux chapitres :

Le **Chapitre 5** expose la problématique, les objectifs de ce travail, ainsi que notre approche pour résoudre le problème de conception de fonctions de hachage

cryptographiques à base d'ACs. Avant de détailler, dans un deuxième temps, les principes de notre approche.

Le **Chapitre 6** est consacré à la description des tests qu'on a effectués pour valider nos propositions. Ce chapitre se termine par l'exposition des principaux résultats de notre travail.

Enfin, la **conclusion** synthétise les principales idées de notre proposition. Nous en profitons pour faire le point sur nos principales contributions et aussi sur les perspectives possibles de notre travail.

Cette thèse a fait l'objet de divers travaux écrits :

- Alaa Eddine Belfedhal, Kamel Mohamed Faraoun,"Fast and Efficient Design of a PCA-Based Hash Function", IJCNIS, vol.7, no.6, pp.31-38, 2015.DOI : 10.5815/ijcnis.2015.06.04
- Alaa Eddine Belfedhal and Kamel Mohamed Faraoun, Conception d'une Fonction de Hachage Cryptographique basée sur les Automates Cellulaires. CISC'2014, Jijel, Algérie, 2014.
- Alaa Eddine Belfedhal et Kamel Mohamed Faraoun, Une Fonction de Hachage Cryptographique basée sur les Automates Cellulaires Programmables. JEESI'14, ESI, Alger, Algérie, 2014.

Première partie

Etat de l'art

Chapitre 1

Les fonctions de hachage cryptographiques

Les fonctions de hachage sont des fonctions qui compriment une entrée de longueur arbitraire pour produire un résultat de longueur fixe. Si les fonctions de hachage répondent à d'autres exigences supplémentaires, elles deviennent un outil très puissant dans la conception des techniques de protection de l'authenticité et de l'intégrité des informations.

Dans ce chapitre nous allons introduire les principes et les concepts de base des fonctions de hachage. Dans une première partie, nous allons définir les fonctions de hachages cryptographiques en mettant en évidence la différence entre ces dernières et les fonctions de hachage ordinaires ("classiques"). La deuxième section présente les notions de sécurité des fonctions de hachage, les attaques génériques contre celle-ci (Attaque par collision, attaque sur la pré-image et attaque sur la deuxième pré-image), et les propriétés statistiques qu'elles doivent vérifier pour résister aux attaques. Ensuite, dans une troisième partie, nous détaillerons les principes de conception des fonctions de hachage à partir des modèles généraux d'algorithmes d'extension du domaine vers une taxinomie détaillée des approches de conception des fonctions de compression internes. Avant de conclure, nous discutons comment les fonctions de hachage peuvent être utilisées dans une multitude d'applications de sécurité.

1.1 Définitions

1.1.1 Fonction de hachage

Une fonction de hachage H (1.1) est un algorithme déterministe et efficace, qui prend comme entrée une donnée (un message) binaire M de taille quelconque, et produit à la sortie un haché h de taille fixe (en général entre 128 et 512 bits). Cet haché, appelé aussi "condensé" ou "empreinte" doit dépendre de tous les bits du message, et il est utilisé comme représentant comprimé de celui-ci [Bou12].

$$\begin{aligned} H : \{0,1\}^* \rightarrow \{0,1\}^n, \\ h = H(M). \end{aligned} \tag{1.1}$$

Les fonctions de hachage ont été inventées par H. P. Luhn dans les années 1950 [Knu81]. Elles sont très utiles en informatique, puisque l'empreinte peut servir à identifier le message sous une forme plus compacte. A l'origine, elles ont été créées pour faciliter la gestion des bases de données. A l'aide d'une structure de données appelée table de hachage, où les données sont identifiées par leurs hachés, on peut tester très efficacement si une donnée est déjà présente dans la base. Cela permet par

la suite d'optimiser des opérations comme le tri, l'insertion ou l'accès à un élément.

On peut aussi employer une fonction de hachage comme somme de contrôle (Checksum), quand on transfère des données sur un réseau : dans le cas où des erreurs de transmission se produisent, cela va éventuellement modifier la somme de contrôle, et on peut détecter l'erreur. Dans ces deux scénarios, il suffit d'utiliser une fonction très simple pour calculer une empreinte, par exemple l'addition (modulo 256) de tous les octets du message [Leu10].

1.1.2 Fonction de hachage cryptographique

Dans cette thèse, on étudie les fonctions de hachage d'une perspective cryptographique. La fonction ne doit pas seulement avoir un bon comportement quand les erreurs ou les modifications éventuelles sont aléatoires, mais on va considérer l'éxistance d'un adversaire (attaquant), qui va intercepter le message envoyé et introduit des modifications (erreurs) intentionnellement pendant la transmission.

Dans ce cas de figure, une fonction de hachage simple, telle que la fonction "Checksum", ne protège pas le message. Un attaquant "intelligent" peut facilement créer un deuxième message dont la somme de contrôle soit la même que celle du message original, simplement en ajustant la valeur du dernier octet. Dans cette situation, pour que la fonction de hachage soit utile, il doit être difficile de générer des collisions ou d'inverser la fonction [Fuh11] : un attaquant ne doit pas pouvoir construire un deuxième message avec la même empreinte que le message original. Une fonction de hachage résistante à des attaques de ce genre est appelée une fonction de hachage cryptographique.

Pour définir formellement une fonction de hachage cryptographique, il faut tout d'abord introduire quelques autres définitions :

Définition 1. Un algorithme est dit polynomial s'il s'exécute en temps polynomial par rapport à la taille de son entrée.

Définition 2. Soient A et B deux ensembles. Une fonction surjective $f : A \to B$ est dite à sens unique si, pour un $b \in B$ donné, il n'existe aucun algorithme polynomial permettant de trouver $a \in A$ tel que $f(a) = b$.

Définition 3. Soient A et B deux ensembles. Une fonction $f : A \to B$ est dite sans collisions s'il est impossible de trouver en temps polynomial $a, a' \in A$ telsque $a' \neq a$ et $f(a) = f(a')$.

Définition

Une fonction de hachage cryptographique $H(M) : \{0,1\}^* \rightarrow \{0,1\}^n$ est une fonction surjective qui transforme un message binaire m de longueur arbitraire en une empreinte h de longueur fixe ($H(m) = h$), avec les propriétés suivantes [Lan11] :

1. $H(m)$ doit être facile et rapide à calculer,
2. H doit être une fonction à sens unique,
3. H doit être sans collision.

Il est à noter que, comme l'ensemble des messages possibles est beaucoup plus grand que l'ensemble des hachés, il devrait toujours y avoir plusieurs exemples de messages m et m' tels que $H(m) = H(m')$. On exige toutefois, que trouver un de ces exemples soit difficile en pratique. La première et la troisième propriété se font en quelque sorte "concurrence".

En effet, si on n'exige pas que la fonction de hachage doit être facile à calculer, il serait, en théorie , possible de s'assurer que la résistance aux collisions soit toujours respectée. Il suffit, par exemple, de ne pas limiter la longueur du haché. Ainsi, cette longueur serait dépendante de la longueur du message en entrée. Toutefois, plus un message serait long et plus il nécessiterait de temps pour trouver son empreinte, ce qui ne serait pas utile. En pratique, il est souvent suffisant de limiter la résistance aux collisions en exigeant qu'il soit impossible, pour un message $m \in M$ donné,de trouver en temps polynomial un $m' \in M$ tel que $m' \neq m$ et $H(m') = H(m)$. Cette propriété reste toutefois la plus difficile à satisfaire lorsqu' on construit une fonction de hachage.

Résumé. Une fonction de hachage est une composante fondamentale dans presque tous les systèmes cryptographiques et dans plusieurs applications de sécurité. Les scénarios courants de son utilisation vont de la réduction de la taille des données à signer, au contrôle de l'intégrité d'un fichier en passant par l'authentification des utilisateurs à travers un réseau. Les fonctions de hachage calculent le condensé d'un message binaire, ce qui représente une chaîne de bits courte et de longueur fixe. Pour un message donné d'une longueur arbitraire, le condensé, ou le haché, est considéré comme son empreinte (c'est-à-dire, une représentation unique et compacte du message). Grâce à ces caractéristiques, les applications des fonctions de hachage en cryptographie sont multiples : ces fonctions sont une partie essentielle des systèmes de signature numérique et des codes d'authentification de message (MAC).

Les fonctions de hachage sont aussi largement utilisées pour d'autres applications, telle que par exemple, le stockage sécurisé des mots de passe ou la dérivation des clés.

Les fonctions de hachage cryptographiques peuvent êtres classifiées selon leurs paramètres d'entrée en deux catégories [MOV96] :

- **Fonctions de hachage sans clé sécrète (Keyless hash functions)** : Ces fonctions prennent comme entrée un seul paramètre (un message). Originalement, elles sont utilisées pour assurer le service de l'intégrité des données. En outre, leurs sorties sont chiffrées avec un algorithme de chiffrement asymétrique pour réaliser la signature numérique.

- **Fonctions de hachage avec clé sécrète (Keyed hash functions)** : Elles sont utilisées pour assurer à la fois l'intégrité et l'authentification des messages. Les fonctions de hachage avec clé, prennent deux entrées : un message et une clé secrète, pour produire une empreinte numérique dépendante de la clé et du message. Elles sont utilisées dans les protocoles de sécurité de la couche de transport comme le SSL et l'IPSEC (Internet Protocol Security), etc.

Les standards en terme de fonctions de hachage cryptographiques depuit 1993, étaient les fonctions de la famille SHA (SHA-0, SHA-1 et SHA-2), mais certaines faiblesses ont été découvertes pour cette serie de fonctions. Pour cela, le NIST [1] a lancé un concours public en 2008, afin de choisir une nouvelle fonction comme standard. La fonction Keccak [BDPA09] a gagné cette compétition en 2012, et est par la suite, devenue la nouveau standard SHA-3 [Bou12]. Dans le reste de ce travail on utilise le terme "fonction de hachage" pour désigner "fonction de hachage cryptographique".

1.2 Notions de sécurité des fonctions de hachage cryptographiques

1.2.1 Résistance aux attaques

Dans le domaine de la sécurité informatique, les fonctions de hachage sont utilisées comme brique de base de différents protocoles cryptographiques. La sécurité de ces protocoles est basée essentiellement sur la résistance de la fonction de hachage à différent types d'attaques. Pour pouvoir être utilisée en cryptographie, une fonction

1. National Institute of Standards and Technology (États-Unis) : www.nist.gov

de hachage doit résister à 3 types d'attaques : l'attaque sur la pré-image, l'attaque sur la seconde pré-image et l'attaque par collision. Bien que d'autres sortes d'attaque soient possibles, la sécurité des fonctions de hachage est souvent évaluée par les trois attaques précédents.

Attaques sur les fonctions de hachage

Attaque sur la pré-image. Dans cette attaque, on donne à l'attaquant un haché, et il doit trouver un message qui produit ce haché. Formellement : étant donné une valeur h de l'ensemble des empreintes, trouver un message m correspondant, tel que $H(m) = h$. La résistance à une telle attaque veut dire que la fonction doit être à sens unique (non-inversible).

Attaque sur la seconde pré-image. Dans une attaque sur la seconde pré-image, on donne un message à l'attaquant, et il doit trouver un autre produisant le même haché que le premier. Formellement : étant donné un message m et son haché $h = H(m)$, trouver m' différent de m tel que $H(m') = h$.

Attaque par collision. Dans ce type d'attaques, on demande à l'attaquant de trouver deux messages arbitraires ayant la même empreinte. Formellement : trouver deux messages quelconques différents m et m' tels que $H(m') = H(m)$.

Complexité des attaques génériques

Cette notion de résistance aux attaques coïncide avec l'idée d'une fonction injective à sens unique [Leu10]. Mais, comme l'ensemble des hachés est plus petit que l'ensemble des massages possibles, pour chacune de ces attaques, il existe des techniques permettant d'attaquer n'importe quelle fonction de hachage. Ces techniques (appelées "attaques génériques") sont basées sur la recherche exhaustive.

Donc la sécurité des fonctions de hachage cryptographiques repose sur la difficulté d'appliquer ces attaques dans la pratique.

Recherche de pré-images et de secondes pré-images. La meilleure attaque générique pour la recherche des pré-images est la recherche par force brute (recherche probabiliste) : étant donné une empreinte h, l'attaquant calcule des empreintes pour des messages aléatoires jusqu'à ce que l'une des empreintes soit égale à h. Chacun des hachés générés est égale à h avec une probabilité de 2^{-n} (si on considère que la fonction de hachage est une fonction aléatoire), donc le nombre moyen de hachés à calculer pour trouver une pré-image de h est de l'ordre de 2^n.

Notion de sécurité	Attaque générique	Complexité
Résistance à la pré-image	Recherche probabiliste	2^n
Résistance à la $2^{ième}$ pré-image	Recherche probabiliste	2^n
Résistance aux collisions	Paradoxe des anniversaires	$2^{n/2}$

Table 1.1 – Complexité des attaques génériques sur les fonctions de hachage

Recherche de collisions. Les fonctions de hachage sont utilisées pour calculer des condensés d'une longueur fixe à partir de données de taille arbitraire. L'ensemble de départ d'une fonction de hachage (l'ensemble des messages possibles) est donc plus large que l'ensemble des hachés. Ceci nécessite l'existence de collisions. La sécurité se base donc sur la difficulté de générer ces collisions dans la pratique.

La résistance aux collisions de n'importe quelle fonction de hachage (même aléatoire) est limitée par une propriété mathématique contre-intuitive appelée **"Paradoxe des anniversaires"**.

Selon le paradoxe des anniversaires, si on prend un groupe de 23 individus au hasard, il y a au moins une chance sur deux pour que deux d'entre eux, aient leur anniversaire le même jour. Cette propriété peut être démontrée par le raisonnement suivant [Fuh11] :

Soit m valeurs $x_1, ..., x_m$ tirées aléatoirement à partir d'un ensemble dénombrable de taille n. Il y a $\frac{(m^2-m)}{2}$ façons de choisir une paire (x_i, x_j) parmi $\{x_1, ..., x_m\}$. Pour chacune de ces paires, la probabilité pour que $x_i = x_j$ est égale à : $\frac{n}{n^2} = \frac{1}{n}$. Si $m^2 \geq n$, c'est-à-dire si $m \geq \sqrt{n}$, alors :

$$\sum_{1\leq i\leq j\leq m} Pr[x_i = x_j] \approx \frac{(m^2-m)}{2n} \approx \frac{1}{2} \tag{1.2}$$

De ce fait, pour le cas des anniversaires, on peut constater que pour $n = 365$ (nombre de jours de l'année), il suffit de prendre $m = \sqrt{356} \approx 23$, pour avoir une probabilité de collision supérieur à $1/2$.

Pour les fonctions de hachage, une attaque générique aux collisions consiste à calculer des empreintes pour des messages aléatoires, ce qui revient à effectuer un tirage uniforme sur l'espace des empreintes $\{0,1\}^n$, de cardinal 2^n. Selon la démonstration ci-dessus, le nombre d'empreintes qu'il faut générer pour produire une collision avec une probabilité supérieure à $1/2$ est de l'ordre de $\sqrt{2^n} = 2^{n/2}$.

Résumé. La complexité des meilleures attaques génériques est présentée dans le Tableau 1.1 [Fuh11].

Dimensionnement des fonctions de hachage

Quand on conçoit une fonction de hachage, on choisira n (la taille de l'empreinte) assez grand pour que ces attaques "exhaustives" soient impossibles dans la pratique. Les plus gros calculs "pratiques" effectués publiquement sont de l'ordre de 2^{64} opérations, alors, afin de fournir un certain intervalle de sécurité par rapport aux puissances de calcul actuelles, on cherche aujourd'hui à se protéger contre des attaquants pouvant calculer jusqu'à 2^{64} empreintes en un temps raisonnable. Lorsque la résistance à la recherche de collisions est exigée, cela oblige d'utiliser des fonctions permettant de calculer des empreintes dont la taille est supérieure à 128 bits (généralement au moins 160 bits).

Il est généralement l'objectif dans la conception d'une fonction de hachage cryptographique qu'aucune attaque ne peut être meilleure que les attaques par force brute. Ainsi, pour une bonne fonction de hachage, il sera impossible en pratique de trouver des collisions ou des pré-images. Ceci permet d'utiliser une fonction de hachage comme si elle était injective.

Relations entre les attaques sur les fonctions de hachage

De manière générale, une attaque en pré-image peut être employée en deuxième pré-image, simplement en supprimant le premier message, et en ne laissant que son empreinte. Aussi, une attaque en seconde pré-image peut être employée pour générer des collisions, en choisissant aléatoirement le premier message. Donc, il est claire que si une fonction est résistante aux attaques par collision alors c'est le cas pour les attaques en pré-image et en deuxième pré-image. Toutefois, ces conclusions ont des restrictions [Bou12] :

- Exploiter une attaque en pré-image pour générer des deuxièmes pré-images ne fonctionne pas dans toutes les situations, puisque une attaque en pré-image sur un haché particulier, peut donner toujours le même message.
- Une attaque en deuxième pré-image peut être utiliser pour générer des collisions, mais cela ne donnera une attaque exploitable, que si la complexité est inférieure à $2^{n/2}$ (complexité de l'attaque générique en collision). Le niveau de sécurité prévu n'est pas le même pour une attaque en pré-image (2^n) et une attaque en collision ($2^{n/2}$).

1.2.2 Propriétés des fonctions de hachage cryptographiques

Il n'est pas facile de prouver la résistance d'une fonction de hachage cryptographique aux attaques mentionnées précédemment, pour cela, on évalue habituelle-

ment la sécurité des fonctions de hachages avec des propriétés qu'on peut vérifier facilement à l'aide de tests statistiques, et qui donnent une bonne estimation de la résistance de la fonction aux attaques.

Comportement pseudo-aléatoire

La sortie d'une fonction de hachage idéale (parfaitement sûre) doit être indifférenciable de la sortie d'une fonction aléatoire. Une telle fonction est alors une fonction publique (principe de Kerckhoffs), mais sans structure exploitable [Leu10] : on ne doit pas pouvoir construire d'empreintes avec des propriétés spécifiques plus efficacement que sur une fonction aléatoire. Ce type de fonction idéale se base sur le modèle de "l'oracle aléatoire".

Le modèle de l'oracle aléatoire est une forme de calcul dans laquelle les utilisateurs ont accès à une boîte noire (oracle) qui renvoie une chaîne aléatoire pour toute nouvelle requête, mais renvoie la même valeur si on envoie plusieurs fois la même requête. Ce modèle a été introduit en cryptographie pour la première fois par Bellare et al. [BR93], qui proposent d'utiliser des fonctions de hachage pour remplacer l'oracle dans les applications pratiques.

Ce modèle est souvent utilisé dans la littérature pour prouver la sécurité des protocoles basés sur les fonctions de hachage [BR96]. Dans la pratique, l'oracle est instancié par une fonction de hachage, et la preuve assure qu'il n'existe pas d'attaque générique sur le protocole : une attaque doit exploiter une faiblesse interne de la fonction de hachage.

Le modèle de l'oracle aléatoire est une façon d'idéaliser une fonction de hachage. Mais, dans la pratique on ne peut pas concevoir une fonction de hachage déterministe avec un comportement aléatoire réel, pour cela on se contente de fonctions pseudo-aléatoires avec de bonne propriétés **statistiques**.

L'effet d'avalanche

L'effet d'avalanche est une propriété souhaitable pour les fonctions de hachage et les algorithmes de chiffrement. Cette propriété tente de refléter l'idée intuitive de la non-linéarité [CSS+05] : un changement minime dans l'entrée (inverser un seul bit) produit une perturbation significative de la sortie correspondante (la moitié des bits sont inversés). On demande généralement que chaque bit de la sortie dépend de chaque bit de l'entrée, de sorte que toute modification de l'entrée ait un effet important sur la sortie. Par conséquence, l'effet avalanche permet de rendre l'inversion de la fonction plus difficile grâce à ses propriétés chaotiques.

Formellement, si une fonction F possède l'effet d'avalanche, alors la distance de Hamming entre la sortie d'un vecteur d'entrée aléatoire, et la sortie d'un autre vecteur généré par l'inversement d'un bit aléatoire du premier vecteur, doit être en moyenne la moitié de la longueur de la chaine en sortie. Mathématiquement, une fonction $F : 2^m \to 2^n$ possède l'effet d'avalanche si [For90] :

$$\forall x, y | Hamming(x, y) = 1, moyenne(Hamming(F(x), F(y))) = \frac{n}{2} \qquad (1.3)$$

La diffusion des modifications sur les entrées est très importante. Si ce n'est pas le cas et que les sorties présentent un biais statistique, il serait possible d'établir des prédictions sur les entrées à partir de l'observation des sorties. L'effet avalanche est donc requis lors de la conception d'une fonction de hachage.

1.3 Conception des fonctions de hachage cryptographiques

La plupart des fonctions de hachage sont basées sur l'itération d'une fonction de compression qui traite un nombre fixe de bits. Le message à hacher est divisé en blocs d'une certaine longueur où le dernier bloc est éventuellement complété avec des bits supplémentaires (bits de bourrage).

Ce concept a été présenté pour la première fois par Michael O. Rabin en 1978 [Rab78]. Ainsi, une fonction de hachage peut être composée de deux processus différents. Le premier processus est souvent une fonction de compression ou une permutation, permettant de traiter un bloc de données de taille fixe. Le deuxième, appelé algorithme d'extension de domaine est un mécanisme reliant les sorties de la fonction de compression interne, d'une itération à une autre pour former le haché final.

1.3.1 Algorithme d'extension du domaine (Mode opératoire)

Dans la pratique, un message ou un fichier traité par une fonction de hachage peut être d'une taille quelconque. Cependant, concevoir un algorithme qui comprime des données de taille arbitraire directement, peut être une tâche très difficile. Une manière beaucoup plus simple de procéder est de diviser le message en blocs de taille fixe, ensuite de traiter les blocs un par un.

Un bon algorithme d'extension de domaine doit conserver les propriétés cryptographiques de la fonction de compression interne, et particulièrement la résistance

aux attaques. Nous allons voir dans ce qui suit les constructions les plus connues dans la littérature.

Construction de Merkle-Damgard

Le premier algorithme d'extension de domaine a été indépendamment proposé par Ralph Merkle [Mer90] et Ivan Damgard [Dam90] en 1989. Il est aujourd'hui connu sous le nom de la construction de "Merkle-Damgard".

Dans ce type de construction, une fonction de compression interne est appliquée de manière itérative. Le message M à haché, est divisé en blocs de longueur fixe $M_1, ..., M_k$. La fonction de compression C est appliquée aux blocs M_i l'un après l'autre (la sortie de la fonction C dans une itération, constitue avec le bloc du message suivant, l'entrée de la même fonction dans l'étape suivante) (voir Figure 1.1).

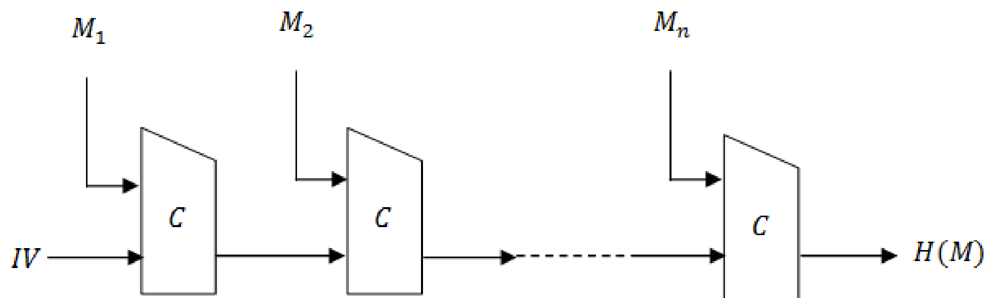

FIGURE 1.1 – Construction de Merkle-Damgard.

Dans un premier temps, un algorithme de remplissage "remp" injectif, appelé aussi bourrage (ou padding en anglais), est appliqué au message. Ceci consiste à ajouter une chaîne binaire à la fin du message, de façon que la taille finale soit multiple de la taille b d'un bloc. Si A et B sont deux messages différents arbitraires, il faut que les chaînes avec bourrage correspondantes soient différentes. L'algorithme "remp" doit être injectif pour empêcher les collisions [Bou12].

Il y a plusieurs manières pour rembourrer un message M, mais la plupart des fonctions connues, comme les fonctions de la famille MD ou SHA, utilisent l'algorithme suivant : tout d'abord, le bit '1' est ajouté au message, suivi d'une suite de '0'. Le nombre de '0' doit être le plus petit nombre tel que la taille de la chaîne finale obtenue vérifie : $t \equiv b - a \ mod \ b$, où a est une valeur constante, généralement égale à 64. Finalement, la longueur de M est codée en a bits et elle est concaténée au résultat :

$$remp(M) = M|10...00|t.$$

Après cette opération, $remp(M)$ est découpé en k blocs $M_1, M_2, ..., M_k$ de longueur b chacun. A chaque itération i, le bloc de message M_i et la valeur de chaînage h_{i-1} (la sortie de la fonction de compression précédente) constituent les paramètres de la fonction de compression C. La fonction C calcule alors la nouvelle valeur de chaînage :

$$h_i = C(h_{i-1}, M_i).$$

Pour les fonctions de hachage sans clé, h_0 est une valeur publique fixe, notée IV (Initial Value), (pour les fonctions avec clé elle représente la clé secrète). Après que tous les blocs du message soient traités, la valeur de chaînage finale h_k représentera l'empreinte du message M. Il est aussi possible d'appliquer une fonction de sortie à la dernière valeur de chaînage, afin d'obtenir la valeur de l'empreinte, néanmoins cette étape est souvent négligée en pratique.

L'algorithme de Merkle-Damgard est utilisé par la plupart des fonctions de hachage. Sa réussite est du à sa preuve de sécurité très simple et utile. En particulier, il est possible de prouver que si la fonction de compression est résistante aux collisions, alors c'est le cas aussi pour la fonction de hachage elle-même [Mer90].

Vulnérabilités de la construction Merkle-Damgard

L'attaque par extension de longueur. Cette attaque, qui est une des vulnérabilités importantes de la construction Merkle-Damgard, permet de générer à partir du haché $H(M)$ d'un message M, l'empreinte d'un message M' qui est suffixe de M, sans connaître le message lui-même.

Soient M et M' deux messages, tels que $remp(M)$ soit un préfixe de $remp(M')$. Il existe alors deux entiers a et b tels que :

$$remp(M) = M_1|M_2|...|M_a.$$

$$remp(M') = remp(M)|M_{a+1}|...|M_l$$

On peut maintenant construire l'empreinte de M' en fonction de $h_a = H(M)$ en calculant les variables de chaînage :

$$h_i = C(h_{i-1}, M_i) \text{ avec } i \geq a + 1.$$

Cette attaque qui ne menace pas la sécurité en pré-images ou en collisions des fonctions utilisant la construction Merkle-Damgard, pose pourtant un vrai problème pour les codes d'authentification (MAC) et pour les fonctions de hachage avec clé.

Plus précisément, à cause de cette faiblesse la construction MAC la plus naturelle, n'est pas sûre, obligeant l'utilisation de constructions plus complexes [Bou12].

Ce problème peut être résolu en ajoutant une fonction de sortie non-réversible à la fin du traitement [Luc05], ou un rembourrage dit "sans préfixe", pour lequel il n'existe aucun couple (m, m') tel que $pad(m)$ est un préfixe de $pad(m')$.

Multi-collisions. Une multi-collision est un ensemble de k messages différents ayant tous la même empreinte. La complexité de la génération des multi-collisions pour une fonction de hachage générique (vue comme une boite noire) qui produit des hachés de taille n, est $2^{\frac{n(k-1)}{k}}$. Ce résultat est la généralisation du paradoxe des anniversaires exécutée par A Joux. [Jou04]. Dans le cas de l'algorithme de Merkle-Damgård, Joux a présenté la technique suivante, représentée sur la figure 1.2, qui permet de générer des 2^k multi-collisions avec une complexité théorique de $k.2^{n/2}$ calculs de la fonction de compression. Donc la structure itérative de la construction de Merkle-Damgård permet de générer des multi-collisions avec une complexité anormalement faible.

Dans la méthode de Joux, on commence par chercher deux blocs de messages M_0^1, M_1^1 tels que $F(IV, M_0^1) = F(IV, M_1^1) = h_1$. D'après le paradoxe des anniversaires, cette étape requiert $2^{n/2}$ évaluations de F. A partir de h_1, on cherche ensuite deux blocs M_0^2, M_1^2 tels que $F(h_1, M_0^2) = F(h_1, M_1^2) = h_2$, ce qui coûte là encore $2^{n/2}$ opérations. On itère cette opération k fois, jusqu'à trouver deux blocs M_0^k, M_1^k tels que $F(h_{k-1}, M_0^k) = F(h_{k-1}, M_1^k) = h_k$. Supposons que H utilise le padding défini plus haut. On définit $M_{k+1} = 100...0|(mk)$ où (mk) est la représentation du produit $m \times k$ sur v bits, et $h = F(h_k, M^{k+1})$.

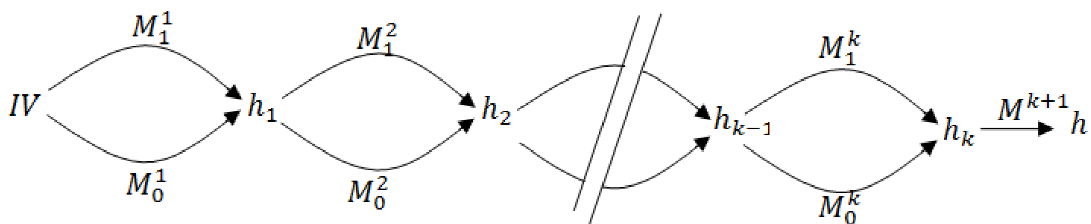

Figure 1.2 – Multi-collision sur une fonction de hachage utilisant la construction de Merkle-Damgård.

Soit $B = (b_1, ..., b_k)$ une suite de k bits quelconque. On definit alors $M_B = M_{b_1}^1|...|M_{b_k}^k$. On peut vérifier facilement que pour $H(M) = h$, il y a 2^k valeurs possibles pour B, qui définissent 2^k messages différents ayant la même empreinte. Cette méthode permet donc de générer des 2^k-multi-collisions. Sa complexité réside

dans les k étapes de recherche de collisions pour la fonction de compression. Elle nécessite donc en moyenne $k \times 2^{n/2}$ applications de F.

La construction Wide Pipe

Afin d'éviter les attaques par extension de longueur, les multi-collisions et d'autres vulnérabilités du mode Merkle-Damgard, Stephan Lucks a proposé dans [Luc05], un nouveau mode opératoire, appelé "Wide Pipe". La complexité des attaques mentionnées dépend de la taille de l'état interne. Dans la construction Merkle-Damgard, la taille de l'état est égale à la taille de l'empreinte, conduisant à des attaques plus efficaces que les attaques génériques sur une fonction idéale. La construction Wide Pipe, qui est assez similaire au mode Merkle-Damgard, permet de calculer des hachés de taille $h < n$ à partir de deux fonctions de compression F et G vérifiant :

$$F : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$$
$$G : \{0,1\}^n \rightarrow \{0,1\}^h$$

Le calcul de l'empreinte d'un message M consiste à appliquer la construction de Merkle-Damgard avec la fonction de compression F, puis à appliquer G au résultat pour obtenir une empreinte de taille h. Cette construction est présenté dans la figure 1.3.

La construction Wide-Pipe englobe plusieurs variantes, parmi lesquelles on peut noter les constructions double pipe, définie par $n = 2h$, et Chop-MD, pour laquelle la fonction G est une simple troncation. Cette dernière construction a été proposée par Coron et al. dans [CDMP05].

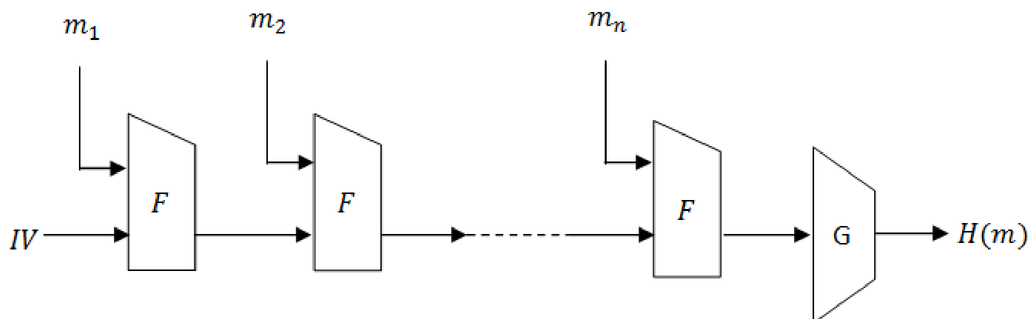

FIGURE 1.3 – Construction Wide-Pipe.

Résistance aux attaques. Nous pouvons constater que la construction Wide-Pipe résiste aux attaques par extension de longueur et à la recherche de multi-collisions décrites au paravent. En effet, étant donnée une valeur de haché x, il

existe en moyenne 2^{n-h} valeurs de y telles que $G(y) = x$. Si $n \geq 2h$, et $2^{n-h} \geq 2^h$ la connaissance de l'empreinte de M ne suffit pas à reconstituer avec certitude la valeur de la variable de chaînage après le traitement du message. Les attaques par extension de longueur ne peuvent donc pas s'appliquer.

L'algorithme de recherche de multi-collisions de Joux [Jou04] peut être utilisé dans le cas de la fonction Chop-MD. Cependant, la complexité de chacune des étapes de recherche de collisions sur la variable de chaînage est $2^{n/2}$. Si $n \geq 2h$ et $2^{n/2} \geq 2^h$ la complexité de la recherche de multi-collisions dépasse la complexité de l'attaque générique en recherche de pré-images. Le fait d'utiliser des variables de chaînage de plus grande taille que les empreintes permet donc de résister à la recherche de multi-collisions.

La Construction éponge (Sponge construction)

Une autre construction appelée construction éponge a été proposée par Bertoni et al. en 2008 [BDPA08]. C'est un algorithme itérée capable de produire des sorties de taille arbitraire. Contrairement à la construction Merkle-Damgard qui est basée sur une fonction de compression, la construction éponge repose sur une transformation interne de taille fixe, c'est-à-dire sur une permutation f, opérant sur les mots de b bits [Bou12].

L'empreinte d'un message m est calculée de la façon suivante : On commence par ajouter une suite de remplissage (padding injectif), puis on divise la chaîne obtenue en blocs $m_1, ..., m_k$ de taille t. Ensuite, les b bits de l'état interne sont initialisés avec la valeur '0'. La procédure se déroule en deux étapes successives (voir la Figure 1.4.) :

– **L'étape d'absorption :** Pendant cette phase, les blocs du message sont traités ("absorbés") de façon itérative. Le premier bloc m_1 est Xoré (OU exclusif) avec l'état interne. Le résultat est ensuite soumis à une transformation f. Puis, le deuxième bloc m_2 est Xoré avec l'état et la transformation f est de nouveau appliquée. Les mêmes opérations sont réitérées jusqu'à ce que tous les blocs de message soient traités.
– **L'étape de pressage :** Durant cette phase, des valeurs de l'état interne sont générés à des moments différents par des appels de la transformation f. La taille des emprientes extraites, peut être choisie selon les besoins.

La sécurité des fonctions éponges est reliée aux valeurs des paramètres internes de la construction, et plus spécifiquement à la capacité c définie par : $c = b - t$. On peut prouver que pour une fonction éponge qui génère des hachés de taille n, la résistance aux collisions est calculée par $min(2^{n/2}, 2^{c/2})$ évaluations de la fonction interne [BDPA08]. De cette façon, l'utilisateur peut choisir la valeur de c en fonction

du niveau de sécurité demandé ; plus la valeur de c est grande, plus le niveau de sécurité est élevé, mais plus les performances de la fonction sont réduites.

La construction éponge a été utilisée comme algorithme d'extension de domaine pour plusieurs fonctions de hachage récemment proposées. Parmi elles, on trouve le nouveau standard SHA-3 (Keccak) [Bou12].

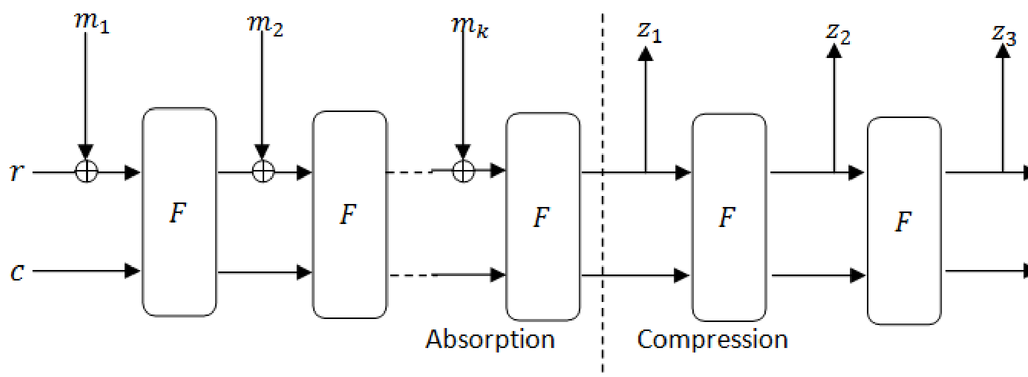

Figure 1.4 – La construction éponge.

La construction parallèle

La figure 1.5. illustre une construction parallèle typique sous forme d'arbre. C'est la classe de constructions la plus parallélisable, et elle est principalement adaptée à des plates-formes multi-core, où plusieurs processeurs peuvent fonctionner indépendamment et simultanément sur différentes parties du message.

Le principe de ce type de construction consiste à décomposer les données d'entrée en un ensemble de blocs de tailles identiques. Ils sont généralement complétés avec des valeurs neutres, comme des zéros, de façon à obtenir des blocs de la taille souhaitée. Ces blocs constituent les feuilles de l'arbre. Les nœuds du niveau suivant sont ensuite obtenus en compressant les nœuds du niveau courant à l'aide d'une fonction de compression jusqu'à n'obtenir qu'un seul nœud : la racine (l'empreinte).

Une des premières propositions de construction parallèle à base d'arbre est du à Damgard [Dam90]. Cette proposition a ensuite été optimisée par Sarkar et Scellenberg dans [SS01]. De même, Bellare et Rogaway [BR97] ont utilisé une approche fondée sur les arbres pour construire leur propositions de fonctions de hachage à sens unique "UOWHFs" (Universal One Way Hash Fonctions) qui, bien que plus faibles que les fonctions de hachage résistantes aux collisions, sont appropriés pour certaines applications.

Les constructions à base d'arbres sont moins populaires que les constructions

itératives, car elles ne sont pas aussi adaptées pour les plates-formes de bas de gamme tel que les cartes à puce, ce qui limite leur utilité.

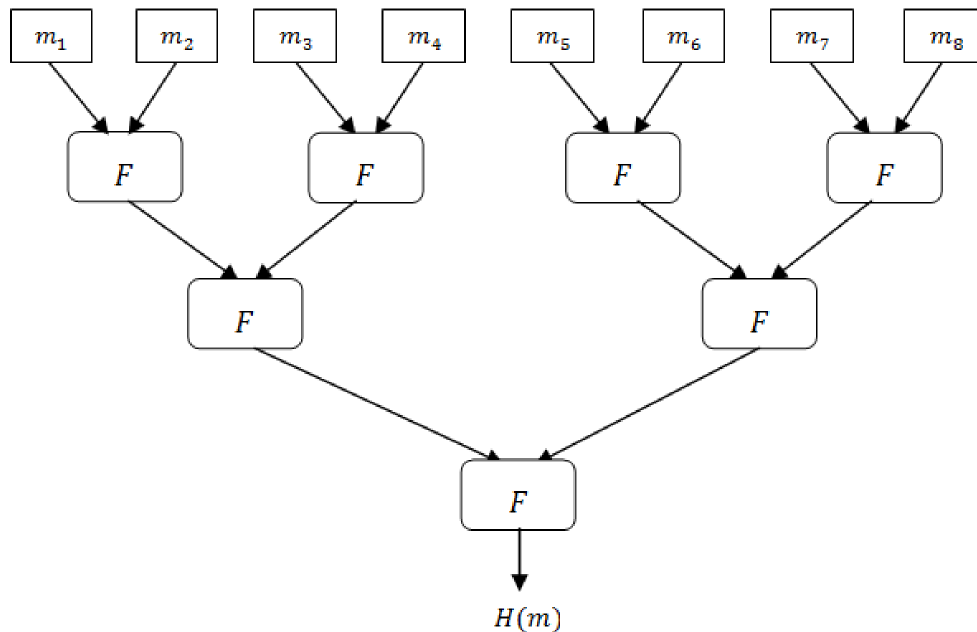

Figure 1.5 – Une construction à base d'arbre.

1.3.2 Fonction de compression

La compression de données est le premier objectif des fonctions de hachage. Une fonction de hachage doit comprimer la donnée en entrée, de telle sorte que la valeur de hachage résultante, peut être utilisée comme une empreinte pour identifier la donnée en entrée.

Pour cette raison, les fonctions de compression sont au cœur des fonctions de hachage. Leur construction est un problème complexe, mais il existe plusieurs manières pour y parvenir. On trouve dans la littérature plusieurs types de fonctions de compression, qui utilisent des méthodes et des techniques différentes. Certaines de ces méthodes dépendent des outils cryptographiques existants tandis que d'autres sont dérivées de problèmes mathématiques ou de systèmes chaotiques. Dans ce qui suit, Nous allons brièvement introduire les constructions les plus importantes. Le tableau 1.2. récapitule ces différentes approches, ainsi que leurs avantages et inconvénients.

Catégories	**Approches**	**Avantages/inconvénients**
Problèmes mathématiques difficiles	Logarithme discret, Factorisation des grand nombres, Théorie des graphes,...,	Sécurité prouvée/ Très lentes et consomment trop de mémoire Non pratiques
Algorithmes de chiffrement symétriques	AES, DES, Chacal,...	Rapides / Sécurité basée sur la sécurité des algorithmes de chiffrement.
Systèmes dynamiques	Théorie de chaos, Automates cellulaires,...,	Très rapides / Sécurité basée sur le comportement chaotique et imprévisible.

TABLE 1.2 – Approches pour la construction des fonctions de compression.

Basées sur des algorithmes de chiffrement par block

Construire une fonction de compression on se basant sur un algorithme de chiffrement par bloc est un concept assez intéressant, car les propriétés standard d'une fonction de hachage (résistance aux attaques sur les Pré-images et par collisions) peuvent être prouvées en supposant que la primitive de chiffrement est idéale. Mais, la motivation principale de l'utilisation d'un chiffrement par bloc comme fonction de compression est la minimisation des efforts de conception. En outre, si les opérations de chiffrement et de hachage sont utilisées ensemble, l'utilisation des mêmes algorithmes pour le chiffrement et le hachage permettra de réduire les coûts de mise en œuvre.

Les premières fonctions de hachage à base de chiffrement par blocs utilisent DES comme fonction de compression. Après la proposition de la construction Merkle-Damgård, ces fonctions de hachage ont devenu plus populaire. La plupart des fonctions de hachage largement utilisées, comme MD4, MD5, SHA-1 et SHA-2, utilisent un algorithme de chiffrement par blocs comme fonctions de compression (Ces fonctions seront étudier en détail dans le chapitre 2).

Dans ce qui suit nous allons montrer que la construction la plus intuitive de ce type présente des problèmes de sécurité évidents.

Une fonction de compression h prend en entrée une valeur de chaînage h_{i-1} et un message m_i et donne en sortie une nouvelle valeur de chaînage, h_i :

$$h(h_{i-1}, m_i) = h_i.$$

Nous voulons cependant que la fonction de compression ne soit pas inversible, c'est-à-dire la connaissance de h_{i-1} et h_i ne permet pas de calculer un message m_i. Si

E_k est un algorithme de chiffrement par bloc (avec une clé secrète k), la construction la plus simple pour arriver à cela consiste à utiliser le message m_i comme clé et à calculer :

$$h(h_{i-1}, m_i) = E_{m_i}(h_{i-1}) = h_i.$$

Toutefois, cette construction présente une faiblesse fondamentale [Bou12], qui nécessite d'utiliser une variable de chaînage 2 fois plus large que le haché, autrement dit un chiffrement par bloc ayant une taille de bloc $b = 2n$. En connaissant la valeur de h_i et en utilisant l'algorithme de décryptage D_k, avec un message m_i arbitraire comme clé, nous pouvons trouver facilement une valeur h_{i-1}. Cette vulnérabilité peut alors mener à une attaque en deuxième pré-image.

Il suffit pour cela de générer $2^{b/2}$ blocs de messages aléatoires, m, ensuite de calculer pour chacun d'eux $x = E_m(h_0)$. De même, nous choisissons $2^{b/2}$ autres blocs de message $m^{'}$ aléatoires et on calcule $y = D_{m'}(h_k)$. Avec une probabilité de $1/2$ nous trouvons alors une paire $(m, m^{'})$ tel que $E_m(h_0) = D_{m'}(h_k)$, ce qui signifie que $(m|m^{'})$ a la même empreinte que le message connu qui produit h_k.

Pour faire face à ce problème, des solutions plus complexes ont été proposées. Les premières propositions sont dû à Davies et Meyer et Matyas, Meyer et Oseas [MMO85]. Les constructions de ce type ont été étudiées par Preneel et al. [PGV93]. Dans une approche générale, les auteurs analysent toutes les fonctions simples de la forme :

$$h_i = h(h_{i-1}, m_i) = E_{x_1}(x_2) \oplus x_3$$

Où x_1, x_2 et x_3 sont des combinaisons linéaires de deux entrées h_{i-1} et m_i. Il existe alors, $2^{2^3} = 64$ schémas possibles. Parmi ces schémas, seulement 49 utilisent à la fois la variable de chaînage et le bloc de message. Pour ces 49 constructions, des faiblesses ont été démontrées pour 37 modes et la sécurité des 12 autres a été prouvée.

Ces douze constructions, sont appelées PGVi [Leu10], pour $1 \leq i \leq 12$. Parmi ces constructions, les trois plus utilisées, sont les schémas Davies-Meyer (PGV5), Matyas-Meyer-Oseas (PGV1) et Miyaguchi-Preneel (PGV7), qui sont présentés dans la Figure 1.6.

Dans la construction Davies-Meyer, chaque bloc de message m_i joue le rôle de la clé du chiffrement tandis que chaque valeur de chaînage h_{i-1} prend la place du bloc de message à chiffrer. Cette valeur est également additionnée au résultat du chiffrement :

$$h_i = E_{m_i}(h_{i-1}) \oplus h_{i-1}$$

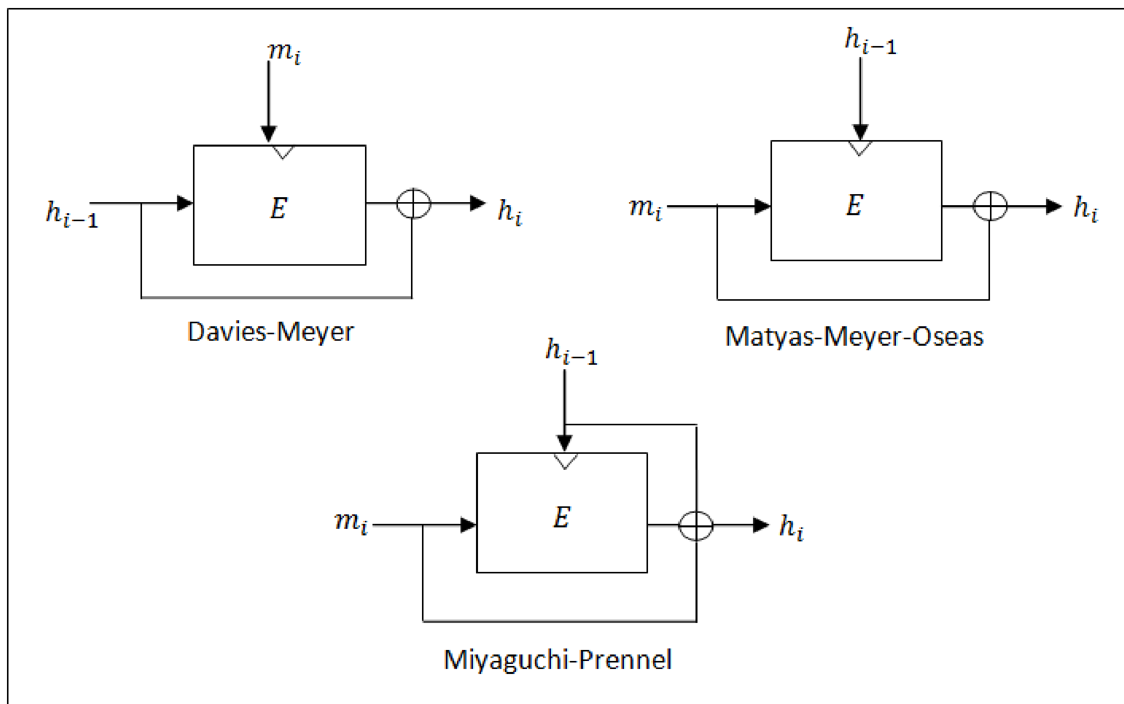

FIGURE 1.6 – Schémas de constructions basés sur des chiffrements par blocs.

La construction Davies-Meyer est utilisée pour construire plusieurs fonctions de hachage connues, comme les fonctions MD4, MD5, SHA-0, SHA-1 ainsi que la famille SHA-2.

Le schéma Matyas-Meyer-Oseas [MMO85] peut être vu comme le dual de la construction Davies-Meyer, car le rôle du message et de la valeur de chaînage sont inversés :

$$h_i = E_{h_{i-1}}(m_i) \oplus m_i$$

Finalement, le schéma Miyaguchi-Preneel, proposé par Shoji Miyaguchi et al. [MOI90] , est une variante généralisée de la construction Matyas-Meyer-Oseas :

$$h_i = E_{h_{i-1}}(m_i) \oplus m_i \oplus h_{i-1}$$

Il existe une forte similitude entre les fonctions de hachage et les algorithmes de chiffrement par blocs actuels. Cette similitude est due au fait que la construction de ces derniers est à ce jour considérée par la plupart des chercheurs comme mieux maîtrisée, et que de ce fait les concepteurs de fonctions de hachage ont cherché à s'en éloigner le moins possible [Pey08].

Basées sur des algorithmes de chiffrement par flux

Les fonctions de hachage à base de chiffrement par bloc ont des restrictions sur la taille de sortie. Pour satisfaire aux critères de sécurité de plus en plus exigeants, des fonctions de compression dont la taille de sortie est d'au moins le double de la longueur du bloc, doivent être généralement utilisées. Toutefois, cela diminue la vitesse et l'efficacité de la fonction de hachage.

Pour cela, plusieurs fonctions de hachage basées sur le chiffrement par flux ont été proposées dans la littérature. Ce type de constructions a été introduit pour des raisons de vitesse et d'efficacité.

La première fonction de hachage a base de chiffrements par flux est la fonction PANAMA [DC98], qui a été proposé en 1998 par Joan Daemen et al. PANAMA est basée sur un automate à états finis de 544 bits. L'automate emploie un registre à décalage à rétroaction linéaire de 8192 bits pour ses opérations [CSS+05]. Il travaille selon trois modes lors d'une itération :

- Push (introduction d'une entrée, pas de sortie)
- Pull (pas d'entrée, mais production d'une sortie)
- Blank (comme Pull mais la sortie est ignorée)

Le registre à décalage est mis à jour grâce aux Push et Pull commandés par l'automate fini. Le hachage se fait grâce à une série de Push dont les entrées proviennent du message à hacher. Une suite de Blank permet à l'algorithme d'assurer la diffusion des informations une fois arrivé à la fin du message.

De par son architecture, PANAMA est similaire à un grand nombre d'algorithme de chiffrement par flux, et diffère sensiblement des algorithmes basés sur la construction de Merkle-Damgård. Ces derniers (MD5, SHA-1, etc.) appliquent un grand nombre de tours sur un bloc, alors que PANAMA effectue un seul "tour" complexe mais parallèle. De plus, PANAMA a une variable interne d'un Kilo-octet de longueur, alors que la variable chaînage dans les fonctions de la famille MD-SHA a une taille égale à l'empreinte.

PANAMA a l'avantage d'être polyvalente, elle peut jouer le rôle d'une fonction de hachage, d'un chiffrement par flux ou encore d'un code d'authentification de messages, ce qui peut être intéressant pour les systèmes embarqués avec des capacités limitées, et aux critères de sécurité peu élevés.

Malheureusement, PANAMA a été cryptanalisée par ses concepteurs, qui ont présenté une attaque pratique sur la fonction de hachage. Cette attaque peut générer une collision en 2^6 évaluations de la fonction de mise à jour (Push). Joan Daemen et al. ont proposé ensuite une variante de PANAMA appelée RadioGatún [BDPA07].

En tant que fonction de hachage, RadioGatún n'a pas les faiblesses connues de PANAMA.

RC4-Hash [DKM06] est une autre fonction de hachage de cette famille. Elle est basée sur un algorithme de chiffrement largement utilisé, qui est le RC4. Cette fonction a aussi été cryptanalysée dans [OPS12].

Les fonctions de hachage basées sur les algorithmes de chiffrement par flux sont en général, plus rapides que celles basées sur les algorithmes de chiffrement par blocs. Mais la plupart de ces fonctions ont été cryptanalysées.

Basées sur des problèmes mathématiques

Une autre approche pour construire des fonctions de compression consiste à utiliser des fonctions issues des problèmes mathématiques difficiles, comme le problème de factorisation des grands nombres. En effet, certaines de ces constructions fournissent des preuves permettant de faire reposer leur sécurité sur un problème supposé difficile, pour l'une ou plusieurs des notions de sécurité habituelles (résistance à la recherche de collisions, de pré-images, de secondes pré-images...).

Cependant, les principaux inconvénients de cette approche sont la nécessité de posséder une très grande quantité de mémoire ou encore une vitesse d'exécution souvent lente à cause d'opérations algébriques très coûteuses.

La majorité des fonctions de hachage bien connus aujourd'hui traitent le message en mélangeant ses bits de manière à ce qu'ils remplissent les différentes exigences statistiques de sécurité, mais leur sécurité ne peut généralement pas être prouvé mathématiquement, car ils ne sont pas basés sur des modèles mathématiques.

D'autre part, les fonctions de hachage dont les propriétés de sécurité sont prouvées constituent une classe de fonctions basés sur des problèmes mathématiques, où des preuves mathématiques rigoureuses peuvent être dérivées, de telle sorte que si quelqu'un peut briser ces fonctions de hachage, il aura alors trouvé une solution à certains problèmes mathématiques difficiles [BP97].

Des exemples de ces fonctions comprennent : les fonctions de hachage basées sur le problème du logarithmique discret [BRP07], les fonctions basées sur le problème de factorisation des grands nombres [CLS06], les fonctions basées sur la recherche de cycles dans les graphes expanseurs (expander graphs) [CLG07], etc.

Pour construire des fonctions de compression, on peut utiliser d'autres problèmes aussi variés que la recherche du plus court vecteur dans un réseau [BPS+06], ou le problème de la résolution de systèmes d'équations quadratiques multi-variées dans un corps fini [BRP07].

Concevoir des fonctions de hachage basé sur le problème de sac à dos était aussi populaire dans les années 90. Bien que ces fonctions aient de bonnes performances pour les implémentations logicielles et matérielles, la plupart d'entre eux sont cassé [Pre93], ce qui a rendu cette approche de conception moins attrayante. Un exemple de telles fonctions de hachage est celle basée sur le problème de sac à dos additif proposée par Damgård [Dam90], mais cryptanalysé dans [Pat95]. Un deuxième exemple est la fonction LPS [CLG07] basée sur le problème de sac à dos multiplicatif, qui a été cryptanalysée dans [TZ08].

D'autres types de fonctions de hachage prouvablement sécurisés comprennent les fonctions basées sur les syndromes [AFS05], qui sont fondues sur un problème NP-complet connu sous le nom : "décodage du syndrome régulier". Cependant, même si les fonctions basées sur les syndromes sont prouvablement sécurisé, des résultats ont été publiés sur la cryptanalyse de plusieurs versions de ces dernières [CJ04].

Une fonction de hachage récente basée sur les syndromes est la fonction FSB [AFG+08], qui a été soumise à la compétition SHA-3, mais n'a pas réussi à progresser au deuxième tour de la compétition, principalement en raison de ses performances lentes et de sa consommation excessive de la mémoire, ce qui semble être le cas avec la plupart des fonctions de hachage basées sur des problèmes mathématiques.

Basées sur les systèmes dynamiques

Les systèmes dynamiques constituent une bonne approche pour concevoir des fonctions de hachages sûres et rapides à cause de leurs caractéristiques :

- Sensibilité aux conditions initiales : qui permet d'avoir un bon effet d'avalanche,
- Comportement chaotique et imprévisible : permet de concevoir des fonctions pseudo-aléatoires avec de bonnes propriétés de sécurité,
- La simplicité des opérations internes : ce qui implique la rapidité et la simplicité d'implémentation.

La théorie de chaos fournit un bon nombre de systèmes dynamiques chaotiques qui peuvent être utilisés comme base de construction de primitives cryptographiques. Ces systèmes sont appelé les "cartes chaotiques", qui sont des fonctions qui présentent certains comportements chaotiques ; des exemples de ces cartes comprennent : la carte logistique et la carte Chebyshev [DSL05].

Depuis les années 1990, de nombreuses primitives cryptographiques basée sur la théorie du chaos ont été proposées pour fournir des systèmes communications sécurisées [Ta91]. En général, la théorie du chaos a été prouvée comme base d'algorithmes

cryptographiques sécurisés et résistants aux techniques de cryptanalyse connues.

La première fonction de hachage basée sur la théorie de chaos a été proposée par Wong [Won03] en 2003. Dans son travail, Wong a combiné un crypto-système chaotique de chiffrement avec une fonction de hachage. Wong a proposé d'utiliser la valeur du traitement du dernier block comme valeur de hachage pour vérifier l'intégrité et l'authentification du message. Mais cette proposition a été cryptanalysée par G. Alvarez et al. dans [lGa04].

En 2005, une fonction de hachage basée sur une carte chaotique a été proposée par Xun Yi. [DSL05] ; cette fonction traite les messages de taille arbitraires pour produire des valeurs de hachage d'une longueur de deux fois la longueur des blocs du message. Plusieurs analyses statistiques de sécurité ont été appliquées, et Xun Yi a montré que la primitive de hachage proposé est assez forte contre plusieurs types d'attaques, et plus efficace que la fonction de Wang [Won03].

Xiao et al. ont proposés une nouvelle fonction de hachage avec clé basée sur une carte chaotique, avec des paramètres modifiables [DLD05]. Leur fonction utilise un système chaotique par morceaux d'une seule dimension, dans laquelle la clé secrète est une variable continue dans l'intervalle [0, 1]. L'algorithme accepte des messages avec des tailles différentes pour produire une valeur de hachage final de 128 bits. La valeur du haché dans l'algorithme de Xiao et al. est produite directement à partir de trois positions fixes dans la trajectoire chaotique. Par conséquent, cette fonction présente une faible résistance aux collisions quand l'espace de hachage n'est pas entièrement couvert par ces trois points.

En 2007, J. Zhang et al. ont proposé une fonction de hachage chaotique avec clé, basée sur un filtre numérique non linéaire [ZWZ07]. Ce filtre est utilisé comme un système dynamique chaotique avec une distribution uniforme, dans le quel, ils ont introduit un système de chiffrement par chaînage de blocs (CBC), afin accélérer la diffusion et la confusion. Ils ont affirmé que la fonction de hachage proposée présente de bonnes propriétés cryptographique et qu'elle est facile à implémenté via le matériel et le logiciel.

En 2008, une fonction de hachage parallèle avec clé a été proposée sur la base d'une carte chaotique linéaire par morceaux de 4 dimensions par Xiao, D et al. [DLD08]. En 2009, un autre groupe de chercheur a proposé une amélioration sur le schéma de hachage d'origine pour fournir une sécurité plus élevée [Z.11].

Wang et Zhao ont proposé une autre fonction de hachage parallèle avec clé basée sur un réseau neuronal chaotique [DLW09]. Un an plus tard, la sécurité de l'algorithme proposé a été analysée dans [WZ10]. En 2011, Huang a analysé la vulnérabilité du système de hachage et a proposé une amélioration pour éliminer les

problèmes de sécurité [Z.11].

En 2010 [DSL10], Xiao et al. ont proposé une autre fonction de hachage basée sur une carte chaotique linéaire par morceaux. Cette fonction prend en charge un mode de traitement parallèle pour fournir un rendement plus élevé. Le message d'entrée est divisé en blocs de 2048 bits et chaque bloc est inséré dans une table de consultation de 256 blocs. La carte chaotique est ensuite itérée pour traiter la table de consultation. Les valeurs de cette table sont regroupées en 16 colonnes et 16 lignes. La valeur finale de hachage est le résultat de l'opération XOR entre les 16 colonnes et les 16 lignes. En 2012, la sécurité de l'algorithme proposé a été analysée dans [WLZ12].

Les automates cellulaires (AC) sont également des systèmes dynamiques utilisés pour concevoir des fonctions de hachage. Les automates cellulaires sont très utiles pour concevoir des fonctions de hachage sûres avec une basse complexité matérielle et logicielle en raison des attributs de leurs opérations logiques. Les ACs fournissent également un haut niveau de parallélisme et par conséquent, ils sont en mesure d'atteindre des vitesses d'exécution très élevées [SAN09]. Les approches utilisant les ACs pour construire des fonctions de hachage sont discutées en détail dans le chapitre 3.

1.4 Applications des fonctions de hachage

Les fonctions de hachage sont utilisées par de nombreux systèmes cryptographiques, à la fois en cryptographie symétrique et en cryptographie asymétrique. Nous présentons ci-dessous une liste non exhaustive de ces systèmes.

1.4.1 Code d'authentification de message

Les codes d'authentification de message (ou MAC) sont des empreintes d'une donnée qui dépendent d'une clé secrète k, et qui ne peuvent être calculées qu'en connaissant k (c'est l'équivalent symétrique d'une signature). En ce sens, ils peuvent être vus comme des fonctions de hachage à clé secrète.

Un MAC doit être difficile à forger c-à-dire qu'un attaquant ne doit pas pouvoir calculer de MAC sans connaître la clé. Une façon simple pour construire un MAC est de faire rentrer le message et la clé dans une fonction de hachage : $MAC_k(M) = F(k|M)$: Cette construction s'appelle Secret-Prefix MAC, elle est sûre si la fonction de hachage se comporte bien comme une fonction aléatoire, puisqu'on ne peut pas prévoir la valeur de la fonction en un point en connaissant seulement sa valeur en

d'autres points. Cependant, comme on l'a vu dans la section 1.3.1.(L'attaque par extension de longueur contre la construction Merkle-Damgard), des fonctions très utilisées comme MD5 ou SHA-1 ne peuvent pas être utilisées de cette façon pour construire un MAC. Pour construire un MAC à partir d'une fonction de hachage, on utilise donc des constructions plus complexes, comme HMAC [BCK96]. HMAC repose sur l'utilisation de deux constantes p_1 et p_2, et sur une fonction de hachage H. Le MAC d'un message M avec une clé K est défini par [Leu10] :

$$MAC_k(M) = H(k \oplus p_1 | H(k \oplus p_2 | M))$$

1.4.2 Signature électronique (Hash-and-Sign)

Les schémas de signature sont sans doute l'application la plus importante des fonctions de hachage. Ils permettent à un utilisateur de signer un message à l'aide de sa clé privée [Pey08]. Chacun peut vérifier la validité de cette signature grâce à la clé publique correspondante.

Des schémas de signature comme RSA ou ElGamal permettent d'authentifier un message, mais ils demandent des calculs complexes et coûteux, car ils appartiennent à la cryptographie asymétrique. Ainsi, leur application à un très long message demande un temps de calcul trop grand dans certains cas. En pratique, au lieu d'appliquer un schéma de signature directement à un long message, on applique la signature à un haché du message. Ainsi, l'opération de signature est faite sur un identifiant de petite taille et elle sera moins coûteuse.

Si on veut que le signataire ne puisse pas répudier ses signatures (la même signature pour plusieurs massages), il faut que la fonction de hachage soit résistante aux collisions.

Si on veut qu'un attaquant susceptible d'obtenir les signatures de certains messages choisis soit incapable de créer une nouvelle signature valide sans connaître la clé privée de l'utilisateur, il faut que la fonction soit résistante aux attaques des secondes pré-images. Aussi, étant donnés plusieurs couples message/signature, il doit être impossible pour lui de deviner la moindre information sur la clé privée. Par « impossible », nous entendons que le meilleur moyen possible pour un attaquant est d'utiliser une attaque générique, c'est-à-dire indépendante du fonctionnement interne de la primitive. Il est donc important que les fonctions de hachage soient résistantes à la recherche de collisions ou de préimages. Par exemple, la connaissance d'une collision pour la fonction de hachage permettrait de calculer deux messages distincts aboutissant à la même signature : en demandant la signature du premier message, on pourrait en déduire celle du deuxième message.

Il est théoriquement possible de construire des mécanismes de signature permettant de signer des données de taille quelconque, cependant de telles constructions seraient très lentes. Pour pouvoir signer des données de taille quelconque, la solution la plus répandue consiste donc à leur appliquer une fonction de hachage, puis d'appliquer la fonction de signature S sur l'empreinte obtenue (voir la Figure 1.7). La signature d'un message M est alors $S(H(M))$.

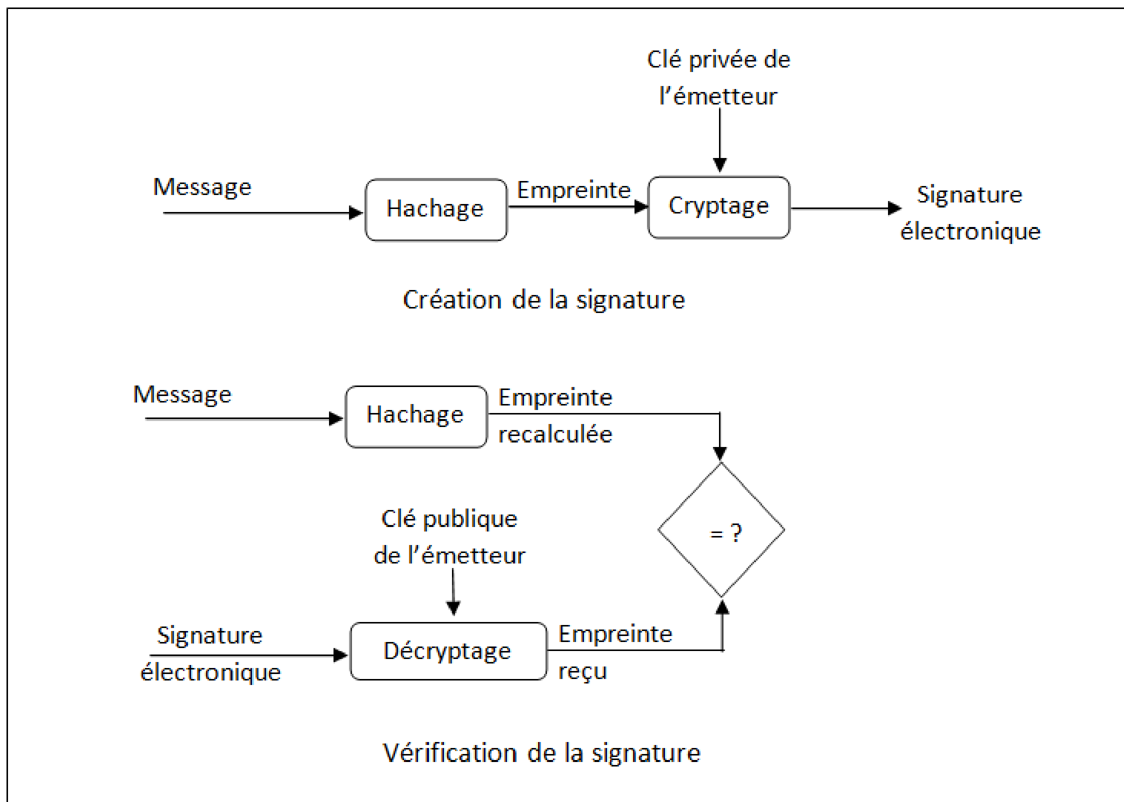

FIGURE 1.7 – Schéma de signature électronique.

1.4.3 Génération de nombres pseudo-aléatoires

Les propriétés statistiques des fonctions de hachage et leur caractère à sens unique peuvent être exploités dans des contextes de génération de nombres aléatoires [Fuh11]. En effet, de nombreux mécanismes cryptographiques nécessitent la génération de nombres aléatoires : génération de clés, signature électronique, chiffrement asymétrique, génération de valeurs d'initialisation pour le chiffrement symétrique... Les nombres ainsi générés doivent être tirés uniformément et être imprédictibles pour un attaquant.

Les nombres aléatoires sont générés à partir de secrets stockés en mémoire et/ou de phénomènes physiques aléatoires (seed). La traduction en chaînes de bits de phé-

nomènes physiques aléatoires peut conduire à des chaînes non uniformément distribuées. Dans ce cas l'utilisation de fonctions de hachage permet d'extraire l'entropie de ces chaînes, c'est-à-dire d'obtenir des suites de bits indépendants et uniformément distribués.

D'autre part, le caractère non inversible des fonctions de hachage permet de dériver des nombres pseudo-aléatoires à partir de secrets sans les compromettre. Ces fonctions permettent également de mettre à jour les valeurs secrètes, afin que leur compromission éventuelle n'affecte pas leurs valeurs passées. Le schéma défini par Barak et Halevi dans [BH05], fait un tel usage des fonctions de hachage.

On peut facilement construire un générateur pseudo-aléatoire à partir d'une fonction de hachage. Par exemple, si x est le "seed" du générateur, on peut utiliser $F(x|0), F(x|1), F(x|2), ...$ Cette suite pseudo-aléatoire peut être utilisée comme suite générée dans un schéma de chiffrement par flux.

Certains protocoles de communication reposent entièrement sur la cryptographie symétrique. Les équipements qui communiquent possèdent dans ce cas un secret partagé S, défini avant leur déploiement. Afin de limiter l'impact de la compromission des clés utilisées pour protéger des communications, il est nécessaire de renouveler régulièrement leurs valeurs. Une manière de procéder consiste à calculer la $i^{ème}$ clé k_i comme le haché du secret S et d'un compteur i.

1.4.4 Stockage de mots de passe

De nombreux systèmes informatiques authentifient leurs utilisateurs grâce à la connaissance de mots de passe. Un inconvénient potentiel de cette méthode est que si les mots de passe des utilisateurs sont gardés en mémoire sur le système et si un attaquant parvient à accéder à ces mots de passe, il peut s'authentifier sous l'identité de n'importe quel utilisateur. Ce problème peut être résolu à l'aide de fonctions de hachage.

Les fonctions de hachage sont utilisées pour éviter de stocker des mots de passe en clair. Ainsi, quand on se connecte sur un ordinateur, la machine calcule un haché du mot de passe, et le compare au haché préalablement connu. Ceci permet d'éviter de stocker le mot de passe en clair, et si la machine est compromise, l'attaquant ne pourra pas retrouver les mots de passe des utilisateurs (voir la Figure 1.8). Cette méthode semble sûre puisqu'une fonction de hachage est à sens unique. Ainsi, même si un attaquant mettait la main sur les hachés des mots de passe stockées sur la machine , il lui serait pratiquement impossible de retrouver le mot de passe d'accès [Lan11].

Mais cette méthode possède une faille de sécurité importante, à la quelle on peut

remédier. En effet, supposons que deux personnes ont le même mot de passe. Alors, les hachés de leurs mots de passe seront également identiques. Si un attaquant s'en apercevait, il pourrait en conclure que le mot de passe n'est pas une suite aléatoire de symboles, mais plutôt un mot du dictionnaire. Ainsi, il peut trouver le moyen d'accéder aux comptes désirés en essayant tous les mots figurant dans le dictionnaire (attaque par dictionnaire). Pour contrer ce type d'attaque, on ajoute généralement une suite aléatoire, appelée sel, qui peut être par exemple l'heure et la date d'attribution du mot de passe. Cette suite aléatoire est stockée en clair dans la mémoire du système. Le haché est donc celui du mot de passe concaténé avec la composante aléatoire, créant ainsi deux hachés totalement différents (par effet d'avalanche).

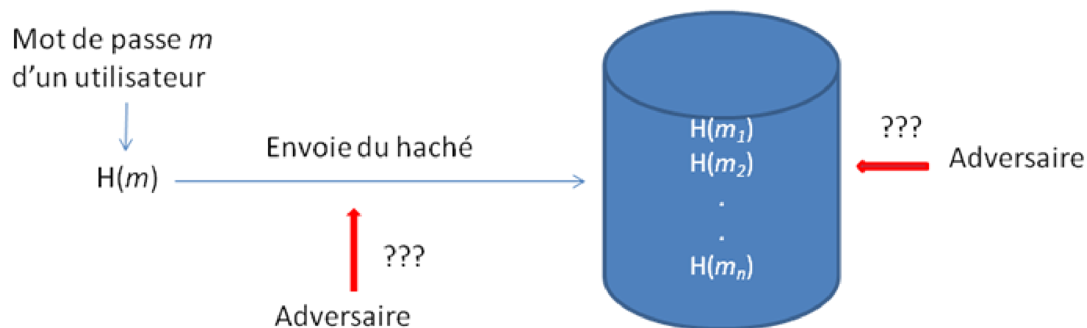

FIGURE 1.8 – Stockage sécurisé de mots de passe.

1.4.5 Schémas d'engagement

Les fonctions de hachage sont aussi utilisées dans certains protocoles pour s'engager à l'avance sur le choix d'une certaine valeur ou même pour confirmer la connaissance d'un certain secret, sans le révéler. Par exemple, le calcul de secret partagé entre deux entités utilise souvent ce genre de techniques [Pey08].

Une fonction de hachage permet de s'engager sur un message : on dévoile d'abord le haché d'un message, puis on révèle plus tard son contenu. Le haché ne révèle pas d'information exploitable sur le message, mais il garantit qu'on ne peut pas modifier le message après avoir révélé le haché. On peut utiliser un schéma d'engagement pour réaliser des enchères secrètes, par exemple. Ce type d'engagement est aussi utile dans de nombreux protocoles cryptographiques, par exemple, pour que plusieurs participants choisissent une valeur aléatoire sans possibilité de triche : chaque participant s'engage sur une valeur aléatoire, puis les valeurs sont révélées et on calcule leurs hachés pour vérifier.

Un schéma d'engagement permet aussi d'horodater un document, pour garantir qu'il existait à une certaine date. Pour construire un horodatage, ou time-stamp, on enregistre le haché d'un message auprès d'une autorité, et cette autorité peut ensuite certifier la date à laquelle le haché, et donc le message, était connu. Un time-stamp peut servir à prouver l'antériorité d'une découverte [Leu10].

Pour ce type d'utilisation, on n'a pas forcément besoin de résistance en collision (si on sait fabriquer une collision, il est vrai que les deux messages sont connus au moment où le haché est publié), mais on a besoin d'une notion plus forte que la résistance en pré-image. En effet, il est possible de s'engager sur une valeur particulière, pour laquelle il sera plus facile de modifier le message ultérieurement.

1.4.6 Protection des fichiers

Une des façons d'utiliser une fonction de hachage est de considérer l'empreinte d'un document comme un identifiant unique. En effet, une bonne fonction de hachage est résistante aux collisions, i.e. on ne peut pas trouver deux messages ayant la même empreinte [Leu10]. On peut utiliser ces identifiants pour vérifier l'intégrité d'un document, ou pour identifier un document si l'empreinte est mieux protégée que le document lui-même. Ceci sert notamment dans certains protocoles de téléchargement pair-à-pair : on obtient le haché du document depuis un serveur central, et le haché sert à vérifier les données reçues depuis les pairs.

1.4.7 Authentification par défi/réponse

Les MAC sont souvent utilisés pour construire des protocoles d'authentification. Dans un protocole d'authentification simple, un client veut s'identifier auprès d'un serveur avec qui il partage un mot de passe. Le serveur envoie un message aléatoire appelé défi ou challenge au client, et le client répond avec un MAC du défi, en utilisant le mot de passe comme clé. Cela ne révèle pas d'information utile sur la clé à un adversaire, mais le serveur peut vérifier que le calcul est correct et donc identifier l'utilisateur [Bou12].

Les protocoles par défi/réponse sont très utilisés en pratique. Par exemple, le mode d'authentification CRAM-MD5 [KCK97], utilisé dans SASL, POP3, IMAP, et SMTP, est un protocole défi/réponse construit avec HMAC-MD5.

1.5 Conclusion

Une fonction de hachage est une composante fondamentale dans un système de sécurité de l'information, et joue un rôle important dans plusieurs applications cryptographiques. Ainsi, elle permet la vérification de l'intégrité des données, l'authentification symétrique de source des messages (MAC), et la validation des signatures numériques (authentification asymétriques). Autrement, sans authentification ou signature, un intrus peut intercepter le message transmis, le modifier et le renvoyer avec la valeur de hachage recalculée pour le message modifié.

Dans ce chapitre, plusieurs approches de construction fonctions de hachage cryptographique ont été présentés, avec l'accent sur leurs avantages et inconvénients. Les notions de sécurité et les attaques sur les fonctions de hachage ont été discutés. Il a été aussi montré comment les fonctions de hachage offrent un moyen efficace pour protéger l'intégrité des données, accélérer les signatures numériques et générer des nombres pseudo-aléatoires de qualité.

Chapitre 2

Les fonctions de hachages MD-SHA

Les fonctions de hachage de la famille MD-SHA (Messages Digest - Standard Hash Algorithm), ont été pendant de longues années les normes en matière de fonctions de hachage cryptographiques, mais la plus part de ces fonctions ont été cryptanalysées.

La première fonction de cette famille est MD2, qui était conçu par R. Rivest en 1989. Elle a été remplacée rapidement par une version améliorée : le MD4. Le MD5, publié en 1991 pour remplacer le MD4, produit une empreinte de 128 bits. Comme nous l'avons vu déjà au premier chapitre, cela n'est pas suffisant actuellement pour assurer une bonne sécurité. En 1996, une faiblesse de sécurité a été découverte dans la fonction de compression de MD5, montrant qu'il y avait une technique permettant de générer des collisions plus rapidement qu'une attaque générique. Ainsi, le MD5 a progressivement été retiré pour être remplacé par le SHA-1. En 2004, des collisions complètes ont été trouvées, ce qui implique que le MD5 n'est plus sûr pour des applications cryptographiques. En 2005 le SHA-1 a été aussi cryptanalysé par Wang et al. [WYY05a] . Donc, les seules fonctions de cette famille à utiliser actuellement dans des systèmes de sécurité sont : SHA-2 et la nouvelle norme SHA-3

Toutes les fonctions de hachage présentées dans ce chapitre utilisent l'algorithme de Merkle-Damgård comme mode opératoire. Donc, ici, nous décrivons uniquement les fonctions de compression, qui calculent une variable de chaînage h' à partir d'un bloc du message M_i de taille fixe, et d'une autre variable de chaînage h. Toutes les fonctions de hachages décrites ici, divisent le bloc M_i, en mots de 32 bits chacun avant de le traiter, ce qui explique la rapidité de ces fonctions pour les implémentations logicielles, puisque toutes les opérations utilisées sont disponibles sur les microprocesseurs actuels (architectures 32 bits).

Soit n la taille de sortie de la fonction de compression C, m le nombre de mots du bloc du message pour chaque itération de la fonction C, et r le nombre de mots de la variable de chaînage (l'état interne). La variable de chaînage h peut donc être divisée en r mots de 32 bits $h^0, \ldots, h^{r-1}$.

Les mots de la variable de chaînage sont traitées par un réseau de Feistel non équilibré à plusieurs branches [Pey08], chaque mot correspondant à une branche. À chaque étape i, un registre (qui contient un mot) est transformé à l'aide d'une fonction g_i dépendante de i. Les registres sont ensuite permutés par le réseau de Feistel. Certains registres subissent aussi un décalage circulaire avant de passer à l'étape suivante.Il existe t fonctions g_i différentes, chacune d'entre elles est appliquée dans le tour i correspondant. Soit t le nombre de tours de la fonction de compression

et e le nombre d'étapes par tour. Le nombre d'étapes pour traiter chaque bloc est égal donc à $t \times e$.

Un mot dépendant du message est requis à l'entrée de chaque étape i, pour générer l'expansion de message : à partir du message M constitué de m mots, construir le message étendu W comportant $s = t \times e$ mots.

À chaque étape, une chaine binaire W de s mots est calculé à partir de M. Ce procédé est appelé "expansion" du bloc M. Il existe deux types d'expansion [Pey08]. Le premier type est celui de la famille MD et RIPEMD. Ce dernier utilise des permutations des mots de M pour chaque tour. Chaque mot du bloc sera employé une seule fois par tour. Dans le cas des fonctions SHA, une formule (qu'on va présenter par la suite) est définie pour calculer les mots de W, dont l'initialisation est donnée par les mots de M.

À chaque tour, un seul mot sera traité. Ce mot est appelé "le registre cible". Nous pouvons ne considérer que les registres de l'état initial et les registres cibles pour la description de la fonction puisque la variable de chaînage à chaque étape peut être reconstruite grâce à ces valeurs. Ainsi, l'état interne durant l'exécution peut être vu comme un vecteur de $s + r$ mots de 32 bits $A_{r+1}, \ldots, A_s$ représentant les registres cibles et l'état initial.

A_{i+1} représente la valeur du registre cible traité pendant l'étape i pour $0 \leq i \leq s$. Lors de l'initialisation, les registres de la variable de chaînage d'entrée seront insérés dans les r premiers registres de l'état interne. Cette description simplifiée des fonctions de compression permet de comprendre mieux leurs fonctionnement.

Toutes les fonctions de la famille MD-SHA utilisent la construction de Davies-Meyer (décrite dans la section 1.3.1 du chapitre précédant). Dans ce type de constructions la variable de chaînage d'entrée est additionnée avec les registres internes finaux pour former la variable de chaînage de sortie. Ce procédé est utilisé pour éviter l'inversion de la fonction de compression (ce qui génère une attaque par pré-image sur la fonction de hachage complète) [Dau05]. Dans ce cas la, la fonction de compression est similaire à un algorithme de chiffrement par blocs E (avec une structure de réseau de Feistel non équilibré) placé dans une construction de Davies-Meyer. Cette fonction E crypte la variable de chaînage h en utilsant le bloc du message M comme clé, la phase d'expansion de message représente la le processus de préparation des clés de E. Ce formalisme est utilisé pour créer un algorithme de chiffrement par blocs indépendant nommé "SHACAL" [HN02], utilisant l'algorithme interne de la fonction de compression de SHA-1. Comme expliquer dans la section 1.3.2 du chapitre 1, le schéma de Davies-Meyer a été étudié par Preneel et al. [PGV93], puis prouvé sûr par Black et al. [BRS02], sous l'hypothèse que la fonction de chiffrement par bloc E

soit une fonction parfaite.

Le principal avantage des fonctions de hachage MD-SHA est leur rapidité dans une implémentation logiciel. En effet, seules des opérations très simples et bien supportées par les microprocesseurs 32 bits sont utilisées pour le hachage (additions modulaires, fonctions booléennes, rotations et décalages) [Bou12]. Dans ce qui suit nous allons présenter les principales fonctions de cette famille.

2.1 Les fonctions "Messages Digest"

2.1.1 MD2

C'est la première fonction de hachage de la famille MD a être implémenté. Elle a été conçue par Rivest en 1989, ensuite standardisée par l'IETF [1] en 1992 [Kal92]. Rogier et al. ont montré en 1997 comment générer des collisions contre la fonction de compression [RC97]. Plus tard, des attaques en recherche de collisions [KM05] et de pré-images [Mul04] sur la fonction de hachage complète ont été découvertes. Cette fonction n'est plus utilisée actuellement.

2.1.2 MD4

Cette fonction peut être considérée comme l'origine de base de la famille MD-SHA. En effet, les principes de conception de sa fonction de compression sont largement repris par les fonctions ulterieurs. Cependant, cette fonction n'a été que peu utilisée, car elle a été remplacée par MD5. Elle a été proposée par Rivest en 1990 pour les laboratoires RSA et normalisée par l'IETF dans [Riv92a].

MD4 utilise des variables de chaînage de 128 bits et des blocs de message de 512 bits, qui sont divisés en variables de 32 bits. La fonction de compression est construite selon le mode de Davies-Meyer, avec une addition modulo 2^{32} des 4 mots de 32 bits qui composent la variable de chaînage. La fonction de compression consiste à modifier alternativement chacun des 4 registres de la variable de chaînage, par une opération faisant intervenir l'un des 16 registres du message. 48 étapes sont ainsi effectuées [Leu10].

La fonction de compression produit des hachés de 128 bits pour un état interne de $r = 4$ registres de 32 bits chacun, initialisé par la variable de chaînage d'entrée :

$A_{-3} = h_0, \quad A_{-2} = h_3, \quad A_{-1} = h_2, \quad A_0 = h_1$

À chaque execution de la fonction de compression, 16 mots de message seront traités, avec 3 tours de 16 étapes chacun (c'est à dire 48 étapes en tout). L'expansion

1. The Internet Engineering Task Force (https ://www.ietf.org/)

de message est très simple : pour chaque tour j une permutation π_j de l'ordre des mots de message est définie (π_0 est difinie comme l'application identité). Ainsi, pour la $k^{ième}$ étape du tour j , avec $0 \leq j \leq 2$ et $0 \leq k \leq 15$, nous avons :

$$W_{j \times 16+k} = M_{\pi j(k)}$$

Puisque l'expansion de message est une permutation par tour des mots du message d'entrée M , chaque mot de M sera utilisé une fois pour chaque tour. Durant chaque étape i, le registre A_{i+1} est mis à jour par la fonction f_j , dépendante du tour j auquel appartient i :

$$\begin{aligned} A_{i+1} &= f_j(A_i, A_{i-1}, A_{i-2}, A_{i-3}, W_i, s_i) \\ &= (A_{i-3} + F_j(A_i, A_{i-1}, A_{i-2}) + W_i + K_j)^{<<<s_i}, \end{aligned}$$

Avec K_j des constantes prédéfinies pour chaque tour, les s_i sont des valeurs de rotation prédéfinies pour chaque étape, et les fonctions F_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée. A la fin des 48 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$$h'_0 = A_{45} + A{-}3, \quad h'_1 = A_{48} + A_0, \quad h'_2 = A_{47} + A_{-1}, \quad h'_3 = A_{46} + A_{-2}$$

Une description visuelle d'une étape de la fonction de compression est donnée dans la figure 2.1. Les caractéristiques complètes de la fonction peuvent être trouvées dans [Riv92a].

Sécurité de la fonction MD4

Des failles de sécurité pour la fonction de compression ont été rapidement trouvées, juste après la publication de MD4. En 1991, Den Boer et al. [dBB91] ont attaqué une version réduite à 2 tours au lieu des 3 tours de la fonction MD4 complète (c'est l'une des raisons pour lesquelles la fonction de compression de MD5 est composée de 4 tours au lieu de trois). Cinq ans plus tard, Dobbertin et al. [Dob96] ont publié en 1996 une attaque pratique permettant de trouver des collisions pour la fonction complète dont la complexité est équivalente à 2^{20} appels à la fonction de hachage. Après plusieurs avancées, Wang et al. [WLF+05] ont publié une série d'attaques contre des fonctions de la famille MD-SHA, dont MD4, pour laquelle la recherche de collisions nécessite moins de 2^8 appels à la fonction.

En 2007, les avancées de la cryptanalyse ont permis de trouver des collisions pour MD4 avec une complexité à peu près égale à celle pour vérifier ces collisions

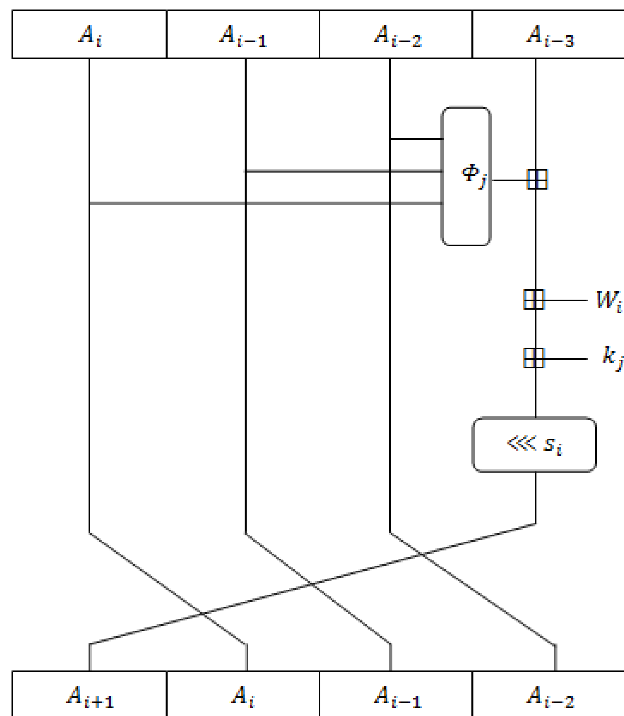

FIGURE 2.1 – Une étape de la fonction de compression de MD4.

[SWOK07]. La résistance aux attaques par pré-images de MD4 a ensuite été cassée par Leurent en 2008 [Leu08].

Plusieurs protocoles cryptographiques basés sur MD4 (tels que HMAC [FLN07, WOK08] et APOP [Leu07]) ont été aussi attaqués. Ce qui implique que toute utilisation de MD4 comme primitive cryptographique est aujourd'hui, déconseillé.

2.1.3 MD5

La fonction MD5 a été conçue par Rivest en 1991 [Riv92b]. Cette fonction a été très utilisée, et elle est toujours disponible aujourd'hui dans plusieurs systèmes et protocoles. Elle représente une amélioration de MD4 : un tour est ajouté à la fonction de compression (ce qui représente 16 étapes), et les étapes élémentaires sont aussi modifiées.

Donc la fonction de compression de MD5 possède un tour de plus, par rapport à MD4, de nouvelles fonctions booléennes et des constantes définies pour chaque étape. Les principes généraux restent fixes : des hachés de taille $n = 128$ bits pour un état interne de $r = 4$ registres de $w = 32$ bits chacun, initialisé avec la variable de chaînage d'entrée :

$$A_{-3} = h_0, \quad A_{-2} = h_3, \quad A_{-1} = h_2, \quad A_0 = h_1$$

À chaque exécution de la fonction de compression, 16 mots de message sont traités, durant 4 tours de 16 étapes chacun (c'est à dire 64 étapes en tout). L'expansion de message reste très simple : pour chaque tour j , une permutation π_j de l'ordre des mots de message est définie (π_0 représente l'application identité). Ainsi, nous avons pour la $k^{ième}$ étape du tour j , avec $0 \leq j \leq 3$ et $0 \leq k \leq 15$:

$$W_{j \times 16+k} = M_{\pi_j(k)}$$

Comme pour MD4, on peut observer que puisque l'expansion de message est une permutation par tour des mots du message d'entrée M, chaque mot de M sera utilisé une fois pour chaque tour. L'expansion de message de MD5 est donc similaire, sauf que les permutations ont été légèrement modifiées.

Pour chaque étape i le registre cible A_{i+1} est mis à jour par la fonction f_j , qui dépend du tour j auquel i appartient :

$$\begin{aligned} A_{i+1} &= f_j(A_i, A_{i-1}, A_{i-2}, A_{i-3}, W_i, K_i, s_i) \\ &= A_i + (A_{i-3} + F_j(A_i, A_{i-1}, A_{i-2}) + W_i + K_i)^{<<<s_i} \end{aligned}$$

,

Où K_i sont des constantes prédéfinies pour chaque étape, les s_i sont des valeurs de rotation prédéfinies pour chaque étape, et les fonctions F_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée. À la fin des 64 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$$h'0 = A_{61} + A_{-3}, \quad h'_1 = A_{64} + A_0, \quad h'_2 = A_{63} + A_{-1}, \quad h'_3 = A_{62} + A_{-2}$$

Une représentation d'une étape est donnée dans la figure 2.2 et les caractéristiques complètes de la fonction peuvent être trouvées dans [Riv92b].

Sécurité de la fonction MD5

La fonction de hachage MD5 a résisté un peu plus longtemps que MD4 aux attaques de cryptanalyse, pour cette raison, elle a réussi à mieux intégré le monde industriel et a été implantée dans de très nombreux systèmes. Une pseudo-collision a été tout d'abord découverte en 1993 [dBB93] par Den Boer et al. pour la fonction de compression de MD5, puis en 1996, une attaque trouvant des collisions contre la fonction de hachage complète (mais avec une valeur d'initialisation différente de celle prédéfinie) a été présentée par H Dobbertin [Dob96]. C'est grâce au travaux de Wang et al. [WY05] en 2005 que la première collision pour MD5 a pu être calculée, avec

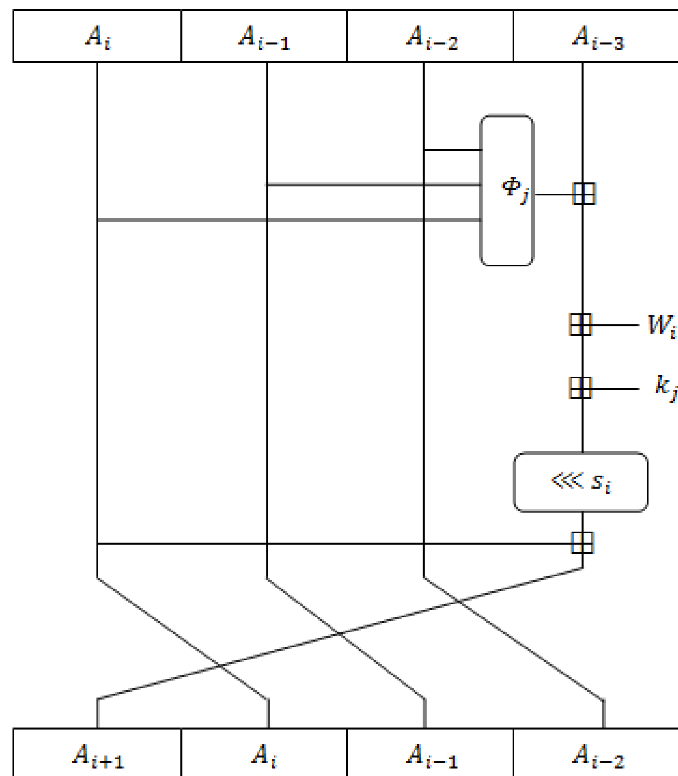

FIGURE 2.2 – Une étape de la fonction de compression de MD5.

une complexité de 2^{39} appels à la fonction de hachage. Beaucoup d'amélioration ont été faite depuis pour cryptanalyser cette fonction. Par exemple, la technique optimisée de Klima [Kli06] permet de trouver des collisions pour MD5 en quelques secondes sur un ordinateur personnel standard.

Comme pour MD4, beaucoup de chercheurs continuent d'analyser MD5, car MD5 est encore utilisée dans certaines applications, ce qui pose la question des conséquences de ces attaques. Par exemple, de nombreux travaux [Leu07, SYA07, FLN07] analysent la possibilité de casser la sécurité de HMAC ou d'APOP lorsque MD5 est utilisé comme fonction interne. De nouvelles failles de sécurité ont récemment été identifiées concernant l'utilisation de MD5 pour vérifier l'intégrité de programmes exécutables [SLdW07b] ou pour la signature de certificats X.509 [SLdW07a].

Même si la résistance de la fonction MD5 contre la recherche de pré-images n'est pour l'instant pas mise en question, MD5 ne doit plus être implantée dans des applications cryptographiques.

2.1.4 HAVAL

La fonction de hachage HAVAL [ZPS92], très similaire à MD4, a été inventée par Zheng et al. en 1992. L'une des différences est que le nombre de tours et la taille des hachés peuvent être modifiés, donnant ainsi naissance à une famille de fonctions HAVAL pouvant être personnalisées selon les besoins. Cette particularité sera reprise par la suite pour les nouvelles fonctions de hachage conçues. Par défaut, les hachés sont de taille $n = 256$ (les tailles 128, 160, 192 et 224 bits en sortie sont aussi possibles grâce à une méthode de troncature de la sortie de 256 bits) pour un état interne de 8 registres de 32 bits chacun, initialisé par la variable de chaînage d'entrée.

À chaque exécution de la fonction de compression, 32 mots de message seront traités, avec t tours de 32 étapes chacun (c'est-à-dire $t \times 32$ étapes en tout). Suivant la version utilisée, le nombre de tours t peut être égal à 3, 4 ou 5. Cette deuxième possibilité de paramétrage semble être un élément nécessaire pour les futures fonctions de hachage, un paramètre de sécurité permettant facilement d'augmenter la robustesse du schéma (au détriment de la rapidité) sans devoir réimplanter complètement la fonction. Cela permet de mieux anticiper et gérer les avancées de la cryptanalyse. Une telle fonctionnalité a été requise pour tout candidat à l'appel de soumissions du NIST pour le standard SHA-3. Les caractéristiques complètes de la fonction HAVAL sont décrites dans [ZPS92].

Sécurité de la fonction HAVAL

Comme pour MD4 ou MD5, les chercheurs ont analysé tout d'abord des versions réduites d'HAVAL [HSK03], et ont tenté de trouver des comportements non aléatoires dans la fonction [YBC+04]. La première attaque contre une version complète d'HAVAL (version 3 tours et n'importe quelle taille de sortie) a été réalisé par Van Rompay et al [RBPV03] et permet de trouver des collisions avec une complexité de 2^{29} appels à la fonction de hachage.

en 2004, Wang et al [WFLY04] ont réduit la complexité à seulement 2^6 appels pour la version 3 tours avec 128 bits du haché. Deux ans plus tard Wang et al. ont cryptanalysé les versions 4 et 5 tours de HAVAL [YWYP06] (avec 2^{36} et 2^{123} appels à la fonction de hachage respectivement). Pour ces raisons, HAVAL ne doit plus être implémenté dans des applications cryptographiques.

2.2 Les fonctions "Standard Hash Algorithm"

La famille des fonctions de hachage la plus connue est la SHA (Secure Hash Standard) qui est développée par la NSA et certifiée par le NIST. Les fonctions SHA-256 et SHA-512 sont à ce jour, les plus sûres et les plus utilisées. Le dernier membre de cette famille est la fonction Keccak [BDPA09] qui a remporté la compétition de NIST en 2012, et est ainsi devenue la nouveau standard SHA-3 [Bou12].

2.2.1 SHA-0

SHA-0 est la première fonction de la famille " Secure Hash Standard ". Elle a été développée par la NSA [2] en 1993 et normalisée par le NIST [3] [oST93], avant d'être retirée peu après et remplacée par SHA-1. Inspirée des fonctions de la famille MD, la fonction de compression de SHA-0 n'en diffère de celle de MD5, que par l'utilisation d'une nouvelle expansion de message : au lieu d'utiliser des permutations des mots de message pour chaque tour, les mots de message étendu sont obtenus par un procédé récursif initialisé par les mots du message d'entrée [Fuh11]. SHA-0 permet de calculer des empreintes de 160 bits, ce qui permet de résister mieux aux attaques génériques. En 1995, le NIST a publié la fonction SHA-1, que nous détaillons par la suite, et qui représente une modification mineure par rapport à SHA-0.

SHA-0 produit des hachés de $n = 160$ bits pour un état interne de $r = 5$ registres de $w = 32$ bits chacun, initialisé par la variable de chaînage d'entrée :

$$A_{-4} = (h_4)^{<<<2}, A_{-3} = (h_3)^{<<<2}, A_{-2} = (h_2)^{<<<2}, A_{-1} = h_1, A_0 = h_0$$

Durant l'exécution de la fonction de compression, 16 mots de message seront traités, avec 4 tours de 20 étapes chacun (c'est à dire 80 étapes en tout). L'expansion de message n'utilise donc pas de permutation. Les 16 premiers mots du message sont utilisés pour initialiser les 16 mots du message d'entrée de la fonction de compression. Le reste des W_i sont calculés par une formule de récurrence :

$$W_i = \begin{cases} M_i, & pour 0 \leq i \leq 15 \\ W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, & pour 16 \leq i \leq 79 \end{cases}$$

Durant chaque étape i, la fonction f_j, dépendante du tour j auquel i appartient, met à jour le registre cible A_{i+1} :

2. National Security Agency (www.nsa.gov)
3. National Institute of Standards and Technology (www.nist.gov)

$$A_{i+1} = f_j(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, W_i)$$
$$= (A_i)^{<<<5} + F_j(A_{i-1}, (A_{i-2})^{>>>2}, (A_{i-3})^{>>>2}) + (A_{i-4})^{>>>2} + W_i + K_j,$$

Où K_j représentent des constantes prédéfinies pour chaque tour et les fonctions F_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée. À la fin des 80 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$h'_0 = A_{80} + A_0$, $h'_1 = A_{79} + A_{-1}$, $h'_2 = (A_{78})^{>>>2} + (A_{-2})^{>>>2}$, $h'_3 = (A_{77})^{>>>2} + (A_{-3})^{>>>2}$, $h'_4 = (A_{76})^{>>>2} + (A_{-4})^{>>>2}$

Une description visuelle d'une étape est donnée dans la figure 2.3 et les caractéristiques complètes de la fonction peuvent être trouvées dans [oST93].

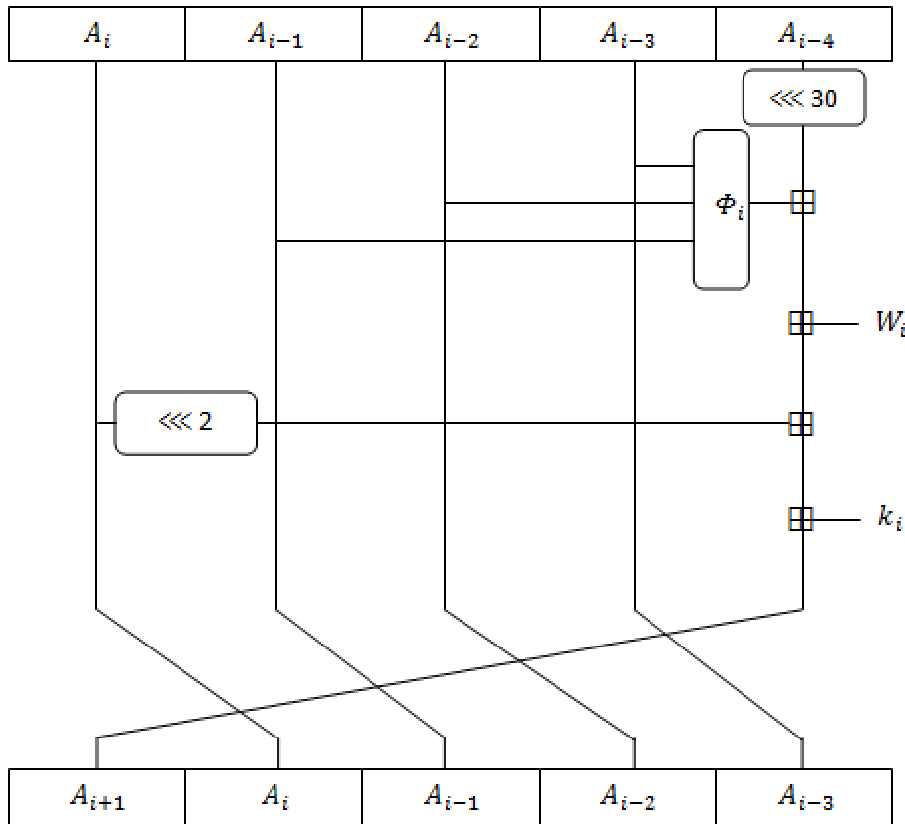

FIGURE 2.3 – Une étape de la fonction de compression de SHA-0.

Sécurité de la fonction SHA-0

Les premières vulnérabilités de SHA-0 ont été découvertes par Chabaud et al. dans [CJ98], qui sont parvenus à démontrer une attaque théorique contre SHA-0 d'une complexité de 2^{61} appels à la fonction de hachage.

En comparaison avec la famille MD, le principe d'une formule de récurrence pour calculer les mots du message étendu a permet d'obtenir de meilleures propriétés en termes de diffusion. Par exemple, une perturbation sur un mot du message d'entrée M (modifiant directement l'un des premiers mots de W) influera très rapidement l'intégralité des mots suivants de W , ce qui n'est pas vrai pour une permutation des mots de M par tour comme dans le cas de la famille MD. Cependant, dans le cas de SHA-0, cette diffusion ne s'opère que sur une seule position de bit [Bou12]. Ainsi, si l'on se limite à ne modifier M que sur une position j, cela n'aura des conséquences que sur les bits en position j des mots de W. Cette faiblesse est directement corrigée dans SHA-1, où une rotation est ajoutée dans l'expansion de message.

Après cette première attaque théorique pour la recherche de collisions, les attaques de Wang et al. [WYY05a] s'appliquant également à SHA-0 ont permis d'accélérer la recherche de collisions. Ensuite, en 2005, en utilisant plusieurs blocs de message contenant des différences, Biham et al. [BCJ+05] ont publié la première collision pour SHA-0. La complexité étant de l'ordre de 2^{51} appels à la fonction de compression, un très grand nombre d'ordinateurs étaient nécessaires pour trouver la collision en seulement trois semaines. Cette complexité a été améliorée tout d'abord par les travaux de Wang et al. [WYY05c], calculant des collisions pour une complexité de 2^{39} appels à la fonction de compression, puis par Naito et al. [NSS+06]. Une attaque de Peyrin et al. permet de trouver des collisions pour SHA-0 en pratique, avec une puissance de calcul raisonnable [MP08]. Du fait de la proximité entre la publication de SHA-0 et celle de SHA-1, SHA-0 n'a que rarement été utilisée dans les applications pratiques.

2.2.2 SHA-1

SHA-1 a été publiée en 1995 par le NIST pour remplacer la fonction SHA-0 [oST95]. Malgré ses vulnérabilités connues qui conduisent à son remplacement progressif par SHA-2, elle reste probablement aujourd'hui la fonction la plus utilisée dans la pratique. Nous décrivons maintenant les détails de son fonctionnement.

SHA-1 est construite avec l'algorithme d'extension de domaine de Merkle-Damgård. Sa fonction de compression permet le traitement de blocs de message de 512 bits et de variables de chaînage de 160 bits. Elle est construite à partir d'une permutation paramétrée en mode de Davies-Meyer, où l'opération XOR est remplacé par 5 additions modulo 2^{32} entre les 5 registres de 32 bits d'entrée de la variable de chaînage et les 5 registres de sortie de la permutation. Le bloc de message, divisé en 16 mots de 32 bits $(M_0, ..., M_{15})$ passe par une expansion linéaire (dans $GF(2)$) permettant d'obtenir 80 mots de 32 bits, de la manière suivante [Fuh11] :

$$\forall i \in 0, ..., 15, W_i = M_i$$

$$\forall i \in 16, ..., 79, W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) << 1$$

Cette rotation de 1 bit vers la gauche dans l'expansion de message constitue une des différences entre SHA-0 et SHA-1. A partir de la variable de chaînage d'entrée, on initialise 5 registres A, B, C, D et E.

$$A_0 = H_0^{i-1}, \quad B_0 = H_1^{i-1}, \quad C_0 = H_2^{i-1}, \quad D_0 = H_3^{i-1}, \quad E_0 = H_4^{i-1}$$

La permutation paramétrée de SHA-1 se décompose en 4 tours de 20 étapes. Chaque étape est définie par les opérations suivantes :

$$A_{i+1} = (A_i << 5) \oplus \phi_i(B_i, C_i, D_i) \oplus E_i \oplus W_i \oplus k_i$$

$$B_{i+1} = A_i$$

$$C_{i+1} = B_i^{<<<30}$$

$$D_{i+1} = C_i$$

$$E_{i+1} = D_i$$

,

Où k_i est une constante dépendante du numéro de l'étape et ϕ_i est la juxtaposition de 32 fonctions identiques de 3 bits vers 1 bit. Une étape de la fonction de compression de SHA-1 est décrite dans la Figure 2.4. La nouvelle valeur de la variable de chaînage est définie par :

$$H_i^0 = A_0 \oplus A_{80}$$

$$H_i^1 = B_0 \oplus B_{80}$$

$$H_i^2 = C_0 \oplus C_{80}$$

$$H_i^3 = D_0 \oplus D_{80}$$

$$H_i^4 = E_0 \oplus E_{80}$$

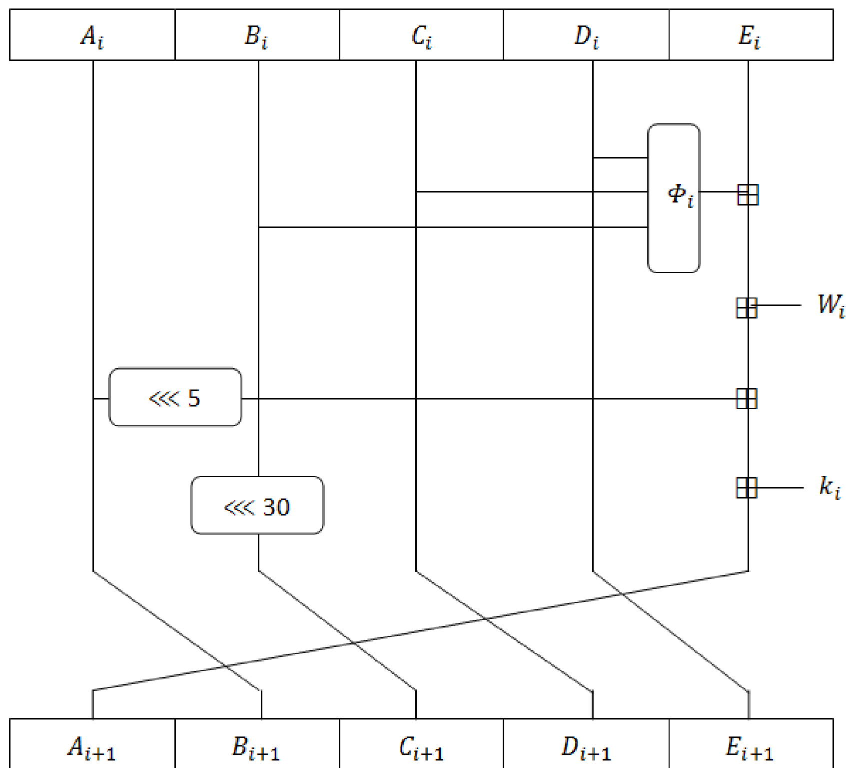

Figure 2.4 – Une étape de la fonction de compression de SHA-1.

Sécurité de la fonction SHA-1

SHA-1 a résisté quelques années relativement bien aux attaques connues contre sa version antérieure SHA-0, seules des vulnérabilités sur des versions réduites ont été publiées jusqu'à 2005 [BCJ+05]. Ceci est dû au fait que la diffusion des mots de message est bien meilleure grâce à la rotation ajoutée. Mais, en 2005 Wang et al. [WYY05a] ont présenté la première attaque théorique contre SHA-1, en 2^{69} appels à la fonction de compression (améliorée plus tard pour une complexité de 2^{63} appels à la fonction de compression [WYY05b]). Plusieurs travaux ont été ensuite publiés pour essayer d'implémenter ces nouvelles attaques [CR06, JP07]. Même si aucune collision sur la version complète n'a encore été trouvée et même si la résistance en pré-images n'est pour l'instant toujours pas mise en question, il est très déconseillé d'utiliser SHA-1 dans une application cryptographique. Ceci explique en partie la décision du NIST d'organiser un appel à soumissions en 2008, pour de nouvelles fonctions de hachage, dans le but de standardiser SHA-3 [oST08].

2.2.3 SHA-256

Publiée en 2002 [oST02], SHA-256 fait partie des derniers membres de la famille MD-SHA. Outre sa taille du haché, elle contient plusieurs nouveautés par rapport à ses prédécesseurs. Par exemple, l'expansion de message est beaucoup plus complexe et corrige les précédentes erreurs de SHA-0 et SHA-1. De plus, la fonction d'étape met à jour deux registres à la fois pour une meilleure diffusion. Nous notons dans la suite A_i et B_i ces registres cibles. Aussi, il n'y a plus réellement de notion de tour dans SHA-256, car les mêmes fonctions booléennes sont utilisées dans toutes les étapes. Comme son nom l'indique, SHA-256 produit des hachés de $n = 256$ bits, mais il existe aussi une version 224 bits [oST02] introduite 2 ans plus tard. On maintient donc un état interne de 8 registres de 32 bits chacun, initialisé par la variable de chaînage d'entrée :

$$A_{-3} = h_3, \quad A_2 = h_2, \quad A_{-1} = h_1, \quad A_0 = h_0$$

$$B_{-3} = h_7, \quad B_{-2} = h_6, \quad B_{-1} = h_5, \quad B_0 = h_4$$

À chaque exécution de la fonction de compression, 16 mots de message sont traités, avec 64 étapes. L'expansion de message est beaucoup plus complexe que dans les autres versions de SHA, mais utilise toujours une formule de récurrence :

$$W_i = \{(M_i, pour 0 \leq i \leq 15, \sigma^1(W_{i-2}) + W_{i-7} + \sigma^0(W_{i-15}) + W_{i-16}, pour 16 \leq i \leq 63)?$$

Avec :

$$\sigma^0(x) = (x^{>>>7}) \oplus (x^{>>>18}) \oplus (x^{>>>3})$$

$$\sigma^1(x) = (x^{>>>17}) \oplus (x^{>>>19}) \oplus (x^{>>>10})$$

Durant chaque étape i, les registres cibles A_{i+1} et B_{i+1} sont mis à jour par les fonctions f et g respectivement :

$$A_{i+1} = f(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i)$$

$$B_{i+1} = g(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i)$$

Où les K_i sont des constantes prédéfinies pour chaque étape et les fonctions MAJ et IF sont des fonctions booléennes. Les fonctions Σ_0 et Σ_1 sont définies par :

$$\Sigma_0(x) = (x^{>>>2}) \oplus (x^{>>>13}) \oplus (x^{>>>22})$$

$$\Sigma_1(x) = (x^{>>>6}) \oplus (x^{>>>11}) \oplus (x^{>>>25})$$

À la fin des 64 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$h'_0 = A_{64} + A_0, \quad h'_1 = A_{63} + A_{-1}, \quad h'_2 = A_{62} + A_{-2}, \quad h'_3 = A_{61} + A{-3}$

$h'_4 = B_{64} + B_0, \quad h'_5 = B_{63} + B_{-1}, \quad h_6 = B_{62} + B_{-2}, \quad h'_7 = B_{61} + B{-3}$

La version 224 bits ne diffère de la version 256 bits que par ses valeurs d'initialisation différentes et par sa troncature à la fin de la fonction de compression, afin d'obtenir la bonne taille de sortie (le dernier bloc $h'7$ est retiré). Une représentation visuelle d'une étape est donnée dans la Figure 2.5 et une description complète de la fonction 256 bits peut être trouvée dans [oST02].

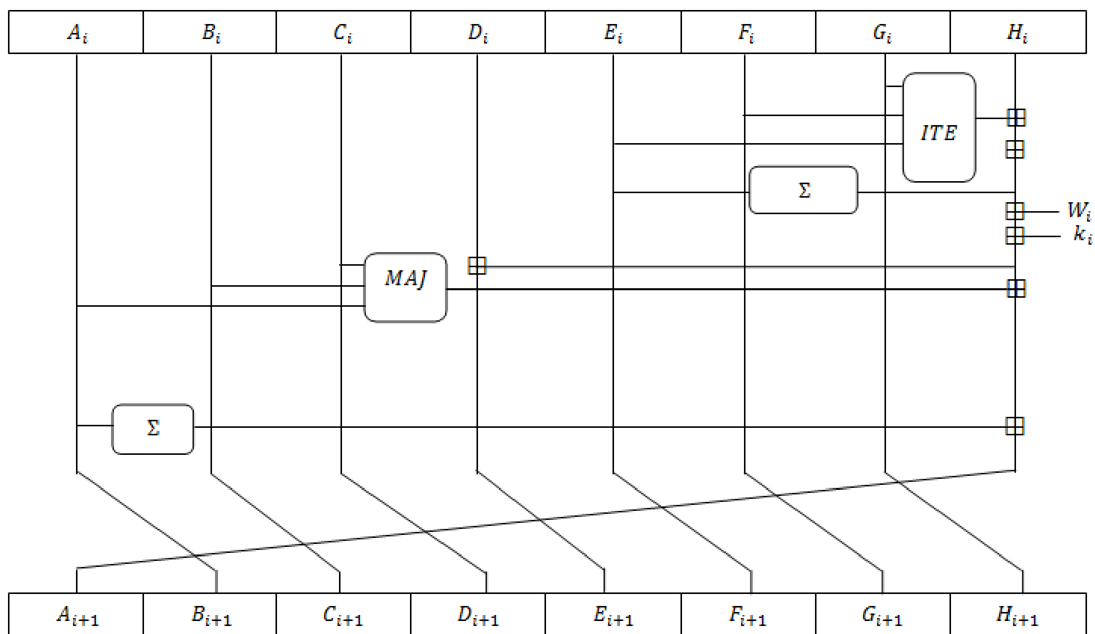

FIGURE 2.5 – Une étape de la fonction de compression de SHA-2.

Sécurité de la fonction SHA-256

Pour l'instant, aucune attaque n'a été publiée pour la version complète de SHA-256. Les essais des chercheurs ne s'appliquent que sur des versions très réduites [IMPR08]. Ceci peut s'expliquer par les améliorations de sécurité apportée par SHA-256 par rapport à ses prédécesseurs de la famille SHA. La mise à jour de deux registres par étape permet une meilleure diffusion, mais surtout l'expansion de message n'est plus linéaire dans F_2 comme cela était le cas dans SHA-0 ou SHA-1. En effet, l'utilisation d'additions modulaires rend le processus non linéaire dans F_2 et complique grandement le contrôle du message étendu pour un attaquant. Mais cette évolution parait logique puisque l'attention a été portée sur l'intégration des mots

de message qui a été largement sous-estimée dans le passé, comparée au traitement robuste de la variable de chaînage [Bou12].

Bien qu'aucune attaque n'ait encore été trouvée, on peut se poser la question de la sécurité inhérente aux membres de la famille MD-SHA du fait de leur passé.

2.2.4 SHA-512

SHA-512 a été publiée en même temps que SHA-256 [oST02] et représente son équivalent pour les processeurs 64 bits, qui vont progressivement remplacer ceux de 32 bits dans les ordinateurs. Les mots traités seront donc de taille 64 bits pour profiter pleinement de cette nouvelle architecture. Les autres différences par rapport à SHA-256 concernent la taille de sortie, qui est doublée pour obtenir une fonction fiable sur le long terme, et le nombre d'étapes qui est augmenté. Ainsi, SHA-512 produit des hachés de $n = 512$ bits, mais une version 384 bits [oST02] a été aussi introduite en même temps. Comme pour SHA-256, deux registres sont mis à jour durant une étape de SHA-512, et nous notons A_i et B_i ces registres cibles. On maintient donc un état interne de $r = 8$ registres de $w = 64$ bits chacun, initialisé par la variable de chaînage d'entrée :

$$A_{-3} = h_3, \quad A_2 = h_2, \quad A_{-1} = h_1, \quad A_0 = h_0$$

$$B_{-3} = h_7, \quad B_{-2} = h_6, \quad B_{-1} = h_5, \quad B_0 = h_4$$

À l'exécution de la fonction de compression, 16 mots de message sont traités, avec 80 étapes. L'expansion de message, toujours de type récursif, est similaire à celle de SHA-256 excepté les fonctions σ_0 et σ_1 :

$$W_i = \begin{cases} M_i, & pour 0 \leq i \leq 15 \\ W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, & pour 16 \leq i \leq 79 \end{cases}$$

avec :

$$W_i = \begin{cases} \sigma_0(x) = (x^{>>>1}) \oplus (x^{>>>8}) \oplus (x^{>>>7}) \\ \sigma_1(x) = (x^{>>>19}) \oplus (x^{>>>61}) \oplus (x^{>>>6}) \end{cases}$$

De la même façon que dans SHA-256, durant chaque étape j , les registres cibles A_{j+1} et B_{j+1} sont mis à jour par les fonctions f et g respectivement :

$$\begin{aligned} A_{i+1} &= f(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i) \\ &= \Sigma_0(A_i) + MAJ(A_i, A_{i-1}, A_{i-2}) + B_{i-3} + \Sigma_1(B_i) + IF(B_i, B_{i-1}, B_{i-2}) + W_i + K_i, \end{aligned}$$

$$\begin{aligned} B_{i+1} &= g(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i) \\ &= A_{i-3} + B_{i-3} + \Sigma_1(B_i) + IF(B_i, B_{i-1}, B_{i-2}) + W_i + K_i, \end{aligned}$$

Où les K_i sont des constantes prédéfinies pour chaque étape et les fonctions MAJ et IF sont des fonctions booléennes. Les fonctions Σ_0 et Σ_1 sont différentes de celles qui sont utilisées dans SHA-256 et sont définies par :

$$\begin{aligned} \Sigma_0(x) &= (x^{>>>28}) \oplus (x^{>>>34}) \oplus (x^{>>39}) \\ \Sigma_1(x) &= (x^{>>>14}) \oplus (x^{>>>18}) \oplus (x^{>>41}) \end{aligned}$$

Enfin, du fait du rebouclage, à la fin des 80 étapes, les mots de la sortie de la fonction de compression sont calculés par :
$h'_0 = A_{80} + A_0, \quad h'_1 = A_{79} + A_{-1}, \quad h'_2 = A_{78} + A_{-2}, \quad h'_3 = A_{77} + A{-3}$
$h'_4 = B_{76} + B_0, \quad h'_5 = B_{75} + B_{-1}, \quad h_6 = B_{74} + B_{-2}, \quad h'_7 = B_{73} + B{-3}$
La version 384 bits ne diffère de la version 512 bits que par ses valeurs d'initialisation différentes et par sa troncature à la fin de la fonction de compression, pour ainsi obtenir la bonne taille de sortie (les deux derniers blocs h'_6 et h'_7sont retirés).

Une représentation visuelle d'une étape est donnée dans la figure 2.6 et les caractéristiques complètes des deux versions de la fonction sont décrites dans [oST02].

Sécurité de la fonction SHA-512

Les remarques sur la sécurité de SHA-256 sont tout aussi valables pour SHA-512, même si l'augmentation du nombre d'étapes et la plus grande taille de sortie semblent la rendre encore plus résistante à des attaques pratiques. Ainsi, SHA-512 peut pour l'instant toujours être utilisée dans des applications cryptographiques.

2.3 Les fonctions RIPEMD

2.3.1 RIPEMD-0

RIPEMD-0 est l'une des primitives recommandées en 1992 à l'issue d'une étude d'un consortium dans le cadre du projet européen RACE Integrity Primitives Eva-

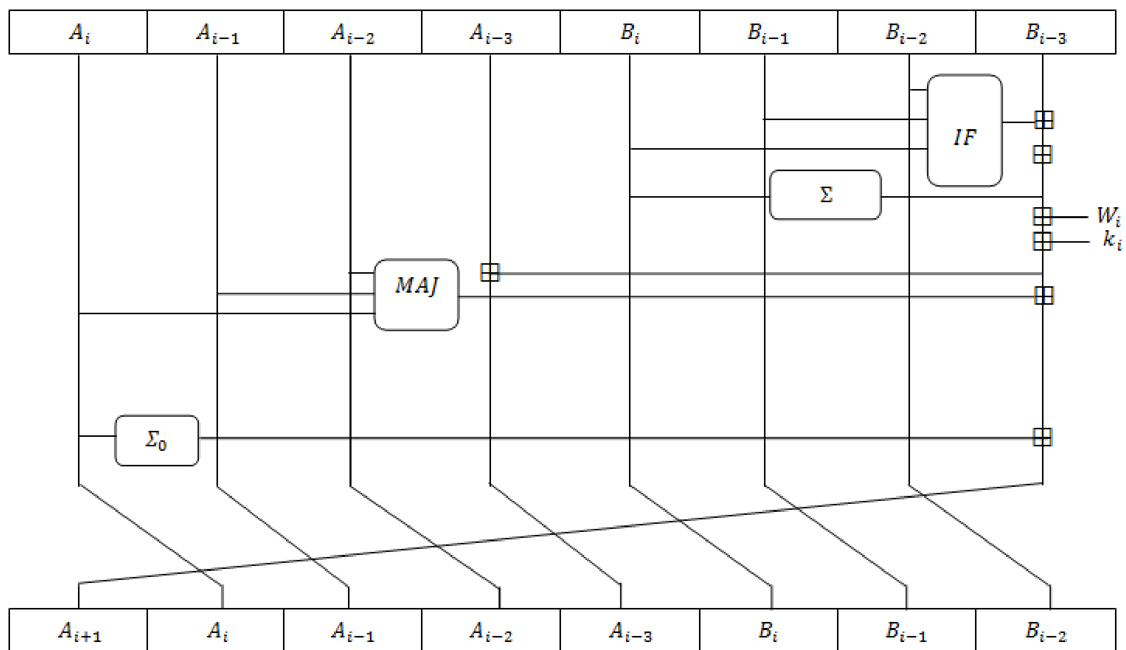

FIGURE 2.6 – Une étape de la fonction de compression de SHA-512.

luation (RIPE) sur les primitives permettant de garantir l'intégrité [RIP95]. Originalement nommée RIPEMD, la fonction de compression se compose de deux branches parallèles, chacune quasiment identique à la fonction de compression de MD4. Les deux lignes parallèles de calcul ne diffèrent que par l'emploi de constantes différentes. Les paramètres de chaque branche sont donc égaux à ceux de MD4, mais l'ordre d'introduction des mots du bloc de message étendu et les longueurs de rotation lors des étapes sont différents de ceux de MD4. Les hachés sont de taille $n = 128$ bits pour un état interne de 4 registres de 32 bits chacun pour chaque branche. On note A_i^L les registres cibles de la branche de gauche et A_i^R ceux de la branche de droite. On initialise chaque branche par la variable de chaînage d'entrée :

$$A_{-3}^L = h_0, \quad A_{-2}^L = h_1, \quad A_{-1}^L = h_2, \quad A_0^L = h_3,$$

$$A_{-3}^R = h_0, \quad A_{-2}^R = h_1, \quad A_{-1}^R = h_2, \quad A_0^R = h_3$$

À l'exécution de la fonction de compression, 16 mots de message sont traités, durant 3 tours de 16 étapes chacun dans chaque branche (c'est à dire 48 étapes en tout). L'expansion de message est légèrement modifiée : pour chaque tour j , une permutation π_j de l'ordre des mots de message est définie (p_0 est l'application identité). Ainsi, nous avons pour la $k^{ième}$ étape du tour j, avec $0 \leq j \leq 2$ et $0 \leq k \leq 15$:

$$W_{j\times 16+k} = M_{\pi_j(k)}$$

Comme pour son prédécesseur MD4, on peut observer que puisque l'expansion de message est une permutation tour par tour des mots du message d'entrée M , chaque mot de M est utilisé une fois pour chaque tour. L'expansion de message de RIPEMD-0 est donc presque identique à celle de MD4, sauf que les permutations ont été légèrement modifiées.

Durant chaque étape i, les registres cibles A^L_{i+1} et AR_{i+1} sont mis à jour à l'aide des fonctions f^L_j et f^R_j, dépendantes du tour j auquel appartient i :

$$
\begin{aligned}
A^L_{i+1} &= f^L_j(A^L_i, A^L_{i-1}, A^L_{i-2}, A^L_{i-3}, W_i, s_i) \\
&= (A^L_{i-3} + F_j(A^L_i, A^L_{i-1}, A^L_{i-2}) + W_i + K^L_j)^{<<<s_i}, \\
\\
A^R_{i+1} &= f^R_j(A^R_i, A^R_{i-1}, A^R_{i-2}, A^R_{i-3}, W_i, s_i) \\
&= (A^R_{i-3} + F_j(A^R_i, A^R_{i-1}, A^R_{i-2}) + W_i + K^R_j)^{<<<s_i},
\end{aligned}
$$

Où les K^L_j et K^R_j sont des constantes prédéfinies pour chaque tour et pour chaque branche, les s_i sont des valeurs de rotation prédéfinies pour chaque étape, et les fonctions F_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée. On peut noter que les seules différences entre la fonction de mise à jour de la branche de gauche et de celle de droite sont les différentes constantes utilisées K^L_j et K^R_j. À la fin des 48 étapes des deux branches, on applique l'addition modulaire avec la variable de chaînage pour calculer les mots de la sortie de la fonction de compression :

$$
\begin{aligned}
h'_0 &= A^L_{47} + A^R_{48} + A_{-2}, \qquad & h'_1 &= A^L_{48} + A^R_{45} + A_{-1}, \\
h'_2 &= A^L_{45} + A^R_{46} + A_0, \qquad & h'_3 &= A^L_{46} + A^R_{47} + A_{-3}
\end{aligned}
$$

Une description d'une étape est donnée dans la figure 2.7 et les caractéristiques complètes de la fonction peuvent être trouvées dans [RIP95].

Sécurité de la fonction RIPEMD-0

Comme pour les fonctions MD, des versions réduites de RIPEMD-0 ont été tout d'abord analysées, soit en diminuant le nombre de tours [Dob97], soit en analysant une seule des branches à la fois [DG01]. La première cryptanalyse contre le schéma complet a été publiée en 2004 par Wang et al. [WFLY04, WLF+05], et présente une complexité de 2^{16} appels à la fonction de hachage. Comme pour MD4 ou MD5, l'utilisation de RIPEMD-0 dans une application cryptographique est déconseillée.

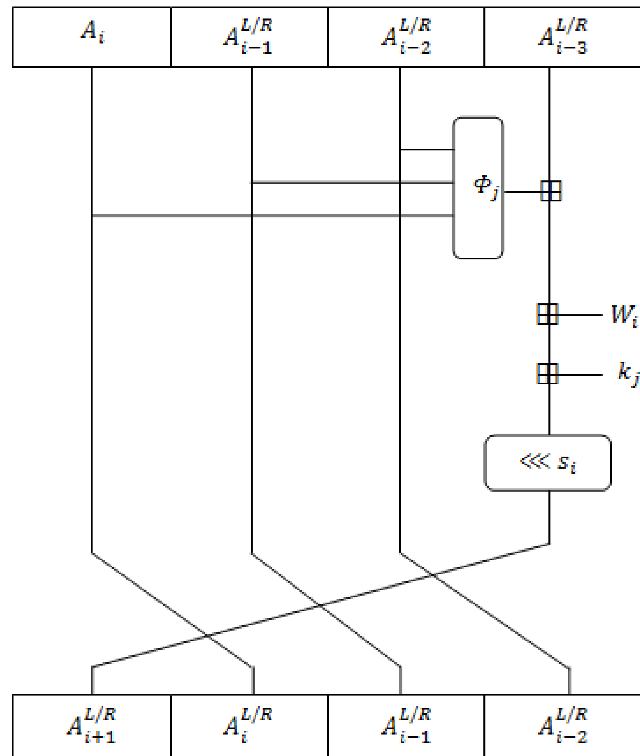

FIGURE 2.7 – Une étape de la fonction de compression de RIPMED-0.

2.3.2 RIPEMD-128

En 1996, Hans Dobbertin, Antoon Bosselaers et Bart Preneel [DBP96] ont proposé une version améliorée de RIPEMD-0 pour résister aux premières cryptanalyses de MD4 et de RIPEMD-0. De plus, une version 256 bits a aussi été définie, mais pour une sécurité équivalante à une fonction de 128 bits. Cette nouvelle primitive comporte toujours 2 branches de 4 registres de 32 bits. On note A_i^L les registres cibles de la branche de gauche et A_i^R ceux de la branche de droite. On initialise chaque branche par la variable de chaînage d'entrée :

$$A_{-3}^L = h_0, \quad A_{-2}^L = h_1, \quad A_{-1}^L = h_2, \quad A_0^L = h_3,$$

$$A_{-3}^R = h_0, \quad A_{-2}^R = h_1, \quad A_{-1}^R = h_2, \quad A_0^R = h_3$$

À l'exécution de la fonction de compression, 16 mots de message seront traités, avec maintenant 4 tours de 16 étapes chacun dans chaque branche (c'est à dire 64 étapes). L'expansion de message est modifiée par rapport à RIPEMD-0 puisque pour chaque tour j et pour chaque branche, des permutations de l'ordre des mots de message π_j^R et πL_j sont définies. Ainsi, nous avons pour la $k^{ième}$ étape du tour j , avec $0 \leq j \leq 3$ et $0 \leq k \leq 15$:

$$W^L_{j\times 16+k} = M_{\pi^L_j(k)}$$

$$W^R_{j\times 16+k} = M_{\pi^R_j(k)}$$

L'expansion de message reposant toujours sur des permutations des mots de message pour chaque tour, chaque mot de M sera utilisé une fois pour chaque tour dans chaque branche. Une des nouveautés par rapport à RIPEMD-0 est donc que l'ordonnancement des mots de message est différent dans les deux branches.

Pendant chaque étape i, les registres cibles A^L_{i+1} et A^R_{i+1} sont mis à jour par les fonctions f^L_j et f^R_j, dépendantes du tour j auquel appartient i :

$$\begin{aligned} A^L_{i+1} &= f^L_j(A^L_i, A^L_{i-1}, A^L_{i-2}, A^L_{i-3}, W_i, s_i) \\ &= (A^L_{i-3} + F_j(A^L_i, A^L_{i-1}, A^L_{i-2}) + W_i + K^L_j)^{<<<s_i}, \end{aligned}$$

$$\begin{aligned} A^R_{i+1} &= f^R_j(A^R_i, A^R_{i-1}, A^R_{i-2}, A^R_{i-3}, W_i, s_i) \\ &= (A^R_{i-3} + F_j(A^R_i, A^R_{i-1}, A^R_{i-2}) + W_i + K^R_j)^{<<<s_i}, \end{aligned}$$

Où les K^L_j et K^R_j sont des constantes prédéfinies pour chaque tour et pour chaque branche, les s^L_i et s^R_i sont des valeurs de rotation prédéfinies pour chaque étape et pour chaque branche, et les fonctions F^L_j et F^R_j sont des fonctions booléennes définies pour chaque tour et pour chaque branche et prenant 3 mots de 32 bits en entrée. On peut noter que les fonctions booléennes F^L_j et F^R_j (et les constantes de rotation s^L_i et s^R_i) utilisées dans chaque branche sont différentes et ceci constitue la deuxième grande différence entre RIPEMD-0 et RIPEMD-128. Du fait du rebouclage, à la fin des 64 étapes des deux branches, les mots de la sortie de la fonction de compression sont calculés par :

$$\begin{aligned} h'_0 &= A^L_{63} + A^R_{64} + A_{-2} & h'_1 &= A^L_{64} + A^R_{61} + A_{-1} \\ h'_2 &= A^L_{61} + A^R_{62} + A_0 & &= h'3 = A^L_{62} + A^R_{63} + A_{-3} \end{aligned}$$

En ce qui concerne la version 256 bits, les deux branches sont gardées séparées durant l'addition modulaire pour obtenir la bonne taille de sortie, un mélange est effectué en inversant certains registres à la fin de chaque tour. Une représentation visuelle d'une étape est donnée dans la figure 2.8 et la description complète de la fonction RIPEMD-128 peut être trouvée dans [DBP96].

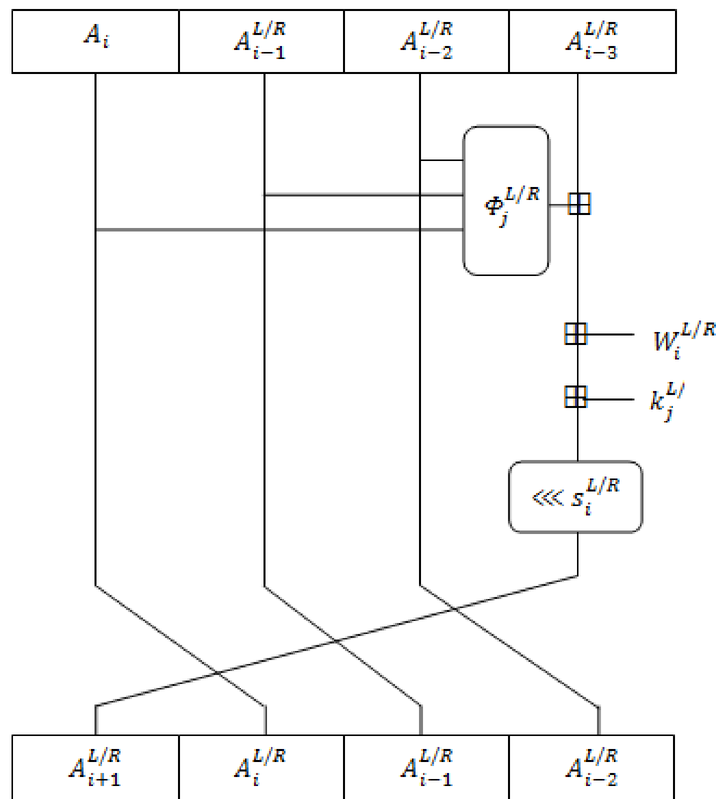

FIGURE 2.8 – Une étape de la fonction de compression de RIPMED-128.

Sécurité de la fonction RIPEMD-128

Jusqu'à présent, RIPEMD-128 et sa version 256 bits n'ont pas été attaquées et peuvent donc être considérées comme sûres. Les différences plus marquées entre les deux branches semblent en effet grandement compliquer le travail des cryptanalystes.

2.4 Conclusion

Dans ce chapitre nous avons présenté les fonctions de la famille MD-SHA, en comançant par les fonctions "Messages Digest", qui ont constitué la base historique des fonctions de hachage cryptographiques. Ces fonctions ont été toutes cryptanalysées et ne doivent pas être utilisé dans les applications cryptographiques. Nous avons ensuite étudier les fonctions "Standard Hash Algorithm" standardisés par le NIST. A la fin nous avons présenté deux fonctions "RIPMED" (RIPMED-0 et RIPMED-128), qui sont très simlaires aux fonctions MD.

Chapitre 3

Automates Cellulaires

Les automates cellulaires sont un modèle puissant de calcul, introduits au début des années 1950 par le mathématicien John Von Neumann, qui s'intéressait alors à l'autoreproduction des systèmes artificiels.

Depuis leur création ils ont été étudiés dans divers domaines comme par exemple la physique ou la biologie, où ils permettent de modéliser et de simuler divers phénomènes qui ne peuvent pas être analysés directement. Ils ont aussi été considérés en tant que systèmes dynamiques discrets car, bien que leur définition soit très simple, ils sont capables de produire des comportements complexes difficiles à prévoir, et fournissent ainsi un outil théorique pour l'étude des systèmes complexes discrets. Ces comportements complexes générés à partir de règles simples rendent les automates cellulaires des candidats idéals pour la conception de primitives cryptographiques.

Les automates cellulaires possèdent une structure intéressante pour atteindre des débits maximum. Cependant, malgré leur simplicité, prédire le comportement d'un CA en fonction de la règle utilisée est très difficile.

Dans ce chapitre nous allons définir les automates cellulaires et montrer leurs caractéristiques favorables à l'utilisation comme base de fonctions de hachage rapides et sûres.

3.1 Définition

Un automate cellulaire (AC) est un vecteur (4-uplet) (Q, d, V, δ) où

- Q est l'ensemble des états,
- $d \in N^*$ est la dimension,
- $V = v_i | i \in [1, |V|]$ est un ensemble fini de vecteurs de Z^d, appelé voisinage,
- $\delta : Q^{|V|} \rightarrow Q$ est la règle de transition.

L'ensemble des cellules est soit $U = Z^d$ pour un automate infini, soit $U = (Z/nZ)^d$ pour un automate cellulaire fini (i.e. ayant un nombre fini de cellules), où n est la taille de l'automate. Dans ce mémoire nous utilisons des automates finis à une dimension, pour lesquels n sera donc le nombre de cellules.

Une cellule est un élément de la grille U, et une configuration est obtenue en fixant l'état de chaque cellule, c'est donc une fonction $c : U \rightarrow Q$. La règle d'évolution associe à la configuration c la configuration $c^{'}$ définie par :

$$c^{'}(z) := \delta(c(z + v_1), ..., c(z + v_{|V|}))$$

C'est donc une évolution :

- Locale : l'état de chaque cellule ne dépend que des états précédents des cellules voisines,
- Parallèle : toutes les cellules évoluent en même temps,
- Uniforme : si toutes les cellules ont la même règle (sinon elle est Non-uniforme).

Donc un AC est système complexe avec une évolution synchrone, déterministe et discrète. Un automate cellulaire peut être vu comme un ensemble de petites machines identiques [POU06] : les cellules, réparties sur une grille régulière (espace discret). Les cellules n'ont qu'une mémoire finie, c'est-à-dire qu'elles ne peuvent être que dans un nombre fini d'états possibles et qu'elles ne portent pas d'autre information que leur état. À chaque étape de temps (temps discret et synchrone), chaque cellule change son état courant en fonction de son état et de celui de ses voisines uniquement. On peut également concevoir des automates plus au moins éloignés de cette définition de base, tels que par exemple :

Automates cellulaires probabilistes. on introduit une variable aléatoire dans la fonction F. Dans le cas où la fonction F ne fait pas intervenir le hasard l'automate est appelé déterministe.

Automates cellulaires non uniformes. ils ne sont pas homogènes. Les cellules ne sont pas toutes soumises aux mêmes règles.

Automates cellulaires asynchrones. toutes les cellules ne sont pas traitées au cours d'un cycle unique.

Automates cellulaires à valeurs réelles. espace d'états infini. Les valeurs stockées dans les cellules ne sont plus des nombres naturels, mais des nombres réels. Elles peuvent être placées dans un intervalle, par exemple $[0; 1]$.

Automates réversibles. la fonction F prend en argument, en plus de l'état du voisinage de la cellule à l'instant $t-1$, l'état de la cellule elle-même à l'instant $t-2$. Cette démarche rend la fonction F réversible.

Architectures non uniformes. les connexions entre les cellules peuvent être plus compliquées que les simples règles de voisinage d'automate cellulaire classique.

Le choix est vaste et l'on peut choisir le type d'automate selon la complexité du problème à résoudre.

3.2 Les automates cellulaires élémentaires

Wolfram [Wol02] un des pionniers dans les recherches sur les ACs comme des modèles mathématiques pour les systèmes statistiques auto-organisateurs, a suggéré l'utilisation des ACs élémentaires, qui représentent les ACs les plus simples avec :

Motif initial du voisinage (t)	111	110	101	100	011	010	001	000
Valeur suivante de la cellule centrale (t+1) (Règle 30)	0	0	0	1	1	1	1	0
Valeur suivante de la cellule centrale (t+1) (Règle 2)	0	0	0	0	0	0	1	0

Table 3.1 – Les règles 30 et 2 d'ACs élémentaires.

- $Q = 0, 1$ (deux états possibles pour chaque cellule).
- $d = 1$ (une seule dimension).
- V = -1,0,1 (un voisinage composé de 3 cellules : la cellule elle-même et ces deux voisines de gauche et de droite).

Donc les ACs élémentaires sont constitués d'un tableau d'une seule dimension composé de n cellules. Chacune des cellules prend un état discret q égal à 0 ou 1, à un instant donné t. Une fonction δ de transition est utilisée pour déterminer l'état suivant $q_i(t+1)$ de la $i^{ième}$ cellule à l'instant $t+1$. Le prochain état d'une cellule à l'instant $t+1$ est influencé par son propre état et les états de ses voisines gauche et droite à l'instant t :

$$q_i(t+1) = \delta q_{i-1}(t), q_i(t), q_{i+1}(t)$$

Où $q_{i-1}(t), q_i(t), q_{i+1}(t)$ sont les états de la cellule i et de ses voisines de gauche et de droite à l'instant t. Pour un AC à 2 états et un voisinage de 3 cellules, il y a $2^3 = 8$ configurations locales possibles du voisinage. Si la configuration d'une cellule en fonction de son voisinage est exprimée sous la forme d'une table de vérité, l'équivalent décimal de la séquence de bits de cette table est désigné comme une "Règle" [Wol02]. Pour chaque configuration locale, il faut choisir le résultat de la règle, il y a donc $2^8 = 256$ règles possibles différentes d'un AC élémentaire (voir Tableau 2.1.). [Wol02]. Par exemple, le code de la règle 30 est obtenu en lisant la deuxième ligne du tableau 3.1 comme un nombre en base 2. $(00011110)_2 = 30$.

En pratique, parmi les 256 règles, on n'en étudie que 88. En effet,

- Si deux règles δ et δ' sont identiques modulo une symétrie de la configuration, c'est-à-dire si $\forall q_1, q_2, q_3 \delta(q_1, q_2, q_3) = \delta'(q_3, q_2, q_1)$, alors en connaissant les diagrammes espace-temps de δ on peut déduire ceux de δ' par une simple symétrie. Cette opération préservant toute les propriétés habituellement étudiées, il n'est donc pas nécessaire d'étudier δ'.
- Si deux règles δ et δ' sont identiques modulo un échange des deux états 0 et 1, c'est-à-dire si $\forall q_1, q_2, q_3 \delta(q_1, q_2, q_3) = 1 - \delta'(1-q_3, 1-q_2, 1-q_1)$ alors en connaissant les diagrammes espace-temps de δ on peut déduire ceux de δ' par un simple échange des deux états. De même, il n'est alors pas nécessaire d'étudier δ'.

En ne gardant qu'un représentant de chaque classe d'équivalence modulo ces deux symétries, il reste 88 règles.

3.2.1 Diagramme espace-temps d'un automate cellulaire

Une méthode efficace pour étudier le comportement des automates cellulaire est de visualiser l'évolution du système sous forme d'un diagramme. Ainsi, pour étudier les propriétés d'évolution d'un automate cellulaire dans le temps, on représente ça dynamique temporelle par un diagramme "espace-temps". Les cellules formant la grille sont représentées en ligne. On "empile" ensuite l'ensemble des configurations à chaque instant de temps $(t_0, t_1, t_2, ..., t_m)$ placées les unes au dessous des autres à partir de la configuration initial. Chaque configuration est représentée par une grille de carrés dont les couleurs représentent les états des cellules (Par exemple le noire représente 1 et le blanc représente 0) (voir la Figure 3.1) :

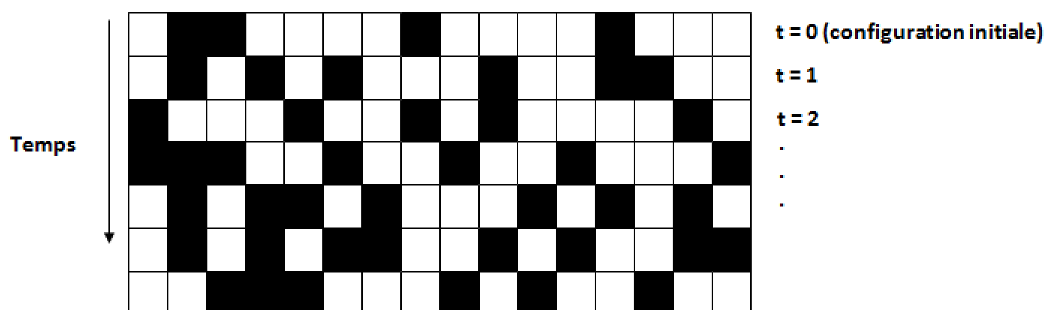

Figure 3.1 – Diagramme espace-temps d'un automate à une dimension et à 2 états.

Un diagramme espace-temps permet d'avoir une idée du comportement global du système en appliquant la règle à toutes les cellules à chaque pas. Le diagramme espace-temps de l'AC élémentaire de règle 90 à partir d'une configuration initiale avec un seul '1' au milieu (représenté par un carré noir) est représenté dans la Figure 3.2.1.

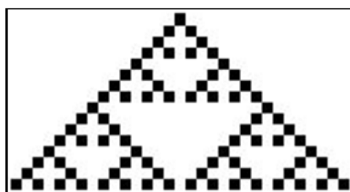

Figure 3.2 – Diagramme espace-temps de l'AC élémentaire 90.

3.2.2 Classes d'automates cellulaires élémentaires

Les automates cellulaires élémentaires présentent une grande variété d'évolutions. La question qui se pose alors est de comment les étudier (et, donc, de les classifier). Une des classifications les plus utilisées est celle proposée par Stephen Wolfram dans [Wol83]. Il a proposé la classification des AC selon leur comportement dynamique, en s'inspirant de la théorie des systèmes dynamiques. Partant d'une étude systématique des AC unidimensionnels à deux états sur un espace constitué de 256 AC.

Cette proposition représente la première tentative pour classifier les ACs d'après leur évolution dans le temps. Si chaque règle conduit à des motifs qui diffèrent dans le détail, tous les AC semblent pouvoir appartenir à seulement quatre classes qualitatives distinctes. Wolfram calcule, pour des configurations initiales typiques (en donnant à chaque cellule un état aléatoire) une portion finie mais significative du diagramme espace-temps, qui représente l'évolution de la configuration, et classe son aspect visuel en quatre catégories : soit on atteint une configuration particulière en un temps fini, soit on boucle sur un ensemble fini de configurations, soit l'évolution semble aléatoire, soit l'évolution est complexe, mais fait émerger des structures particulières.

Cette étude a montré que, Les ACs élémentaires s'intègrent dans quatre classes principales [LAB09] :

1. Classe I : L'évolution de l'automate après certain nombre d'étapes, pour la quasi-totalité des différents états initiaux, tend vers un état homogène où les cellules ont la même valeur. Toute information existante sur l'état initial du système sera complètement détruite, la prédictibilité d'évolution est donc évidente. Partant de l'état initial, le système évolue toujours vers un état homogène. Ce type d'automates cellulaire n'est pas adéquat pour des applications cryptographiques. Un exemple d'AC de la classe I est l'AC 36 (voir Figure 3.3).

2. Classe II : L'évolution de l'automate conduit à un ensemble de structures stables ou périodiques (petite période), mais en tous cas simples et séparées. La prédictibilité d'évolution reste faisable. Les effets d'un état d'une cellule se propagent à un nombre fini de voisins. La modification d'une cellule de la configuration initiale n'affectera qu'une région finie de son entourage. Donc ce type d'AC ne peut pas être utilisé dans des algorithmes cryptographiques. L'AC avec la règle 40 est un automate de classe II (Figure 3.4).

Figure 3.3 – Diagramme espace-temps de l'AC 36 à partir d'une configuration aléatoire.

Figure 3.4 – Diagramme espace-temps de l'AC 40 à partir d'une configuration aléatoire.

3. Classe III : L'évolution de l'automate conduit à un motif chaotique caractérisé par des "attracteurs étranges et des structures apériodiques". Au cours de l'évolution, les cellules propagent les informations à vitesse constante, contrairement à ce qu'on peut observer dans les automates de la classe II. Connaître l'état d'une cellule après un nombre assez grand de pas de temps d'évolution exige la connaissance des états initiaux d'un nombre très grand de cellules. La prédiction de l'évolution n'est donc possible qu'à partir d'un nombre infini d'états initiaux. Et la complexité des figures engendrées y évoque clairement la sensibilité aux conditions initiales. Pour cella, cette classe est la classe la plus adaptée pour des applications cryptographiques. Quelques règles élémentaires d'AC comme la règle 30 (règle utilisée comme base pour le générateur de nombre pseudo-aléatoires du logiciel Mathematica), présentent un comportement d'évolution chaotique et imprévisible. La Figure 3.5. représente l'évolution d'un AC de règle 30 sous forme d'un schéma espace-temps. La règle 18 est aussi une règle chaotique (voir la Figure 3.6).

4. Classe IV : L'évolution de l'automate conduit à un état stable en permettant l'apparition d'un petit nombre de structures complexes stables ou périodiques, parfois persistantes dans le temps. Le jeu de la vie en est le plus représentatif. Le degré du non prédictibilité est encore plus important que dans les automates de la classe

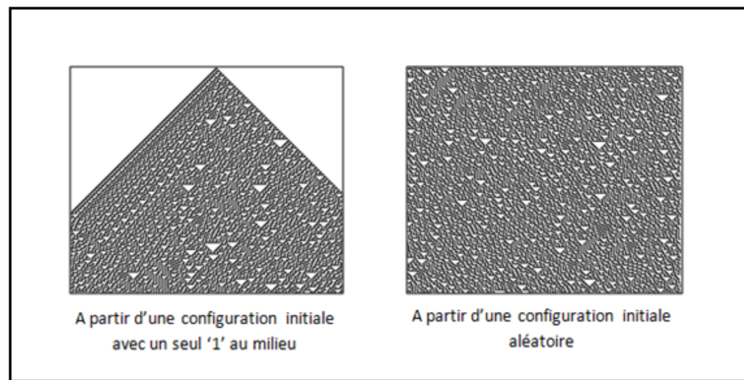

Figure 3.5 – Diagramme espace-temps de la règle 30.

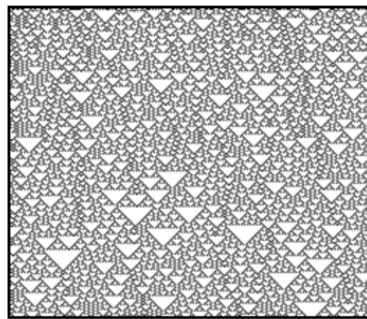

Figure 3.6 – Diagramme espace-temps de l'AC 18 à partir d'une configuration aléatoire.

III. Cette classe peut présenter l'émergence de structures complexes capables d'osciller, de se mouvoir, voire de persévérer plus ou moins dans leur auto-organisation malgré des perturbations structurelles. La Figure 3.7 présente le diagramme de l'AC 20.

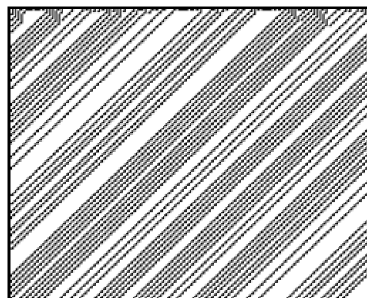

Figure 3.7 – Diagramme espace-temps de l'AC 20 à partir d'une configuration aléatoire.

3.2.3 Automates cellulaires programmables

Automates cellulaires Non-Uniformes. Les automates cellulaires non-uniformes (hybrides) sont une généralisation des automates cellulaires. Tout comme les ACs uniformes, les ACs Non-uniformes agissent en parallèle sur toutes les cellules d'une configuration en fonction de données locales. La différence est que la règle que l'on applique dépend de la position (des règles différentes s'appliquent à des cellules différentes). On n'applique plus uniformément une règle unique.

Un AC programmable (ACP) est un AC Non-Uniforme contrôlé par un certain nombre de signaux pour que des règles différentes puissent être générées. Ce type de construction peut générer des suites aléatoires d'une très bonne qualité cryptographiques, grâce aux capacités de confusion et de diffusion engendrées par le caractère non-uniforme, et à la non linéarité de ces transformations. Pour cela nous avons utilisé ce modèle comme base pour nos propositions. Dans la deuxième partie de cette thèse nous allons détailler nos motivations pour utiliser ce type d'automates cellulaires.

3.3 Automates cellulaires avec mémoire

Un AC avec mémoire est un type d'AC dont l'état d'une cellule à l'instant $t+1$ ne dépond pas uniquement du voisinage de la cellule à l'instant t, mais aussi du voisinage de la cellule à des instants ultérieurs $t-1$, $t-2$,... Les configurations dans les instants $t-1$, $t-2$, ... constituent donc une mémoire pour l'AC. Dans le cas où la mémoire contient uniquement l'état à l'instant $t-1$, l'AC est appelé un AC de second ordre.

Dans cette section nous étudions un type particulier d'ACs avec mémoire. Ce type utilise une mémoire de $(t-1)$ et le voisinage est de 4 cellules (la cellule elle-même et ce deux voisines à l'instant t et la cellule elle-même à l'instant $t-1$). Nous utilisons par la suite cette construction pour concevoir notre fonction de hachage avec clé secrète.

Dans ce cas la il y a $2^{16} = 65536$ règles différentes d'AC. Parmi ces règles, il existe un sous-ensembles de règles non-linéaires avec de bonne propriétés cryptographiques [LMS08].

Une fonction booléenne avec 4 variables est définit par un entier entre 0 et 65536 (avec la notation de Wolfram pour les AC élémentaires), par exemple, la fonction (34680) avec la représentation binaire 100001110111000 correspond à la table de vérité représentée par le Tableau 3.2.

Pour classifier ces fonctions, on peut utiliser leurs forme normale algébrique

x_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$f(x_1, x_2, x_3, x_4)$	0	0	0	1	1	1	1	0	1	1	1	0	0	0	0	1

Table 3.2 – Représentation de la règle 34680.

f	FNA
34680	$x_1x_2 \oplus x_3 \oplus x_4$
6120	$x_4 \oplus x_1x_2 \oplus x_1x_3 \oplus x_2x_3$
7140	$x_2 \oplus x_4 \oplus x_1x_2 \oplus x_1x_3$
11730	$x_1 \oplus x_3 \oplus x_4 \oplus x_1x_2$
34740	$x_2 \oplus x_3 \oplus x_4 \oplus x_1x_2 \oplus x_4x_2$
39318	$x_1 \oplus x_2 \oplus x_3 \oplus x_4x_3x_4$
7128	$x_3 \oplus x_4 \oplus x_1x_2 \oplus x_3x1 \oplus x_4x_2 \oplus x_4x_3$

Table 3.3 – La forme FNA de quelques règles d'AC avec mémoire.

(FNA), par exemple, la FNA de la règle (280) (100011000 en binaire) correspond au polynôme :

$$f(x_1, x_2, x_3, x_4) = x_1x_2 \oplus x_3 \oplus x_4$$

Le tableau 3.3 suivant montre quelques règles chaotiques [LMS08] et leurs représentations FNA.

3.4 Automates cellulaires à deux dimensions 2D

Dans cette section nous allons présenter les automates cellulaires à deux dimensions avec 2 états. Dans ce cas un AC 2D est une grille à 2 dimensions (matrice binaire, voir la Figure 3.8).

Les automates cellulaires à deux dimensions ont une histoire relativement ancienne, de l'automate autoreproducteur de von Neumann qui date des années quarante, en passant par le "jeu de la vie" dû à Conway (1960). La faculté de suivre le mécanisme des interactions entre les automates et la possibilité d'observer l'état du réseau à un instant donné de l'itération en font des systèmes dynamiques discrets précieux.

Dans les ACs à deux dimensions les cellules sont arrangées dans des grilles de 2 dimensions avec des connections entre les cellules voisines.

Deux types de connectivité plus ou moins étendue sont principalement étudiés. Dans le voisinage de von Neumann, chaque automate a cinq entrées : lui-même plus

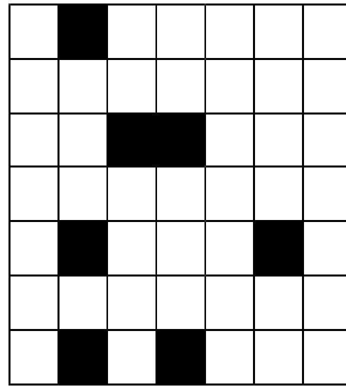

Figure 3.8 – Un Automate Cellulaire à 2 dimensions.

ses quatre voisins. Dans le voisinage de Moore, il en a neuf : lui-même plus ses huit plus proches voisins (voir la Figure 3.9).

Figure 3.9 – Voisinage de Von Neumann et Voisinage de Moore.

L'état de l'AC à n'importe quel moment peut être représenté par une matrice binaire de $N \times M$. Dans le cas général, la fonction du voisinage qui spécifie l'état d'une cellule à l'instant $t+1$ est influencé par l'état de la cellule elle-même et les états de ces 8 voisine directes à l'instant t. Mathématiquement, l'état q de la cellule (i, j) à l'instant $t+1$ est donnée par :

$$q_{i,j}(t+1) =$$
$$f[q_{i-1,j-1}(t), q_{i-1,j}(t), q_{i,j-1}(t), q_{i,j)}(t), q_{i+1,j+1}(t), q_{i+1,j}(t), q_{i,j+1}(t), q_{i-1,j+1}(t), q_{i+1,j-1}(t)]$$

Où f est une fonction de 9 variables (un voisinage de 9 cellules : voisinage de Moore), la sortie de la fonction est soit 0 soit 1. Donc le nombre de règles possibles est $2^{2^9} = 1.3407807929942597099574024998206e+154$ qui est chiffre astronomique. Pour cella nous allons utiliser un sous-ensemble de règles de voisinage (la notion de règle ici est différente de celle de Wolfram) avec la notation de A. Khan [KCa99] décrite comme suit (Figure 3.10) :

La cellule centrale représente la cellule actuelle, et les autres cellules représentent les 8 voisines les plus proches de la cellule. Le nombre dans chaque cellule représente le numéro de la règle associée avec ce voisinage particulier.

64	128	256
32	1	2
16	8	4

Figure 3.10 – Règles d'AC à 2 dimensions.

Dans le cas où la cellule dépond de 2 cellules ou plus, le numéro de la règle sera la somme arithmétique des numéros des cellules correspondantes, ce qui donne des règles non linéaire d'AC.

Selon la règle numéro 2 la cellule centrale sera à 1 dans l'instant $t+1$, si et seulement si sa voisine de droite est à 1 à l'instant t. Selon la règle numéro 8 la cellule centrale sera à 1 dans l'instant $t+1$, si et seulement si sa voisine du bas est à 1 à l'instant t.

Par exemple, la règle 8 peut être illustrée par la figure 3.11 :

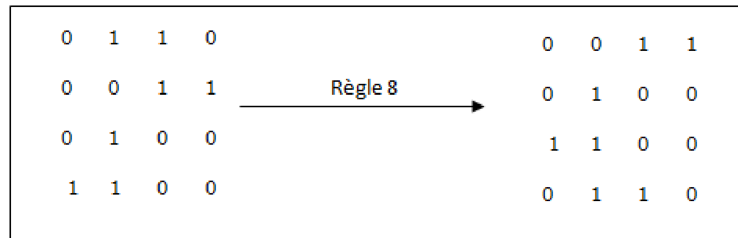

Figure 3.11 – Exemple d'application de la règle 8.

C.-à-d. si l'état de la cellule à l'instant t, ne dépond que de l'état de la cellule elle-même à l'instant $t-1$, alors cela est noté comme la règle "1" (ce qui représente 4 règles possibles pour la notation de Wolfram : $0 \to 0; 0 \to 1; 1 \to 0; 1 \to 1$). Si l'état d'une cellule à l'instant t dépond de l'état de la cellule elle-même et de l'état de sa voisine de droite la règle est égale à 3 (1+2). Si l'état de la cellule dépond de la cellule elle-même est ses voisines de droite, de gauche, du haut et du bas, alors la règle est 171 (1+2+8+32+128). Il faut noter qu'avec ce type de voisinage, l'état d'une cellule à l'instant t dépond du nombre de voisines à 1, mais pas de leurs positions.

Nous allons nous intéresser dans ce travail à une structuration en 2D des cellules qui sont placées sur une matrice ou grille de dimension $N \times N$ (le nombre de cellules vaut donc N^2). Chaque cellule possède un voisinage carré de coté v centré sur elle-même. Ce voisinage est tel que la grille est toroïdale [POU06], (le haut est relié au

bas, le coté droit au coté gauche). Une cellule a donc toujours un voisinage de 9 cellules. La fonction de transition locale δ détermine le nouvel état d'une cellule en fonction des états perçus.

3.5 Automates cellulaires et cryptographie

Les ACs représentent une approche intéressante pour la conception des primitives cryptographiques. Ce sont des systèmes logiques modulaires, simples et hautement parallèles porteurs de comportements globaux complexes. Les ACs peuvent générer des séquences de bits pseudo-aléatoires de bonne qualité pour les systèmes cryptographiques robustes. Un autre avantage des ACs est qu'ils peuvent être facilement et efficacement implémentés dans des systèmes VLSI (Very-Large-Scale Integration) comme les cartes à puce, et pourraient donc trouver des applications dans de nombreux domaines.

Donc les automates cellulaires sont des systèmes dynamiques dont la caractéristique principale est de dépendre des règles pouvant être très simples, mais dont le comportement global résultant peut être très complexe. Ils constituent un nouveau paradigme caractérisé principalement par un traitement des problèmes selon une approche ascendante (du simple vers le complexe), parallèle et en déterminant les comportements des entités élémentaires de façon locale.

Cependant, la difficulté de concevoir un automate pour générer un comportement spécifique ou effectuer une tâche particulière a souvent limité leurs applications [LAB09]. Ce problème est connu sous le nom du problème inverse où il est demandé de trouver un automate cellulaire (règles locales représentant des interactions entre des entités d'un niveau local) possédant des propriétés globales présélectionnées (comportement observé au niveau global comme le comportement chaotique).

Donc le problème principal pour l'utilisation des AC dans des applications cryptographiques est de trouver les meilleures règles d'interactions qui produisent de bonnes suites pseudo-alétoires.

Les ACs ont été utilisés comme base de construction de plusieurs systèmes cryptographiques, tels que les systèmes de cryptographie symétrique, les systèmes de cryptographie à clé publique, les schémas de partage du secret et les fonctions de hachage. La première application des ACs à la cryptographie a été proposée par Wolfram dans [Wol85]. Dans ce travail, Wolfram décrit un chiffrement par flux basé sur un automate cellulaire élémentaire avec la règle 30. L'AC a été utilisé comme un générateur de nombres pseudo-aléatoires (GNPA) pour générer la clé de chiffre-

ment. Hortensius et al. [HMC89] et Nandi et al. [NKC94] ont utilisés des automates non-uniformes avec les regles 90 et 150, et ont démontré la supériorité de la qualité des GNPAs a celle du système de Wolfram.

Plus récemment, Tomassini et Perrenoud [TP00] ont proposé d'utiliser des automates non uniformes unidimensionnels avec les quatre règles 90, 105, 150 et 165, qui fournissent des GNPAs de très bonne qualité et un très grand espace de clé secrètes possible qui rend difficile la cryptanalyse. Au lieu de concevoir eux-mêmes les règles pour les ACs, ils ont utilisé une technique évolutive appelée programmation cellulaire pour rechercher les règles.

Cette technique à été ensuite généralisée à des règles de rayon supérieur à un par Seredynski et al [SBZ04] qui ont à leur tour proposé différentes règles de voisinage 3 et 5 d'automates cellulaires ; ce sont les règles 30, 86 et 101 de voisinage 3 et les règles 869020563, 1047380370, 1436194405, 1436965290, 1705400746, 1815843780, 2084275140 et 2592765285 de voisinage cinq. Leur nouvel ensemble de règles a été testé selon les directive du FIPS 140-2 [oST97].

D'autres chercheurs comme Tomassini et Sipper [TS00], ont investigué l'utilisation des automates bidimensionnels (2D) pour les systèmes de chiffrement par flux.

Les AC ont également été proposés pour le chiffrement par bloc en utilisant les règles réversibles de second ordre [CCZ05]. Les règles réversibles sont introduites pour pouvoir décrypter les messages cryptés. Des systèmes à clé publique basées sur les automates cellulaires ont été aussi proposés dans la littérature. Les ACs pour les crypto-systèmes à clé publique furent proposés la première fois par Guan [Gua87] et Kari [Kar92]. Dans de tels systèmes deux clés sont requises : une clé pour l'encodage et une autre pour le décodage.

Suite aux travaux de Kari [Kar92], montrant l'indécidabilité du problème de savoir si un automate cellulaire en dimension au moins égale à deux est inversible, plusieurs autre systèmes cryptographiques à clés publiques basés sur les automates cellulaires ont été proposés. La confidentialité d'un système cryptographique à clés publiques est basée en général sur la difficulté de trouver la fonction inverse de la fonction de chiffrement. Le calcul de cet inverse peut se faire cependant très facilement si on connaît une brèche secrète dans la construction de la fonction de chiffrement. En ce qui concerne les crypto-systèmes basés sur les automates cellulaires, la clé publique utilisée pour crypter les messages est un automate cellulaire. Pour déchiffrer le message, il suffit d'appliquer au message crypté l'automate inverse de celui utilisé pour le chiffrement.

Or il n'existe pas de procédure connue pour calculer l'inverse d'un automate

cellulaire de dimension supérieure ou égale à 2. La brèche secrète utilisée généralement est le fait que l'automate de chiffrement est la composée d'un certain nombre d'automates triviaux que l'on sait inverser facilement.

Les ACs ont été proposés aussi pour les schémas de partage de secret [dRMS05] par A. Martin del Rey. Ils ont utilisés un système basé sur les ACs avec Mémoire. Les schémas de partage de secret sont des procédures cryptographiques pour partager un secret parmi un ensemble de participants de telle manière que seuls certains sous-ensembles qualifiés de ces participants peuvent recouvrer le secret.

Les automates cellulaires sont également utilisées pour concevoir des fonctions de hachage. Les approches utilisant les AC pour construire des fonctions de hachage sont discutées en détail dans le chapitre suivant.

3.6 Conclusion

Dans ce chapitre, nous avons commencé par présenter les automates cellulaires qui sont des systèmes dynamiques discrèts. Un des aspects les plus intéressants de ces systèmes est l'émergence de propriétés globales qui ne peuvent pas être directement déduites de l'analyse des comportements locaux des règles individuels. Un sous-ensemble de ces systèmes pocèdent un comportement chaotique complexe, qui favorise leur utilisation comme base pour les systèmes cryptographique robusts.

Nous avons identifié, dans ce chapitre, les différents types d'ACs avec un accent particulier sur les ACs élémentaires qui constituent la base pour notre première proposition, et les ACs avec mémoire, sur les quelles se base notre deuxième proposition.

A la fin nous avons présenté une synthèse sur l'utilisation des ACs dans la construction des primitives et protocoles cryptographiques (Symétriques et Asymétriques).

Chapitre 4

Fonctions de hachage basées sur les Automates Cellulaires

Les automates cellulaires (ACs) sont très utiles pour concevoir des fonctions de hachage sûres, avec une basse complexité matérielle et logicielle en raison des attributs de leurs opérations logiques. Les AC fournissent également un haut niveau de parallélisme et par conséquent, ils sont en mesure d'atteindre des vitesses d'exécution très élevées.

Dans ce chapitre nous étudions quelques propositions de fonctions de hachage cryptographiques basées sur les ACs.

4.1 Proposition de Damgard

4.1.1 Description de l'approche

Damgard a été le premier à proposer une fonction de hachage basée sur les ACs [Dam90]. Dans ce travail Damgard a proposé pour la première fois l'utilisation de la construction de Merkle-Damgard (Figure 4.1).

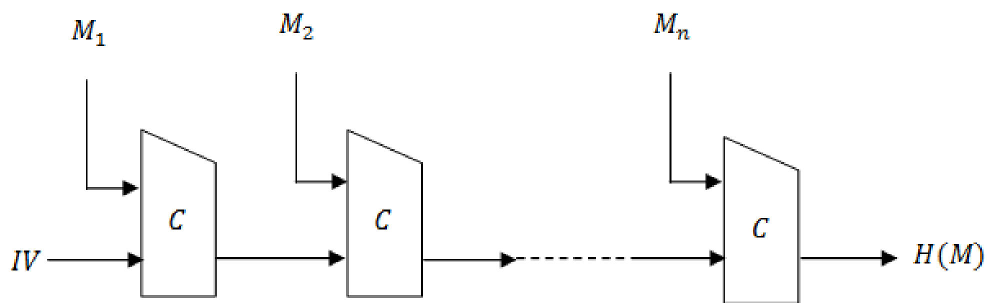

FIGURE 4.1 – Construction de Merkle-Damgard.

La fonction de compression F (4.2) utilise le générateur de bits pseudo-aléatoires de Wolfram [Wol86], basé sur l'AC élémentaire de règle 30. L'equation 4.1 représente la règle 30 en opérateurs logiques.

$$x_i(j) \leftarrow x_{i-1}(j-1) \oplus [x_{i-1}(j) \vee x_{i-1}(j+1)] \tag{4.1}$$

Le générateurs de bits $b(x)$ commence à partir d'une valeur aléatoire de x et génère la séquence $x_i(0)$. Soit $b_{c-d}(x)$ la chaine binaire $x_c(0), x_{c+1}(0), ..., x_d(0)$. Damgard a esseyer d'utiliser la propriété de sens unique de ce générateur pour concevoir une fonction de hachage résistante aux collisions, de la manière suivante :

$$f = b_{c-d}(X_i|H_{i-1}|Z) \tag{4.2}$$

Où Z est une chaine binaire aléatoire de r bits, ajoutée pour soutenir la résistance aux collisions. Les valeurs des paramètres proposées par Damgard sont les suivantes : $s = 512$, $r = 256$, $c = 257$, et $d = 384$. l'avantage de cette construction est qu'elle est très rapide.

F prend en entrée un bloc de message de 128 bits et une variable de chainage de 128 bits, et produit en sortie une chaine de 128 bits (voir figure 4.2).

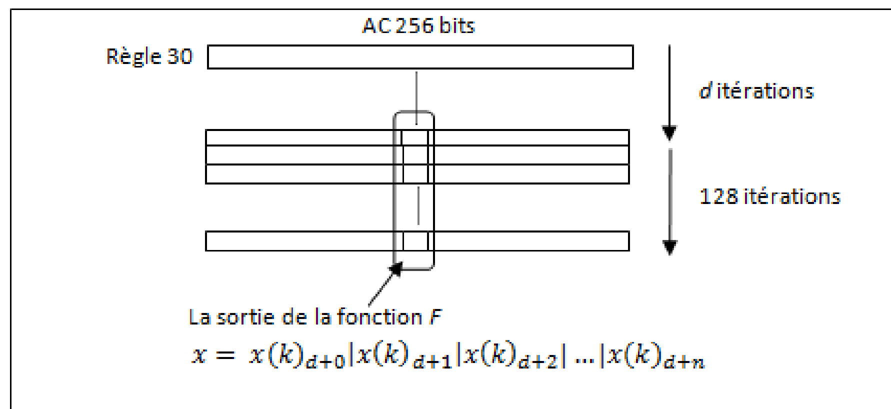

FIGURE 4.2 – Générateur de nombres pseudo-aléatoires de Wolfram utilisé par Damgard.

Cette construction est simple et très rapide, spécialement si elle est implémentée directement sur le matériel, mais elle a été cryptanalysée par Daemen et al. [DGV91].

4.1.2 Attaque de Meier et Staffelbach

Nous détaillons dans ce qui suit l'attaque de Meier et Staffelbach [MS91] connus contre le générateur de bits pseudo-aléatoires de la règle 30. Cette attaque a permis à Daemen et al. [DGV91] de cryptanalyser la fonction de Damgard.

Le but de l'attaque est de déduire l'état initial de l'automate cellulaire, à partir de la suite pseudo-aléatoire générée. La méthode proposée dans [MS91] est la suivante : On s'intéresse à la suite $\{x_i\}_t$ des valeurs prises par la cellule x_i étant donnée une configuration initiale $S(t) = \{x_{i-n}^t, ..., x_i^t, ..., x_{i+n}^t\}$, la clé.

L'évolution de l'automate cellulaire est régie par la règle 30 qui peut être réécrite en utilisant la linéarité partielle :

$$x_{i-1}^t = x_i^{t+1} \oplus (x_i^t \vee x_{i+1}^t) \qquad (4.3)$$

Par l'équation 4.3 , les valeurs des cellules autour de x_i forment un triangle (voir la figure 4.3).

Etant données les valeurs des cellules de deux colonnes adjacentes, la complétion arrière [MS91] permet de reconstruire par 4.3 la totalité du triangle à gauche de la suite $\{x_i\}_t$. Par la complétion arrière, $N-1$ valeurs de $\{x_i\}_t$ et $N-2$ valeurs de $\{x_{i+1}\}_t$ déterminent la configuration initiale. De manière similaire, la clé peut être reconstruite à partir de $N-2$ valeurs de $\{x_i\}_t$ et $N-1$ valeurs de $\{x_{i-1}\}_t$.

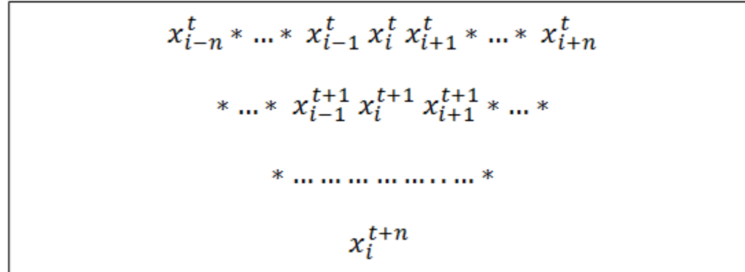

Figure 4.3 – Triangle déterminé par la configuration initiale.

Si on se donne $N-1$ valeurs de la suite $\{x_i\}_t$, connaître la clé équivaut à la connaissance d'une des suites adjacentes. Celles-ci peuvent être considérées chacune comme une clé déterminant le reste de la suite. Le problème devient : trouver une suite adjacente puis déterminer la clé par complétion arrière.

Il y a certaines différences entre les suites adjacentes gauches et droites que nous expliquons pour un automate cellulaire général de largeur $2n+1$ sans condition de bord. On suppose connue $\{x_i\}_t^{t+n}$. Selon la Figure 4.3, le problème de trouver la suite adjacente gauche est équivalent par 4.3 à celui de compléter la totalité du triangle gauche. Réciproquement, la connaissance de $L = \{x_{i-n}^t, ..., x_{i-1}^t\}$ et de $\{x_i\}_t$ équivaut à la connaissance de tout le triangle gauche.

Remarquons que les valeurs de L ne peuvent pas être choisies au hasard. Par exemple, si $x_i^t = 1 \Rightarrow x_{i-1}^t = x_i^{t+1} + 1$.

D'un autre coté, tout choix de $R = \{x_{i+1}^t, ..., x_{i+n}^t\}$ mène à une complétion consistante vis-à-vis de $\{x_i\}_t$. En effet, par (4.3), pour tout élément de la suite adjacente droite x_{i+1}^t il existe un élément de la suite adjacente gauche x_{i-1}^t consistant avec la valeur suivante de la suite de référence x_i^{t+1}. De plus, selon (4.3), n'importe quel choix de $\{x_{i+2}^t, ..., x_{i+n}^t\}$ mène à une extension consistante en $\{x_{i+2}^{t+1}, ..., x_{i+n-1}^{t+1}\}$. En itérant ce procédé, on obtient le triangle de droite de la figure 4.3. Ce procédé, appelé complétion avant, construit une suite adjacente droite consistante avec la suite de référence pour tout choix de R.

Supposons à présent que l'automate cellulaire est un anneau de N cellules (ce qui signifie en particulier que $L = R$). Pour retrouver la clé, on peut calculer soit $R = \{x_i^t, ..., x_{i+N-1}^t\}$, soit $L = \{x_{i-N+1}^t, ..., x_i^t\}$. En s'appuyant sur les remarques précédentes, on obtient l'algorithme MS suivant :

Algorithme MS

1. engendrer une clé aléatoire ;
2. complétion avant :
   Pour $k \in \{1, ..., N-2\}$
   Pour $j \in \{1, ..., N-k-1\}$
   $x_{i+j}(t+k) \leftarrow x_{i+j-1}(t+k-1) \oplus (x_{i+j}(t+k-1) \vee x_{i+j+1}(t+k-1))$;
3. complétion arrière : Pour $j \in \{1, ..., N-1\}$
   Pour $k \in \{N-1-j, ..., 0\}$
   $x_{i-j}(t+k) \leftarrow x_{i-j+1}(t+k+1) \oplus (x_{i-j+1}(t+k) \vee x_{i-j+2}(t+k))$;
4. exécuter la règle 30 sur la configuration initiale obtenue ci-dessus pour générer la suite. Si coïncidence Fin, sinon aller à 1.

Le fonctionnement de l'algorithme est illustré sur un exemple avec N = 5 par la figure 4.4. Dans ce cas, la clé est $(x_i, ..., x_{i+4}) = (0, 1, 0, 1, 1)$ et la suite pseudo-aléatoire $(0, 0, 1, 0, 0)$.

A l'étape 1 de l'algorithme, les valeurs $(x_i, ..., x_{i+4})$ sont choisies au hasard. Si x_{i+1} est à 1, alors par 4.3, la suite adjacente droite produite est indépendante du choix de x_{i+2}, x_{i+3} et x_{i+4}. C'est pour cela qu'ils sont marqués par "*" dans la figure 4.4. Ainsi, il n'y a qu'une seule suite adjacente droite avec $x_{i+1} = 1$. Donc, avec probabilité 1/2, la clé correcte est trouvée dès le premier essai.

4.2 Proposition de Daemen et al. (CellHash)

Daemen et al. qui ont cryptanalysé la proposition de Damgard ont proposé une autre fonction de hachage [DGV91]. Leur proposition est appelé CellHash qui est une fonction de hachage orienté matériel et capables d'attendre des vitesses extrêmement élevées. La fonction de compression est composée de deux automates cellulaires et une permutation pour garantir la confusion et la diffusion des bits du message.

La valeur du Haché h d'un message M de longueur n est calculé en deux phases :

– Préparation du message : le message est étendu par une suite de 0 de telle sorte que sa longueur est au moins 248 et congrue 24 Modulo 32. Le message obtenu aura une longueur qui est multiple de 32 bits. Le message pourra donc être écrit comme $M_0, M_1, ..., M_{N-1}$ c-à-dire une concaténation de N bloc de 32 bits chacun.
– Application de la fonction de compression : $F_c(H, A)$ qui est une fonction d'arguments H, une suite binaire de 257 bits et A, une suite binaire de 256

x_{i-4}	x_{i-3}	x_{i-2}	x_{i-1}	x_i	x_{i+1}	x_{i+2}	x_{i+3}	x_{i+4}
1	0	1	1	0	1	0	1	1
	0	1	0	0	1	0	1	
		1	1	1	1	0		
			0	0	0			
				0				

Génération de la suite temporelle

x_{i-4}	x_{i-3}	x_{i-2}	x_{i-1}	x_i	x_{i+1}	x_{i+2}	x_{i+3}	x_{i+4}
1	0	1	1	0	1	*	*	*
	0	1	0	0	1	*	*	
		1	1	1	1	*		
			0	0	0			
				0				

Détermination de la racine par complétion arrière

FIGURE 4.4 – Un exemple simple de cryptanalyse de la règle 30.

bits. Cette fonction donne en sortie une chaine binaire de 257 bits. La fonction de hachage est ensuite définie comme :

- $H_0 = IV$
- $H_j = F_c(H_{j-1}), M_{j-1}, M_{jmodN}...M_{j+6modN}), \quad j = 1..N$
- H_N est la valeur du haché.

Description de la fonction $F_c(H, A)$

Le calcul peut être considéré comme 5 transformations de H. Le calcul dans chaque transformation est simultanément appliqué a tous les bits du H. Soit $h_0, h_1, ..., h_{256}$ dénotent les bits de H et $a_0, a_1, ..., a_{256}$ dénotent les bits de A.

- Etape 1 : $h_i = h_i \oplus (h_{i+1} \vee \overline{h}_{i+2}), \quad 0 \leq i \leq 257$
- Etape 2 : $h_0 = \overline{h}_0$
- Etape 3 : $h_i = h_{i-3} \oplus (h_i \oplus h_{i+3}), \quad 0 \leq i \leq 257$
- Etape 4 : $h_i = h_i \oplus a_{i-1}, \quad 1 \leq i \leq 257$
- Etape 5 : $h_i = hi * 10, \quad 0 \leq i \leq 257$

L'étape 1 est une opération d'AC non linéaire où chaque variable h_i est mis à jour selon les valeurs de ces voisins. Cette non-linéarité assure la confusion des bits. L'étape 2 consiste à éliminer la symétrie circulaire dans le cas ou tous les h_i sont à 0. L'étape 3 est une opération d'automate cellulaire linéaire qui assure la diffusion.

Dans l'étape 4 les bits du message sont injectés dans H. l'étape 5 est une permutation de bits où les bits sont placé loin de leurs voisins. La longueur de H est 257 bits (un nombre premier), pour rendre les étapes 1 et 3 inversibles.

Les mêmes auteurs ont proposé plus tard une autre fonction appelée Subhash [DGV92], qui est une version améliorée de CellHash ; les deux fonctions CellHash et Subhash sont orientées matériel, mais elles ont été cryptanalysées dans [Cha06].

4.3 Proposition de Mihaljevic, et al.

Une autre recherche sur les fonctions de hachage à base d'AC a été rapporter par Mihaljevic, et al. [MZI98b]. Ils ont proposé une famille de fonctions de hachage dédiés et rapides, basées sur des AC linéaires sur les corps fini de cardinal $q(GF(q))$. La proposition est une extension de la fonction de hachage qui a été proposé plus tôt dans [MZI98a] et emploie le modèle approuvé de fonction de hachage itérative, et deux type de fonctions : une fonction de compression et une fonction de sortie. Leur fonction de compression est l'une des types de Davies-Meyer employant les ACs et leur fonction de sortie est un générateur de clé qui est également basé sur les ACs sur $GF(q)$.

Voici l'algorithme général de leur proposition :

1. **Les entrées.** Le message M, et la valeur initiale IV de n-mots.
2. **Prétraitement.** Ajouter un padding et diviser les messages en blocs de m-mots : $M = (M_1, M_2, ...M_m)$.
3. **Traitement itérative.** Mettre $H_0 = IV$, pour chaque $i = 1, 2, ..., m$, faire : Calculer la valeur de la fonction de compression $h()$: $H_i = h(M_i, H_{i-1})$
4. si H_m est un vecteur dont tous les bits sont à 0, recalculer H_m comme suit : $H_m = h(M_m, H_0)$.
5. **La fonction de sortie.** Calculer (H_m).
6. **La sortie.** Haché de n-bits : $hash(M) = g(H_m)$.

La fonction de compression hash() : pour définir la fonction de compression les auteurs utilisent les notions suivants :

n est le nombre de mots de chaque M_i et l est le nombre de bits dans chaque mot. $M_{i,k}$ est le $k^{ième}$ mot de M_i, qui est un élément de $GF(2^l)$. $H_{i-1,k}$ est le $k^{ième}$ mot de H_{i-1}, qui est aussi un élément de $GF(2^l)$. $f_k()$, k=1, 2, ... sont des fonctions qui transforment, d'une façon non linéaire 2 éléments de $GF(2^l)$ en un élément de

$GF(q)$ tel que q est un nombre premier. $CA()$ est un opérateur qui transforme l'état courant d'un AC vers l'état suivant.

La fonction est ensuite définit comme suit :

1. Une combinaison non-linéaire avec compression de M_i et H_{i-1} : la transformation $\{0, 1, ..., 2^{l-1}\}^2 \to 0, 1, ..., q-1$ génère un vecteur n-dimensionnel X_i avec des éléments $X_{i,k}$ de $GF(q)$. Selon l'équation : $X_{i,k} = f_{((M_{i,k}+H_{i-1,k})mod \quad l)mod \quad k}(M_{i,k}+H_{i-1,k})$
2. Application des opérations de l'automate cellulaire : générer un vecteur n-dimensionnel Y_i à partir de $GF(q)$. $Y_i = CA(X_i)$.
3. Appliquer des permutations non-linéaires
4. Une deuxième application des opérations d'automate cellulaire : générer un vecteur n-dimensionnel Z_i à partir de $GF(q)$. $Z_i = CA(Y_i^{'})$.
5. Appliquer des transformations non-linéaires et une compression.

La fonction de sortie g() : La fonction de sortie transforme H_m en un vecteur binaire n-dimensionnel. La fonction $g()$ est une variante du générateur de clé par flux basé sur les automates cellulaires, proposé et analysé dans [MZI98a]. L'argument en entrée du générateur est une transformation de H_m en un vecteur binaire n-dimensionnel, qui sert de clé secrète, selon la règle suivante : le ième bit du vecteur binaire est la somme modulo 2, des bits dans le ième mot de H_m. En utilisant cette clé la fonction $g()$ génère une sortie de n bits.

Les parties principles du générateur de clé qui réalisent la fonction de sortie sont les suivantes : un AC de n cellules sur $GF(2)$, une mémoire ROM qui contient les règles de l'AC (les règles sont nommées R_0,R_1,...), un buffer binaire de taille n et une permutation binaire n-dimensionnel. La fonction de sortie est définie comme suit :

– Initialement l'AC est configuré avec la règle R_0, avec cette configuration l'AC s'exécute une fois. Ensuite il est reconfiguré avec la règle R_1, ensuite R_2 et ainsi de suite. A chaque fois l'automate s'exécute une seule fois.
– Après chaque exécution de l'AC la cellule centrale de l'AC est prise et stockée dans le buffer binaire.
– Après n exécution le contenu du buffer est permuté selon une permutation contrôlée par l'état de l'AC.

Les auteurs ne donnent pas assez de détails sur les paramètres de l'algorithme, pour cela une étude cryptanalyse complète ne peut pas être faite.

4.4 Proposition de Dasgupta et al.

Dasgupta et al. [DCS01] ont proposé un système à base d'AC pour l'authentification des messages (fonction de hachage avec clé secrète). Ils ont étudié une classe particulière d'AC non-groupe, qui peut être utilisé pour générer une fonction efficace d'authentification de message.

L'AC est initialement chargé par la clé d'authentification secrète. Pour une clé de n bits la longueur de l'AC est de n cellules. Les n bits du bloc du message sont utilisés pour contrôler la configuration des cellules individuelles de l'AC. Si le $i^{ème}$ bit du message est à 0 alors, la $i^{ème}$ cellule est configurée pour exécuter la règle 90, sinon elle exécute la règle 150. Après la configuration des règles par les n bits, l'AC est exécuté une seule fois, ensuite l'AC est reconfiguré avec les n bits suivants du message, et ainsi de suite.

Pour augmenter la complexité de la cryptanalyse de leur proposition, les auteurs utilisent un algorithme nommé 90/150 TPSA CA. Cet algorithme est défini comme suit :

L'algo 90/150 TPSA CA :

Construct-CA (n)

Entrées : n, la taille de l'AC

Sorties : n bits de l'AC avec un ensemble de règle r[i], $i = 1..n$.

début

si $n = 1$ alors $r[i] := 90$; sinon si $(n = 2)$ alors

{ r[1] :=150 ; r[2] :=150 ; }

Sinon si $(n = 3)$

{ r[1] :=90 ; r[2] :=90 ; r[3] :=90 ; }

Sinon si $(n = 2k)$ /*n est pair*/ alors

{ Diviser l'AC en 2 parties de longueur k

Pour chacune des parties :

Appeler récursivement Construct-CA (k)

Changer la règle de la cellule k et la cellule $k + 1$ du 90 vers 150 et vice-versa }

Sinon si $(n = 2k + 1)$ /*n est impair*/ alors

{ Diviser l'AC en 2 parties de longueur k

r$[k + 1]$:=90 ;

Pour chacune des parties : Appeler récursivement Construct-CA (k)

}

FinSi ;

FinSi ;

Fin.

Les étapes de la primitive d'authentification :

- Etape 1 : générer n bits d'AC en exécutant l'algorithme TPSA CA.
- Etape 2 : générer l'AC complémentaire pour avoir un vecteur F.
- Etape 3 : soit k la clé secrète de longueur n, et soit M la longueur du message. Ajouter un padding de tel sorte que la longueur totale de la clé, du message et du padding soit un multiple de n. Le premier élément du tableau de données est maintenant la clé.
- Etape 4 : charger l'AC par la clé.
- Etape 5 :
  - Prendre la chaine de taille n suivante du message, et reconfigure l'AC avec cette chaine binaire, ensuite, exécuter l'AC une seule fois.
  - Exécuter l'AC une seule fois avec le mode TPSA avec les règles formées dans l'étape 1.
  - Executer l'AC pour une autre fois dans le mode complémentaire avec des règles du vecteur complémentaire F calculé dans l'étape 2.
- Etape 6 : Si tous les blocs du message sont traités alors FIN (l'AC contient maintenant le code d'authentification) sinon retour à l'étape 5.

4.5 Proposition de Cheol Jeon (CAH-256)

Cheol Jeon a proposé une fonction de hachage nommée CAH-256 (Cellular Automata Hash) [Jeo13]. Cette fonction utilise la construction de Merkle-Damegard comme algorithme d'extension du domaine, et des ACs linéaires et non linéaires pour construire la fonction de compression. Etant donné un message M, la fonction ajoute tous d'abord une suite de padding. La longueur du message après padding est une multiple de 256, et le padding est toujours appliqué même si la longueur du message original est une multiple de 256. Le dernier bloc du message obtenu contient le nombre de bits du message original.

Soit le message a haché $M = M_0 M_1 ... M_{n-1}$. La fonction CAH-256 commence par le bloc M_0 et un vecteur d'initialisation IV de 256 bits, dont tous les bits sont à 0. Et traite ensuite les blocs du message un par un jusqu'au dernier bloc. La dernière variable de chainage h_n représente le haché du message.

La fonction de compression est définit comme :

$$H_{j+1} = H_{CAH}(H_j \oplus M_j) j = 0..n$$

Le cœur de l'algorithme CAH-256 est un module de traitement qui comprend 64 tours. Tous les tours ont la même structure qui se compose : des opérations XOR (OU exclusif) avec une constante K, de deux fonctions de règles d'AC, et de 3 opérations de décalage de bits. Le module utilise une combinaison de règles linéaires et non-linéaires d'AC. Les règles linéaires fournissent la résistance aux collisions d'un état à un autre et les règles non-linéaires fournissent la résistance à l'attaque par pré-image. La fonction utilise la règle 150 qui est une règle linéaire et la règle 23 qui est une règle non-linéaire.

Le traitement dans la fonction de hachage peut être considéré comme une transformation à 4 étapes de la fonction H_{CAH}. Les calculs, dans chaque étape sont effectuées simultanément sur tous les bits de H. Soit $m_0, m_1, ..., m_{256}$ les bits de M_j, soit $h_0, h_1, ..., h_{256}$ les bits M_j, soit $k_0, k_1, ..., k_{256}$ les bits de la constante K et soit d le nombre de tours. Avant de commencer chaque tours, le calcul : $h_i = h_i \oplus m_i, 0 \leq i \leq 255$ est effectué. La Figure 4.5 illustre une seule étape de la fonction de compression.

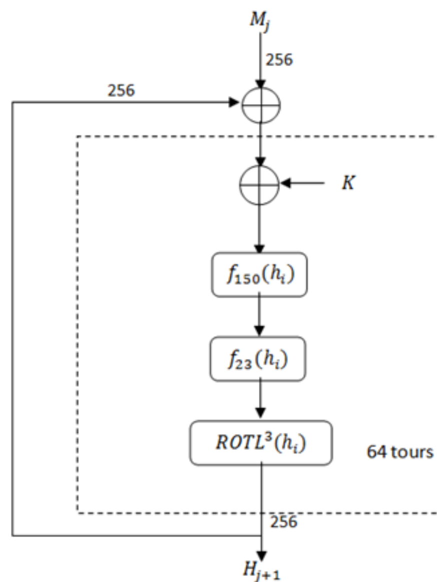

FIGURE 4.5 – La fonction de compression de CAH-256.

4.6 Proposition de Jamil et al.

Jamil et al. [NRR+12] ont présenté une nouvelle fonction de hachage basée sur des ACs avec les règles 30 et 134, et un réseau Omega-Flip (Figure 4.6).

Les blocs constructives de la fonction de compression sont une permutation et une fonction booléenne. Pour la permutation les auteurs utilisent une technique appelée "omega-flip permutation", qui a été proposée par Lee, Shi and Yang [LSY01], et

quelques règles non-linéaire d'automates cellulaires élémentaires pour effectuer la confusion et la diffusion des bits. Une description de la fonction est donnée dans l'algorithme suivant :

Algorithme de la fonction de compression

Entrées : 16 mots de 32 bits d'un bloc du message $m_i = m_1m_2...m_{16}$, et un IV constitué de 4 registres de 32 bits chacun : $IV = r_1r_2r_3r_4$

Les longueurs : bloc du message = 512 bits, mot de bloc = 32 bits, haché = 128 bits.

Pour $j = 0$ à k (k représente le nombre de tours) faire

Calculer $A = AC_1(m_i, r_{i+1}, r_{i+2}, r_{i+3})$

Calculer $AC_2(A)$

Permuter la nouvelle configuration n_i

Effectuer une addition modulo 32 de n_i et IV pour obtenir h_i

Sortie : h.

Jamil et al. ont choisi aléatoirement des règles d'AC non linéaires, ensuite ils les ont intégré avec la règle 30. Les règles qu'ils ont choisi sont les règles :126, 134 et 166. Les formes booléennes des règles sont :

La règle 30 : $p \oplus (q \vee r)$, la règle 126 : $(p \oplus q) \vee (p \oplus r)$, la règle 134 : $(p \wedge (q \wedge r)) \oplus q \oplus r$, la règle 166 : $(p \wedge q) \oplus q \oplus r$.

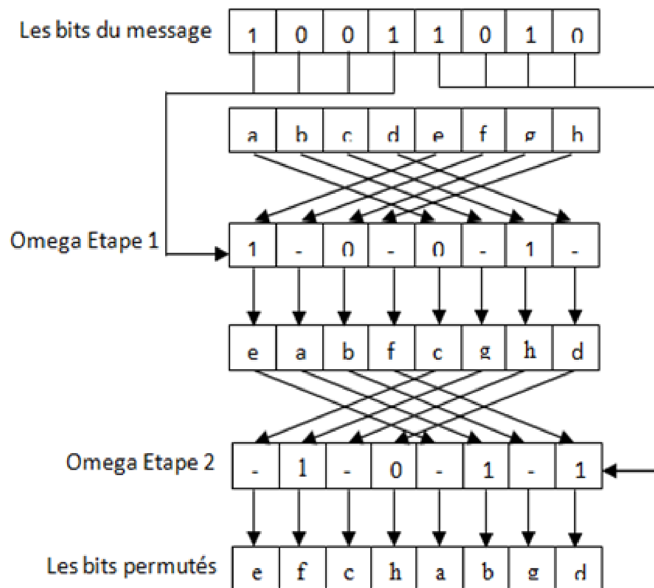

FIGURE 4.6 – Réseau Omega-Flip.

Les auteurs ensuite utilisent le test d'effet d'avalanche pour évaluer les propriétés de leur fonction de hachage.

Jamil et al. [NRR+12] ont proposé une autre fonction de hachage nommé STITCH-256, où ils ont utilisé des règles d'AC équilibrées omme les règles : 29, 39 et 27, et des fonctions de diffusion pour mettre en œuvre la fonction de compression.

4.7 Conclusion

Ce chapitre constitue une étude de l'état de l'art sur les fonctions de hachage cryptographiques basées sur les ACs. Nous avons détaillé six approches différentes de fonctions de hachage à base d'ACs : Damgard [Dam90], Daemen et al. (CellHash) [DGV91], Mihaljevic, et al. [MZI98b, MZI98a], Dasgupta et al. [DCS01], Cheol Jeon (CAH-256) [Jeo13] et Jamil et al. [NRR+12]. Une comparaison entre notre approche et quatre de ces approches est effectuée dans le dernier chapitre.

Deuxième partie

Contribution

Chapitre 5

Proposition de fonctions de hacahges cryptographiques basées sur les ACs Programmables

Ce chapitre représente notre contribution, où nous proposons deux fonctions de hachage cryptographiques à base d'automates cellulaires programmables. La première est une fonction sans clé basée sur les ACs élémentaires, la deuxième est une fonction avec clé secrète basée sur les ACs avec mémoire et les ACs à deux dimentions.

5.1 Proposition 1 : Une fonction de hachage sans clé

Dans cette section nous présentons une nouvelle fonction de hachage cryptographique sans clé basée, sur des Automates Cellulaires [BF15]. La fonction proposée est simple, rapide et efficace. Cette fonction suit un modèle de fonctions de hachage itératives similaire à celui proposée par Merkle [Mer90], et elle est constituée d'itérations de deux fonctions internes : une fonction de compression C et une fonction de transformation T. C utilise un AC programmable avec 4 règles (30, 90, 105 and 150), et T utilise un automate cellulaire hybride avec les règle 30 et 105. La taille des blocs du message est de 256 bits. Nous avons aussi choisi une taille du haché $n = 256$ bits (entre 128 et 512). Le choix du n est motivé par le raisonnement suivant : n est supérieur à 128 pour résister aux attaques génériques de collision de complexité $2^{n/2}$, et inférieur à 512 pour optimiser le temps d'exécution.

5.1.1 Schéma général

Comme algorithme d'extension de domaine (mode opératoire), nous utilisons une variante de l'algorithme de Merkle-Damgard. L'algorithme proposé nécessite une fonction de remplissage (de façon que la longueur du message soit un multiple de 256 bits) dont les 64 derniers bits encodent la longueur du message d'entrée. Après remplissage le message M est divisé en blocs $M_i (i = 1..k)$ tel que $|M_i| = 256$ bits. L'algorithme nécessite aussi un vecteur d'initialisation fixe : $IV \in \{0, 1\}^{256}$.

Ensuite la fonction de compression C est itérée k fois. La fonction C prend comme entrée un bloc du message M_i et une variable de chaînage h'_{i-1} (Avec $h'_0 = IV$), et produit en sortie une chaine de 256 bits de longueur. Cette chaine de sortie h_i est ensuite Xoré (OU exclusif) avec la sortie d'une fonction de transformation T appliquée au bloc du message M_i pour former la variable de chaînage suivante h'_i. Le schéma général de la fonction proposée est présenté dans la figure 5.1, et décrit par l'algorithme suivant :

Algorithme d'extention du domaine :

1. **Les entrées.** Un message M de taille arbitraire, et le vecteur d'initialisation IV de 256 bits.
2. **Prétraitement.** Ajouter un padding et diviser M en k blocs de 256 bits chacun : $M = (M_1, M_2, ...M_k)$; initialiser la variable de chaînage : $h'_0 = h_0 = IV$.
3. **Traitement itérative.** pour chaque $i = 1, 2, ..., k$, faire :
   Calculer la valeur de la fonction de compression $h_i = C(h'_{i-1}, M_i)$;
   Calculer la variable de chaînage suivante : $h'_i = h_i \oplus T(M_i)$.
4. **La sortie.** Haché de 256 bits : $H(M) = h'_k$.

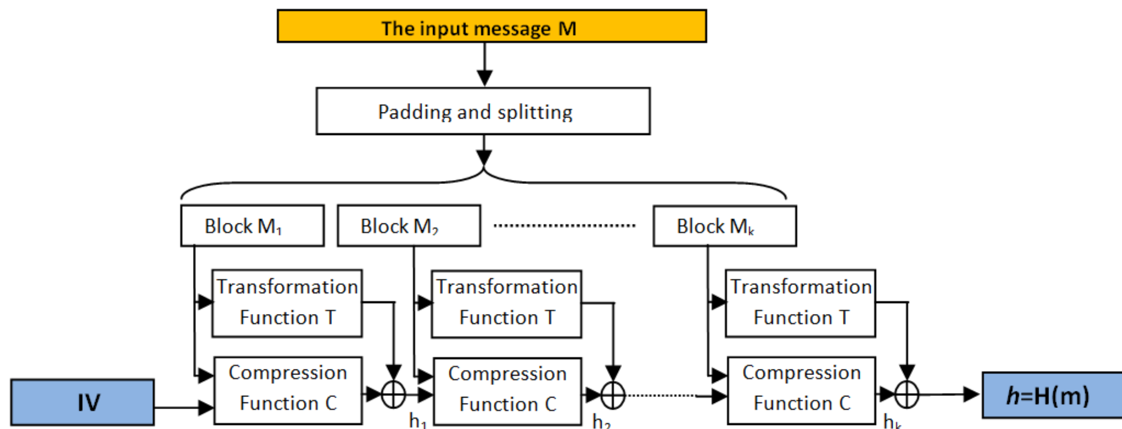

FIGURE 5.1 – Schéma général de la fonction proposée.

5.1.2 La fonction de compression

Afin d'éviter les faiblesses des constructions qui utilisent des automates cellulaires élémentaires homogènes (tel que l'approche de Damgard [Dam90]), nous utilisons un automate cellulaire programmable (un AC hybride dont les règles sont modifiées par des signiaux).

Pour construire la fonction de compression, nous utilisons un AC élémentaire programmable (ACP) avec 4 règles chaotiques : 30, 90, 105 et 150. (Comme il ya beaucoup de travail de littérature pour étudier les propriétés des différentes règles d'AC, nous n'utilisons ici, que les règles qui ont été prouvées d'être de bons générateurs de nombres aléatoires [Wol02]). La figure 5.2 montre les diagrammes espace-temps de ces 4 règles à partir d'une configuration initial aléatoire.

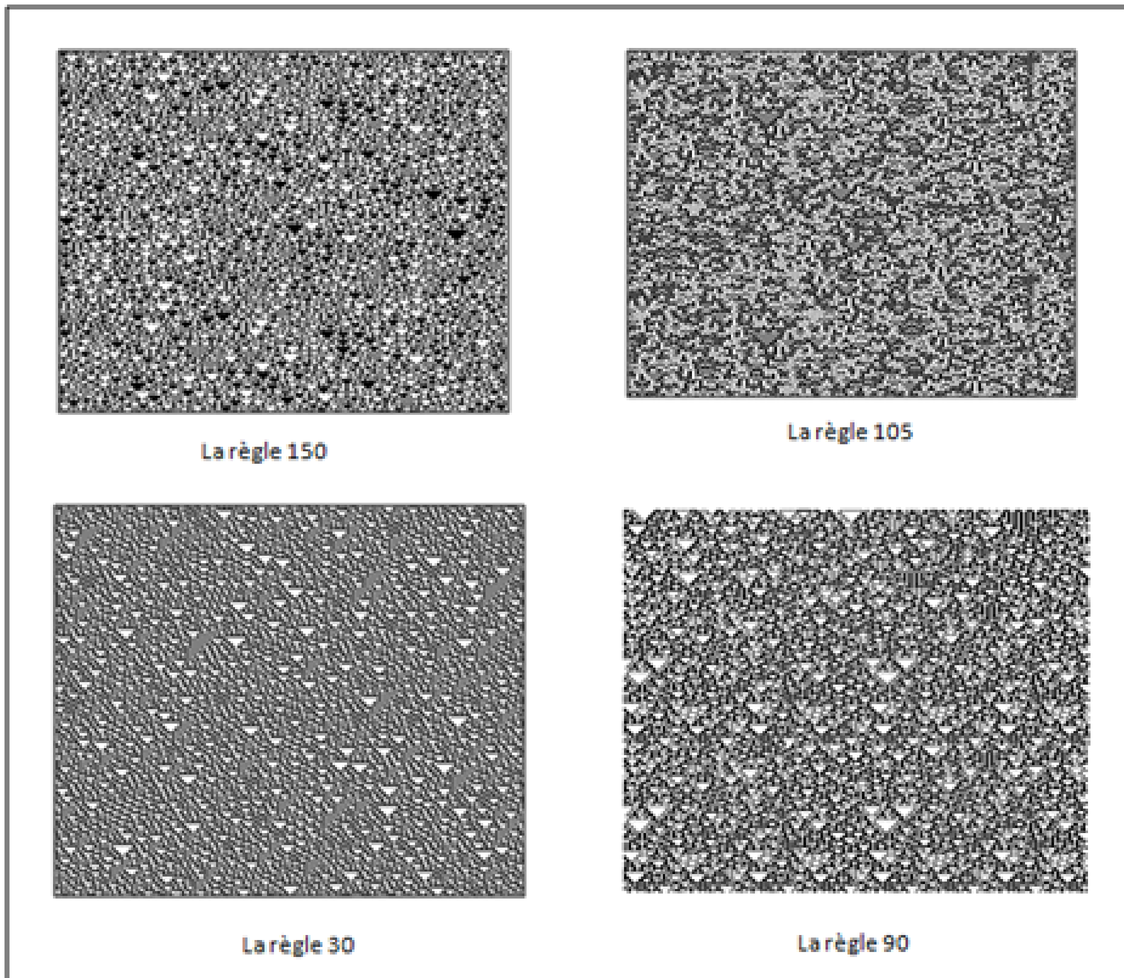

FIGURE 5.2 – Diagrammes espace-temps des 4 règles 30, 90, 105 et 150.

Pour construire C, nous définissons tout d'abord une fonction F (5.1) qui prend comme entrée un bloc du message et une variable de chainage et produit une chaine f de 256 bits en sortie.

$$\begin{aligned} F : \{0,1\}^{256} \times \{0,1\}^{256} \rightarrow \{0,1\}^{256}, \\ f_i = F(h'_{i-1}, M_i). \end{aligned} \tag{5.1}$$

La fonction F est définie comme suit : l'AC Programmable de 256 bits est initialement chargé par le bloc du message M_i. Les 256 bits de la variable de chainage h'_{i-1} sont doublés pour former une chaine binaire s de 512 bits (c-à-dire $s = h'_{i-1}|h'_{i-1}$). Les bits du s sont utilisés avec le numéro de l'itération courante j pour contrôler la configuration des règles de l'automate cellulaire. Chaque 2 bits du s contrôlent la règle d'une seule cellule. La logique du contrôle est présentée dans le tableau 5.1.

L'ACP est par la suite itéré m fois. La valeur du paramètre m peut être fixée

2 bits du bloc du message	j mod 4			
	0	1	2	3
00	Règle 30	Règle 90	Règle 105	Règle 150
01	Règle 90	Règle 105	Règle 150	Règle 105
10	Règle 105	Règle 150	Règle 30	Règle 90
11	Règle 150	Règle 30	Règle 90	Règle 30

TABLE 5.1 – Logique du control de l'ACP.

selon le rapport de performances "fiabilité/vitesse" souhaité (plus m est grand, plus la fiabilité augmente et la vitesse diminue, et inverssement). La sortie de la fonction F est définie par l'état final de l'AC Programmable.

L'AC programmable a un comportement chaotique, et il est capable de généré des suites de bits avec de bonne propriétés pseudo-aléatoire. La figure 5.3 visualise le diagramme espace-temps de l'AC programmable à partir d'une configuration initiale aléatoire.

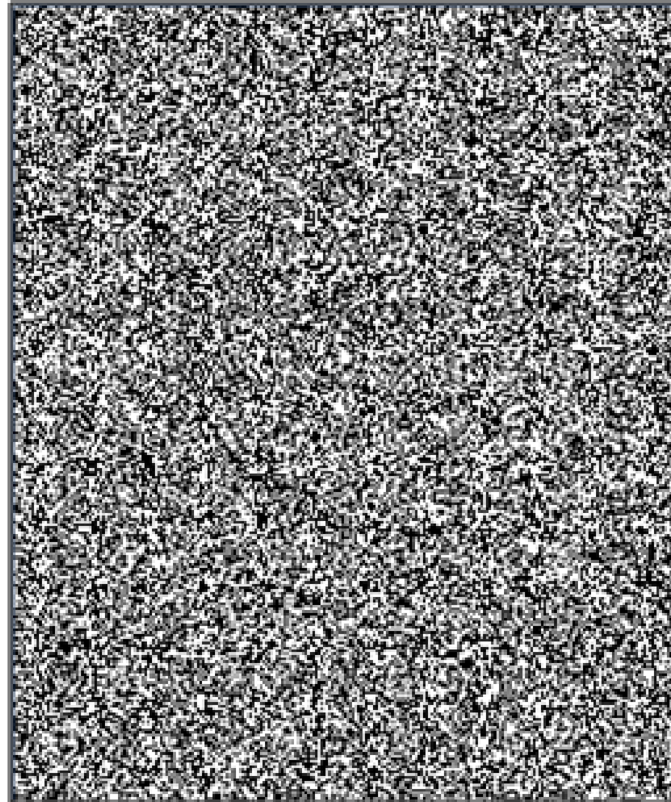

FIGURE 5.3 – Diagramme espace-temps de l'AC Programmable.

Ensuite, La fonction de compression C est définie on utilisant F, comme suit :

$$C : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0,1\}^{256}$$
$$c_i = C(f_i, h'_{i-1}) = f_i \oplus h'_{i-1}$$

Nous utilisons ensuite la fonction de transformation T pour obtenir la variable de chaînage suivante h'_i :

$$h'_i = c_i \oplus T(M_i)$$

5.1.3 La fonction de transformation

Une fonction de transformation est ajoutée à chaque itération pour résister à l'attaque par extension du message décrite dans la section 3.1 du chapitre 1. Ce qui présente une amélioration de la construction de Merkle-Damgard.

La fonction T prend comme entrée un bloc du message de 256 bits de longueur, et produit en sortie une chaine t de 256 bits qui sera Xoré avec la sortie de la fonction de compression.

$$T : \{0,1\}^{256} \to \{0,1\}^{256}$$

La fonction de transformation est définit comme suit : un AC hybride avec les règles 30 et 150 (les règles sont appliquées en alternance c-à-dire : 30, 150, 30...) est chargé initialement par les bits du bloc courant M_i. L'AC est itéré 50 fois pour obtenir une configuration intermédiaire, ensuite itéré 256 fois pour former un carré de 256 par 256 bits. La diagonal de ce carré est ensuite prise comme sortie de T (voir la Figure 5.4). C'est-à-dire que $T(M) = C_1^{256}|C_2^{255}|C_3^{254}|...|C_{255}^{2}|C_{256}^{1}$, où C_j^t représente la configuration de la cellule j à l'instant t.

Afin de résister aux attaques similaires à l'attaque de Meier et Staffelbach [MS91] présentée dans la section 1 du chapitre 4, les configurations de 256 cellules dans 256 instants du temps sont prises comme sortie de la fonction T au lieu des configurations différentes d'une seule cellule.

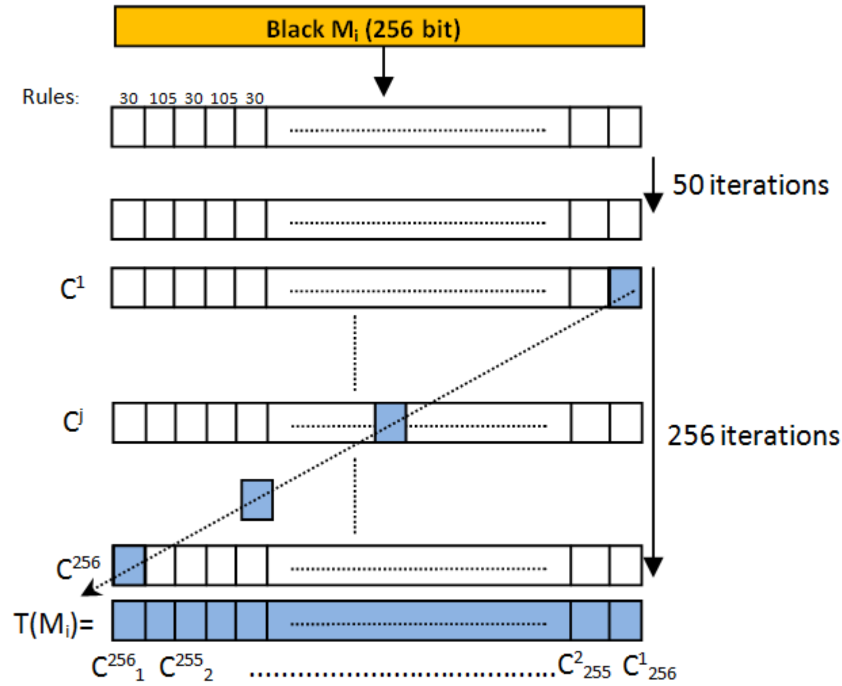

Figure 5.4 – La fonction de transformation T.

Résumé. En utilisant le schéma décrit ci-dessus, la fonction de hachage proposée peut prendre n'importe quel message binaire M de longueur arbitraire comme entrée, le décomposer en blocs consécutifs M_i, et produire le haché correspondant $H(M)$. La fonction proposée est évaluée par plusieurs tests de performances, avec plusieurs expériences illustrées dans le dernier chapitre.

5.2 Proposition 2 : Une fonction de hachage avec clé secrète

Dans cette section nous présentons une fonction de hachage cryptographique avec clé secrète. Cette fonction utilise des ACs avec mémoire de second ordre et un AC à deux dimensions. La fonction prend en entrée deux paramètres : un message et une clé secrète de 256 bits et produit une empreinte d'authentification de 256 bits.

5.2.1 Mode opératoire

La fonction de hachage proposée $H(k, M)$ prend en entrée deux paramètres : une clé secrète k de 256 bits et un message M de taille arbitraire et produit en sortie un haché h_k de 256 bits, qui dépend de M et de k (voir la figure 5.5).

Le message M en entrée est tout d'abord complété par une suite de padding de telle sorte que la taille du message devient un multiple de 256 bits. M est ensuite divisée en blocs de 256 bits ($M = M_1M_2M_3...M_n$).

Les blocs du message sont traités de façon itérative un par un jusqu'au dernier bloc de la manière présentée dans la Figure 5.5. Dans chaque itération la variable de chainage h_i est utilisée comme valeur de pré-configuration pour l'AC avec mémoire, et le bloc du message est utilisé comme valeur de configuration initiale.

Premièrement : L'AC avec mémoire est exécuté 2 fois pour obtenir une configuration intermédiaire C_j. La valeur du C_j est ensuite soumise à une fonction f pour obtenir une nouvelle configuration C_{j+1}. Les configurations C_{j-1} et C_{j+1} sont utilisées comme valeur de pré-configuration et configuration pour l'AC avec mémoire. L'AC est ensuite exécuté 2 fois. Cette opération est répétée 50 fois. Avant de traité le bloc suivant du message, le vecteur C est soumis à une fonction de transformation g. La sortie de la fonction g est utilisée comme valeur de chainage pour la prochaine itération.

La première variable de chaînage prend la valeur de la clé secrète $h_0 = k$. Au tour final, la sortie du traitement du dernier bloc (la dernière variable de chainage) est Xorée avec une transformation de la clé $(f(g(k)))$ pour produire le haché du message $(H(k, M))$. Cette dernière opération est ajoutée pour résister aux attaques d'extension du message et de multi-collisions présentées au chapitre 1.

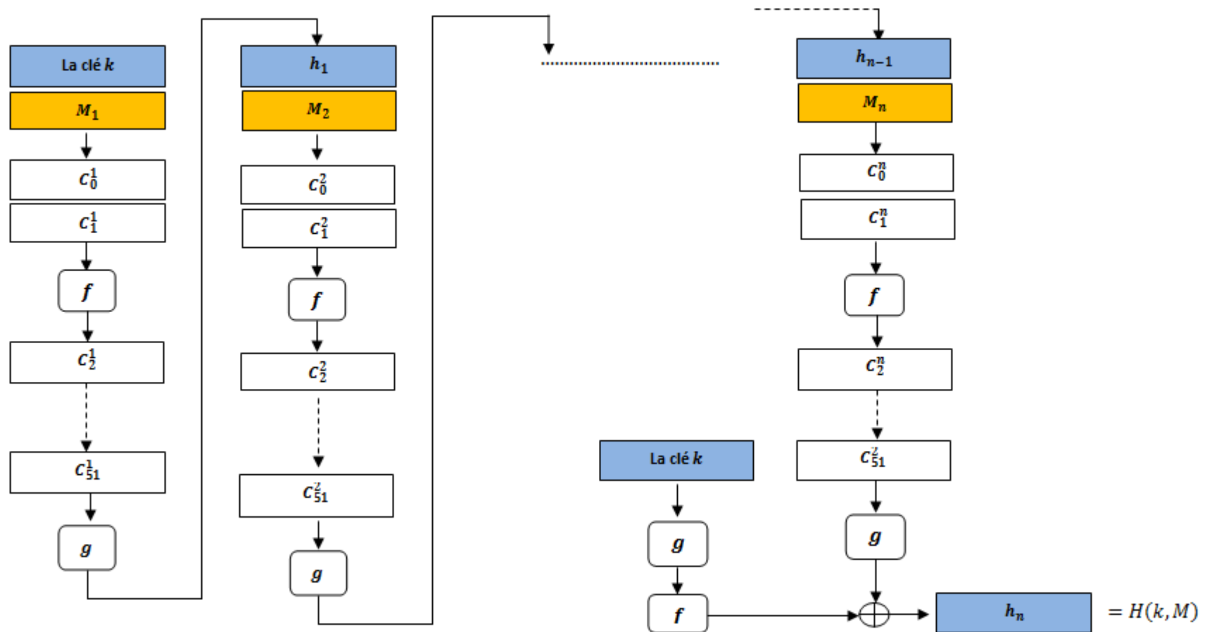

FIGURE 5.5 – Mode opératoire de la fonction proposée.

5.2.2 Automate cellulaire avec mémoire

Le premier type d'automate cellulaire utilisé pour construire la fonction proposée est un AC avec mémoire (ACM). La construction d'ACM utilisée ici est celle de la section 3.3 du chapitre 3. Le voisinage de chaque cellule est composé de 4 cellules : la cellule elle même à l'istant t et à l'instant $t-1$, et les deux cellules voisines de gauche et de droite à l'instant t (Voir la figure 5.6). Ce type de construction nécéssite une configuration initial (l'instant $t=0$) et une pré-configuration (l'instant $t=-1$). Le Tableau 5.2 montre un exemple d'une règle de l'ACM de voisinage 4 (La règle : $(1111111111111110)_2 = (65534)_{10}$).

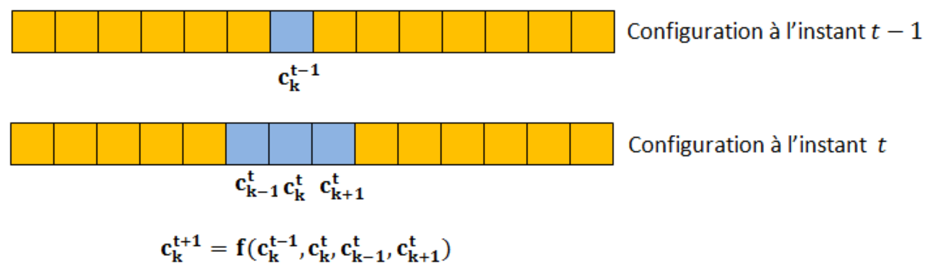

FIGURE 5.6 – Voisinage de l'AC avec mémoire.

Nous avons effectué une recherche exhaustive pour trouver les bonnes règles de cet AC avec mémoire, puisque le nombre de règles n'est pas très grand ($2^{2^4} = 65536$).

x_i^{t-1}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_{i-1}^t	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_i^t	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_{i+1}^t	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$f(x_i^{t-1}, x_{i-1}^t, x_i^t, x_{i+1}^t)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

TABLE 5.2 – La règle 65534 de l'AC avec mémoire.

3 bits de la clé	111	110	101	100	011	010	001	000
La règle correspondante	5163	891	37328	54247	32520	9450	27390	33260

TABLE 5.3 – Configuration de l'ACM programmable.

Pour tester la qualité de chaque règle R, un AC avec mémoire est programmé avec la règle R, puis chargé avec deux suites aléatoires de 1024 bits chacune (Comme configuration initiale et pré-configuration). L'ACM est ensuite exécuté 256 fois. La suite de bits engendrée par l'AC est ensuite soumise au test d'entropie [SBZ04] afin de mésurer le comportement pseudo-aléatoire de la règle.

Nous avons ensuite utilisé un sous ensemble de règles trouvées (les vrègles 5163, 891, 37328, 54247, 32520, 9450, 27390 et 33260) pour construire un AC programmable avec mémoire. Les bits de la clé sont utilisés pour programmer l'ACM (puisque il ya 8 règles, chaque 3 bits de la clé encode une seule règle) selon la configuration du tableau 5.3.

5.2.3 La fonction f

La fonction f est une fonction qui prend comme entrée une variable binaire de 256 bits et produit un vecteur binaire de 256 bits.

$$f : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$$
$$C_{i+1} = f(C_i)$$

La fonction f est définit comme suit (Voir figure 5.7) : La chaine de 256 bits est divisée en 16 mots de 16 bits chacun. Chaque mot subit un décalage circulaire vers la droite selon la règle D_1 suivante :

$$D_1 : bloc_j = bloc_j^{>>>(j^2+i) \ mod \ 15}, \quad j = 1..16, i = 1..50$$

Ensuite les blocs sont permutés selon cette règle P :

$$P : bloc_j^{'} = bloc_{j+4 \ mod \ 16}, j = 1..16$$

A la fin la chaine de 256 bits est divisée en 8 mots de 32 bits chacun. Chaque mot subit un décalage circulaire vers la droite selon la règle D_2 suivante :

$$D_2 : bloc_j = bloc_j^{>>>(2i+j+7) \ mod \ 31}, j = 1..16, i = 1..50$$

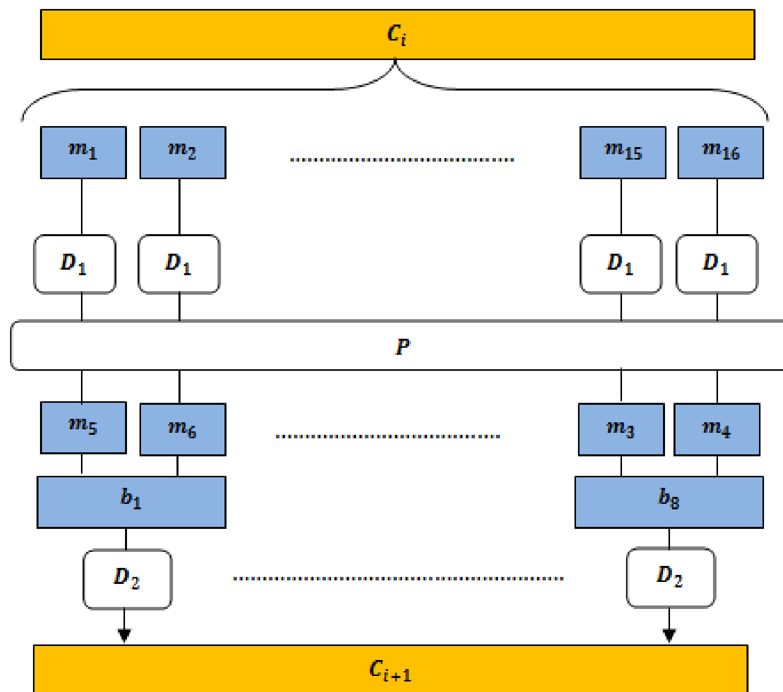

Figure 5.7 – Représentation visuelle de la fonction f.

5.2.4 La fonction g

La fonction g (Voir figure 5.8) est définie par l'exécution d'un automate cellulaire 2D, avec les règles : 11, 3, 171, 71, 12, 66 et 130 (selon la notation de la section 3.4 du chapitre 3). Ces règles sont appliquées l'une après l'autre dans cette ordre.

$$g : \{0, 1\}^{256} \rightarrow \{0, 1\}^{256}$$
$$C_{i+1} = g(C_i)$$

Les bits de la chaine de 265 bits sont réorganisés pour former une matrice carrée de 16×16 bits. Ensuite l'AC 2D est exécuté 15 fois. La matrice résultante reçoit une rotation sur son diagonal. Ensuite la matrice est divisée verticalement(puis horizontalement) en 16 mots de 16 bits chacun, et une fonction D est appliquée a chaque mot i selon la règle suivante :

$$D : m_i = m_i^{>>>(i+6) \ mod \ 16} \oplus m_{(i+3) \ mod \ 16}, i = 1..16$$

A la fin une chaine binaire est extraite à partir de la matrice pour former la sortie de la fonction g.

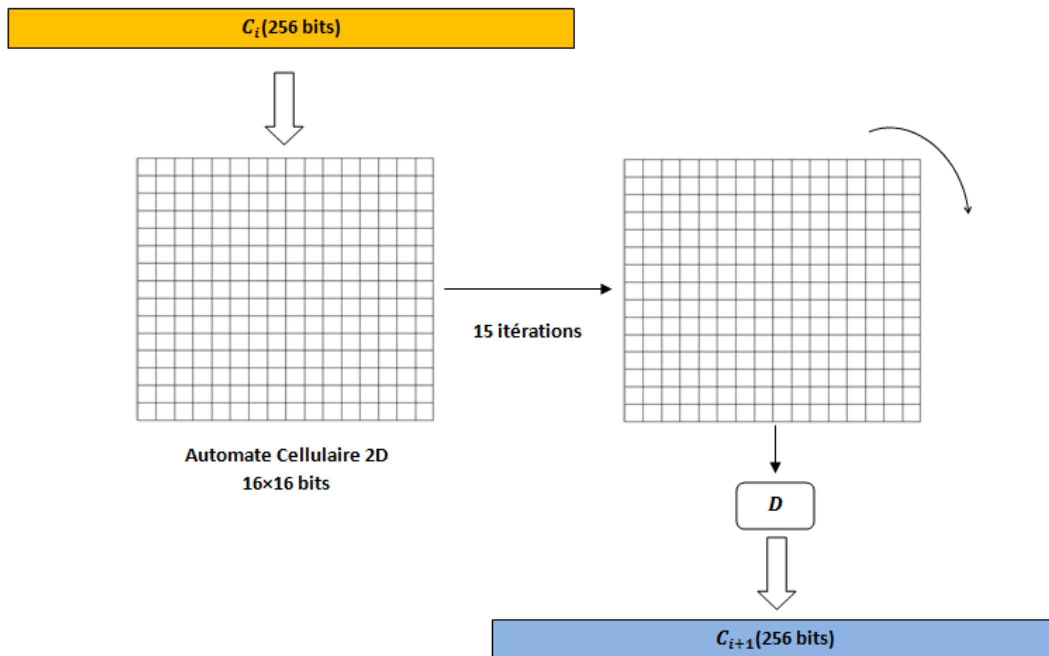

FIGURE 5.8 – Représentation visuelle de la fonction g.

Résumé. La fonction proposée ici est une fonction de hachage cryptographique avec clé secrète, qui peut être utilisée comme MAC (Message Authentication Code), pour garantir à la fois l'intégrité des messages et l'authentification de l'émetteur. Cette fonction a était soumise à plusieurs tests cryptographiques, dont la description et les résultats sont présentées dans le chapitre suivant.

5.3 Conclusion

Dans ce chapitre nous avons présenté deux approches pour la conception des fonctions de hachage à base d'automates cellulaires programmables. La prmière approche représente une fonction de hachage sans clé basée sur les ACs élémentaires. La deuxième décrit la conception d'une fonction de hachage avec clé secrète construite à partir d'ACs avec mémoire et ACs 2D. L'évaluation de la sécurité et des performances de ces deux fonctions est détaillée dans le chapitre suivant.

Chapitre 6

Tests et évaluation de sécurité

Le comportement pseudo-aléatoire et l'effet d'avalanche sont généralement considérés comme les indicateurs d'une bonne sécurité pour une fonction de hachage. Dans ce chapitre, nous effectuons une analyse de la sécurité des schémas de hachage proposés, à travers plusieurs expériences statistiques. Nous montrons également que des meilleures performances de calcul peuvent être atteintes par les fonctions proposées par rapport aux modèles de hachage existants.

6.1 La première fonction

6.1.1 Analyse de sécurité

La sécurité maximale d'une fonction de hachage est bornée par la complexité de l'attaque la plus efficace connue contre celle-ci. Une fonction de hachage idéale est limitée par les attaques génériques présentées dans le chapitre 1 (Avec une complexité de 2^n pour les attaques en pré-images et de $2^{n/2}$ pour l'attaque par collisions).

Comme algorithme d'extension du domaine nous avons utilisé une variante de la construction de Merkel-Damgard, en particulier grâce à sa preuve de sécurité très simple et utile. En effet, selon [Mer90] : avec une tel construction, si la fonction de compression interne est résistante aux collisions et aux attaques par pré-images, alors c'est le cas aussi pour la fonction de hachage elle-même .

Dans notre travail, la fonction interne est constituée de deux fonctions C et T. Selon les explications du chapitre précédant, chaque bloc du message M_i est soumis aux transformations suivantes :

- Une application non-linéaire sur M_i avec la fonction T qui utilise un ACP non-uniforme.
- Une compression non-linéaire du M_i avec h'_{i-1} pour produire un vecteur binaire de 256 bits.
- Une addition bit à bit entre les sorties de C et de T pour calculer la prochaine variable de chaînage h'_{i+1}.

En conséquence, les faits suivants impliquent la sécurité de la fonction de compression proposée :

- Les automates cellulaires ont des caractéristiques chaotiques qui transforment tout état non-nul vers un autre état différent non-nul, qui appartient à l'ensemble de toutes les combinaisons possibles de n bits ($2^n - 1$ combinaisons non-nulles), de telle manière que la distance de Hamming attendue entre l'état actuel et le prochain est de $n/2$.
- Une forte non-linéarité est appliquée sur chaque bloc M_i grâce aux caractéristiques dynamiques de l'ACP non-linéaire avec les règles 30, 90, 105 et 150.

– L'infaisabilité de la reconstruction des configurations précédentes d'une ACP non-linéaire à partir d'une configuration donnée.

La non linéarité, la non-injectivité et le sens unique des fonctions T et C, impliquent que la fonction interne constitué par leur composition est une fonction à sens unique cryptographiquement sûre. Et puisque nous utilisons un algorithme d'extension du domaine similaire à celui de Merkle-Damgard, nous pouvons affirmer que la fonction de hachage proposée est sûre.

Dans le reste de cette section nous allons effectuer des tests statistiques pour confirmer les caractéristiques cryptographiques de la fonction proposée.

6.1.2 Test d'effet d'avalanche

L'effet d'avalanche est une propriété souhaitable pour les fonctions de hachage, qui tente de refléter l'idée intuitive de la non-linéarité [CSS+05] : un changement minime dans l'entrée (inverser un seul bit) produit un changement significatif de la sortie (la moitié des bits sont inversés).

Formellement, si une fonction F possède l'effet d'avalanche, alors la distance de Hamming entre la sortie d'un vecteur d'entrée aléatoire, et la sortie d'un autre vecteur généré par l'inversement d'un bit aléatoire du premier vecteur, doit être en moyenne, la moitié de la longueur de la chaine en sortie. Mathématiquement, une fonction $F : 2^m \rightarrow 2^n$ possède l'effet d'avalanche si [For90] :

$$\forall x, y | Hamming(x, y) = 1, \quad moyenne(Hamming(F(x), F(y))) = n/2$$

La figure 6.1 présente le résultat d'un test d'effet d'avalanche effectué à la fonction proposée, avec 10000 pairs de messages arbitraires (m_i et $m_i^{'}$ $/i = 0..9999$) tel que $Hamming(m_i, m_i^{'}) = 1$. Les résultats du test montre que distances de Hamming entre les hachés (c-à-dire $Hamming(H(m_i), H(m_i^{'}))$) sont concentrés dans le voisinage de 128 bits, ce qui indique que la fonction a un bon effet d'avalanche (c'est-à-dire une très grande sensibilité aux changement d'entrée).

6.1.3 Test du critère d'avalanche strict

Le Critère d'avalanche stricte est une propriété plus exigeante que l'effet d'avalanche, qui a été initialement présenté par R. Forre dans [For90] comme une généralisation de l'effet d'avalanche proposé pour les s-boxes par Webster et Tavares [WT86]. Il a été conçu pour mesurer la quantité de non-linéarité dans les boîtes de substitution (S-boxes).

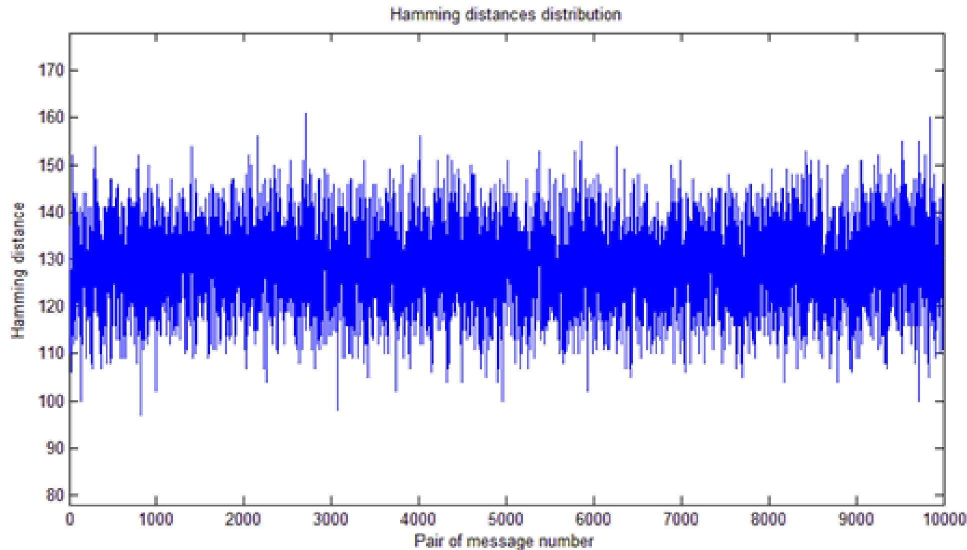

FIGURE 6.1 – Distribution des distances de Hamming.

Formellement, une fonction F satisfait le Critère d'avalanche stricte si, chaque fois qu'un seul bit d'entrée est inversé, chacun des bits de la sortie doit être changé avec une probabilité de un sur deux. Cela implique que la distribution de distances Hamming entre les sorties qui ont des entrées similaires (qui diffèrent dans un seul bit), doit suivre la distribution binomiale, ce qui pourrait être mathématiquement décrit comme [CSS+05] :

$$\forall x, y | Hamming(x, y) = 1, \quad Hamming(F(x), F(y)) \approx B(1/2, n)$$

Pour évaluer notre fonction de hachage H avec le test du critère d'avalanche strict, nous avons utilisé un vecteur v de 257 bits [0-256] (qui représente la taille de sortie de la fonction de hachage). Tous les bits de v sont initialement mises à 0. Puis 1000 messages arbitraires sont choisis et pour chaque message m le haché $h = H(m)$ est calculé, ensuite 1 bit de m est choisi au hasard pour être inversé afin d'obtenir un nouveau message $m^{'}$. Le nouveau message est aussi haché afin d'obtenir une nouvelle empreinte $h^{'}$. Ensuite, la distance de Hamming ($Hamming(h, h^{'})$) entre les hachés h et $h^{'}$ est calculée. Le résultat est utilisé pour déterminer l'élément de v a incrémenté (c.-à-d. $v[H(h, h^{'})] = v[H(h, h')] + 1$). Cette opération est répétée 1000 fois pour chaque message.

Nous avons finalement divisé chaque élément de v par 1000×1000 (ce qui représente le nombre de différents pairs (h, h')). Le vecteur v' obtenu représente la distribution des distances de Hamming. Cette distribution est ensuite comparée avec la distribution binomiale. Le résultat est présenté par le graphe de la figure 6.2.

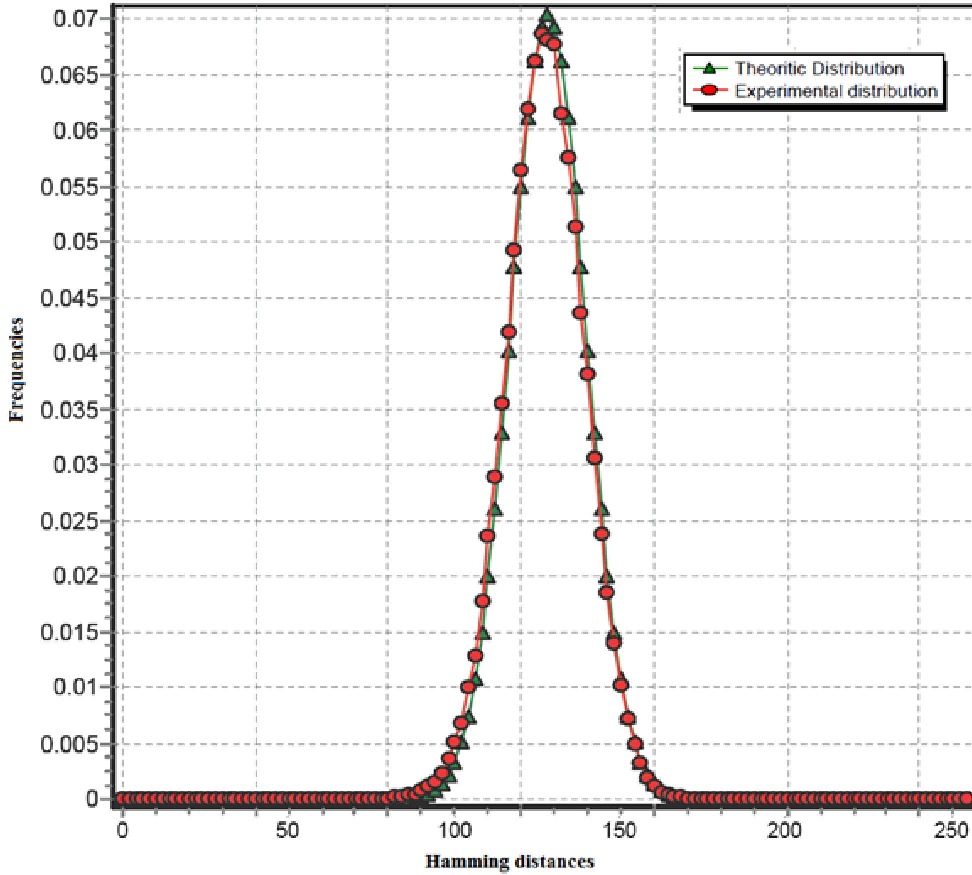

Figure 6.2 – La distribution expérimentale et la distribution théorique des distances de Hamming.

La différence entre la distribution observée et la distribution théorique des distances de Hamming, sous l'hypothèse parfaite du Critère d'Avalanche Strict ($B(1/2, 256)$), pourrait aussi être mesurée avec le test du chi-carré [CSS+05].

Le test du chi-carré (test de qualité d'ajustement) est un test statistique qui permet de vérifier la pertinence d'un ensemble de données à une distribution de probabilité [Mon09]. Dans notre cas, le chi-carré χ^2 6.1 mesure la distance entre la distribution observée des distances de Hamming et la distribution binomiale théorique de probabilité $B(1/2, 256)$.

$$\chi^2 = \sum_{i=0}^{256} \frac{(O_i - A_i)^2}{A_i} \tag{6.1}$$

$O_i : Fréquence - observée_i$

$A_i : Fréquence - attendue_i$

Après avoir effectuer le test, on a trouvé $\chi^2 = 0,005465856$ avec 256 fréquences,

ce qui signifie que la distribution observée est proche à la distribution théorique.

Les résultats des tests précédents montrent que la fonction de hachage proposée possède un bon effet d'avalanche, qui est l'une des caractéristiques les plus importantes des fonctions de hachage cryptographiques sécurisées.

6.1.4 Tests du comportement pseudo-aléatoire

Les primitives cryptographiques et spécialement les fonctions de hachage doivent se comporter comme des fonctions pseudo-aléatoires pour éviter les attaques statistiques. Donc la sortie d'une fonction de hachage sûre doit être statistiquement indiscernable de la sortie d'une fonction aléatoire. Pour cela on a effectué plusieurs tests statistiques du comportement pseudo-aléatoire sur notre fonction.

Pour tester si la fonction de hachage a de bonnes propriétés pseudo-aléatoires, nous avons utilisé la fonction propsée comme générateur de nombres pseudo-aléatoires, pour générer un fichier de 10 Mo de taille. Le fichier est crée en utilisant la méthode "Counter Mode" appliquée à la fonction de hachage : On choisit tous d'abord un nombre binaire n et on calcule ensuite les hachés des valeurs $n+1, n+2, n+3, ..., n+327680$. Les hachés sont concaténés pour former un seul fichier binaire $(H(n+1)|H(n+2)|...|H(n+327680)$.

Ensuite les batteries de tests Diehard [Mar95] et ENT [Wal08] sont appliquées au fichier obtenu. Les résultats des tests statistiques sont montrés dans les tableaux : 6.1 et 6.2.

Les valeurs de probabilité ($p - valeurs$) des tests DIEHARD montrent que la suite binaire générée par la fonction de hachage a passé avec succès tous les tests DIEHARD.

A partir des tableaux 6.1 et 6.2, nous pouvons constater que la séquence binaire générée par la fonction de hachage a passé avec succès tous les tests DIEHARD et ENT. Ce qui signifie que notre fonction possède un bon comportement pseudo-aléatoire.

Test	p-value	Résultats	Test	p-value	Résultats
Birthday Spacing	0.451552	Pass	Count the 1s in a Stream of Bytes	0.69275	Pass
Overlapping 5-permutation	0.654729	Pass	Count the 1s in Specific Bytes	0.39258	Pass
Rank test for 31x31 binary matrices	0.64841	Pass	Parking Lot Test	0.512293	Pass
Rank test for 32x32 binary matrices	0.894523	Pass	Minimum Distance Test	0.64942	Pass
Rank test for 6x8 binary matrices	0.562742	Pass	Random Spheres Test	0.35862	Pass
BITSTREAM TEST	0.329124	Pass	The Squeeze Test	0.43175	Pass
OPSO Test	0.42485	Pass	Overlapping Sums Test	0.58441	Pass
OQSO Test	0.642714	Pass	Runs Up and Down Test	0.74349	Pass
DNA Test	0.42839	Pass	The Craps Test	0.83457	Pass

Table 6.1 – Résultats des tests "DIEHARD".

Test	Value	Norm
Entropy	7.997982	8.0
Optimum compression	0.000003	0.0
Arithmetic mean value of data bytes	127.5586	127.5 = random
Monte Carlo value for	3.140865524	π
Serial correlation coefficient	0.000347	totally uncorrelated = 0.0

Table 6.2 – Résultats des tests "ENT".

6.1.5 Résistance aux collisions (Test de pseudo-collision)

Le test de pseudo-collision est conçu pour evaluer la résistance aux collisions. il a été proposé par Doganaksoy et al. dans [ADS10].

L'objet de ce test est de calculer le nombre de collisions dans des bits spécifiques du haché en sortie, ce qui peut être considéré comme pseudo-collisions. En d'autres termes, un ensemble d'entrée de taille n est évaluée à travers la fonction de hachage H, et le nombre de collisions c dans t bits de la sortie est calculé.

L'objectif est donc de calculer la fréquence de collision dans une sous-chaine du haché, et de la comparer avec une probabilité theorique d'une fonction de hachage parfaite.

La méthode pour calculer cette probabilité a était donnée par Knuth dans [Knu81]. Dans son travail, Knuth a formulé une équation pour calculer la probabilité que c collisions se produisent lorsque n entrées aléatoires distincts sont mappées dans un ensemble de sortie de taille $m = 2^t$ (avec une fonction aléatoire). La probabilité que c collisions se produisent est donnée par :

$$Pr(C = c) = \frac{m(m-1)...(m-n+c+1)}{m^n} \left\{ \begin{matrix} n \\ n-c \end{matrix} \right\} \tag{6.2}$$

Où $\left\{ \begin{matrix} n \\ n-c \end{matrix} \right\}$ est le nombre "Stirling" de second ordre [Knu81] :

$$\left\{ \begin{matrix} n \\ n-c \end{matrix} \right\} = \frac{1}{(n-c)!} \sum_{j=0}^{n-c} -1^{k-j} \left(\begin{matrix} n \\ n-c \end{matrix} \right) j^n$$

Pour effectuer le test, la sélection du paramètre n doit être faite avec soin puisque cela affecte directement le temps d'exécution du test. Si n est trop grand, il peut ne pas être possible de répéter les étapes du test suffisamment de fois pour être en mesure de détecter des éventuelles fluctuations non-aléatoires. Par conséquent, 2^{12} et 2^{14} sont choisis comme valeurs de n pour avoir des temps d'execution raisonnables [ADS10]. Le paramètre m (2^{16} et 2^{20}) est choisi en fonction du paramètre n de telle sorte que les probabilités sont suffisamment proches les unes des autres pour être utilisés dans un test statistique χ^2 (permettant de tester l'adéquation de la série de données pratiques à lois de probabilité théorique).

Ces probabilités théoriques calculées par l'équation 6.2, sont utilisées pour obtenir les résultats données dans le tableau 6.3

	$n = 2^{12}; m = 2^{16}$		$n = 2^{14}; m = 2^{20}$	
Catégorie	Rang	Probabilité	Rang	Probabilité
1	0-116	0.206246	0-117	0.190231
2	117-122	0.194005	118-124	0.215008
3	123-128	0.219834	125-130	0.211585
4	129-134	0.183968	131-137	0.202689
5	135-4096	0.195947	138-16384	0.180487

Table 6.3 – Rangs et probabilités théoriques de collisions pour 12 et 14 bits (une fonction aléatoire)[ADS10].

	$n = 2^{12}; m = 2^{16}$		$n = 2^{14}; m = 2^{20}$	
Catégorie	Rang	Fréquence de collisions (%)	Rang	Fréquence de collisions (%)
1	0-116	21.224	0-117	19.734
2	117-122	19.821	118-124	22.005
3	123-128	20.678	125-130	20.359
4	129-134	17.992	131-137	21.074
5	135-4096	20.284	138-16384	16.826

Table 6.4 – Rangs et fréquences de collisions pour 12 et 14 bits (la fonction proposée).

Le test est appliqué comme suit : d'abord, une entrée (un message) est généré aléatoirement, puis un ensemble de 2^{12} (ensuite 2^{14}) messages est formé à partir du premier message, en attribuant toutes les valeurs possibles aux 12 (respectivement 14) premiers bits de ce dernier.

Ensuite, les hachés de ces messages sont calculés, et le nombre de pseudo-collisions est obtenu à l'aide d'un tableau. Les étapes précédantes sont répétées pour 2^{16} (respectivement 2^{20}) messages aléatoires et les valeurs obtenues des catégories sont évaluées à l'aide du test χ^2 pour être comparées avec les valeurs attendues provenant de l'équation 6.2. Le tableau 6.4 présente les fréquences de pseudo-collisions du test effectué.

Le tableau 6.5 montre une comparaison en terme de pseudo-collision entre notre fonction et d'autres fonctions de hachage.

D'aprés les résultats du tableau 6.5 on peut constater que la fonction proposése présente une bonne valeur de χ^2, et par conséquence, une bonne résistance aux collision en comparaison avec d'autres fonctions de hachage.

6.1.6 Test de vitesse d'exécution

Les automates cellulaires utilisent des opérations binaires simples, pour cela, les primitives basées sur les ACs peuvent atteindre des vitesse d'exécution élevées dans

Fonction de hachage	χ^2	Valeur p-value
Fonction de [WWX11]	11.23	0.024097
Fonction de [Z.11]	3.891	0.420958
STITCH-256 [NRR+12]	9.447	0.050848
SHA-256 [oST02]	2.018	0.732448
Fonction de [AAA13]	4.051	0.399148
Fonction de [Jeo13]	5.249	0.262689
Fonction proposée	2.174	0.703792

Table 6.5 – Comparaison en terme de résistance aux collisions.

Fonction de hachage	Vitesse (MB/s)
MD5	406
SHA-1	158
SHA-256	143
SHA-512	82
RIPEMD-128	240
RIPEMD-256	195
RIPEMD-320	104
Whirlpool	80
Fonction proposée (256 bit)	138

Table 6.6 – Comparaison de vitesse contre les fonctions de hachage largement utilisées.

les implémentations matérielles ou logicielles. Notre fonction de hachage a été implémenté avec Microsoft Visual C++. Le test de vitesse a été réalisé sur une plateforme microprocesseur Intel Core i5 (2.5 GHz). Le tableau 6.6 présente une comparaison entre notre fonction et des fonctions largement utilisées de la librairie Crypto++, qui est aussi implémentés avec Microsoft Visual C++ [W13]. Pour éviter le temps accès disque les fonctions sont testées avec une courte chaine binaire itérativement.

Comme nous pouvons le constater de la comparaison du tableau 6.6, notre fonction est capable d'atteindre une vitesse d'execution très compétitive.

6.1.7 Comparaison avec d'autres fonctions de hachage basées sur les ACs

Afin d'illustrer les avantages de la fonction de hachage proposée par rapport à d'autres fonctions à base d'ACs, une étude comparative basée sur plusieurs critères a été réalisée. Ces critères sont : la sensibilité aux changements élémentaires d'entrée, critère d'avalanche stricte, le comportement pseudo aléatoire et la vitesse d'execution.

La fonction proposée a été comparé aux fonctions : CellHash [DGV91], Subhash

Fonction de hachage	**P-valeur moyenne DIEHARD**	**Entropie moyenne ENT**	**Valeur χ^2 moyenne**	**Vitesse (Mb/s)**
CellHash [DGV91]	0.54368722	7.989825	0,007782	90
Subhash [DGV92]	0.55567534	7.991735	0,006581	105
STITCH-256 [NRR+12]	0.53996212	7.998321	0,005387	113
Damgard [Dam90]	0.48216211	7.998212	0,013556	125
Fonction proposée	0.55307924	7.997982	0,005465	138

Table 6.7 – Comparaison en termes de performences de sécurité avec d'autres fonctions de hachage basées sur les ACs.

[DGV92], STITCH-256 [NRR+12] et à la fonction proposée initialement par Damgard dans[Dam90]. la comparaison des performances et de la vitesse d'execution sont fournis dans la section suivante.

Le tableau 6.7 illustre plusieurs résultats de la comparaison, en utilisant : la mesure du chi-carré (χ^2) pour évaluer le critère d'avalanche stricte, la p-valeur moyenne du test DIEHARD, l'entropie moyenne du test ENT et les vitesses d'exécution moyennes.

Les fonctions ont été implémentées avec Microsoft Visual C++, tandis que les expériences de performance ont été effectuées à l'aide d'un processeur Intel Core i5 (2,5 GHz).

A partir des résultats obtenus, il est clair que la fonction proposée peut atteindre des performances d'exécution très compétitifs, avec des résultats équivalents à d'autres fonctions en termes de d'effet d'avalanche et du comportement psœdo-aléatoire. La figure 6.3 illustre la comparaison des distributions expérimentales des distances de Hamming entre la fonction proposée et ceux à base des ACs.

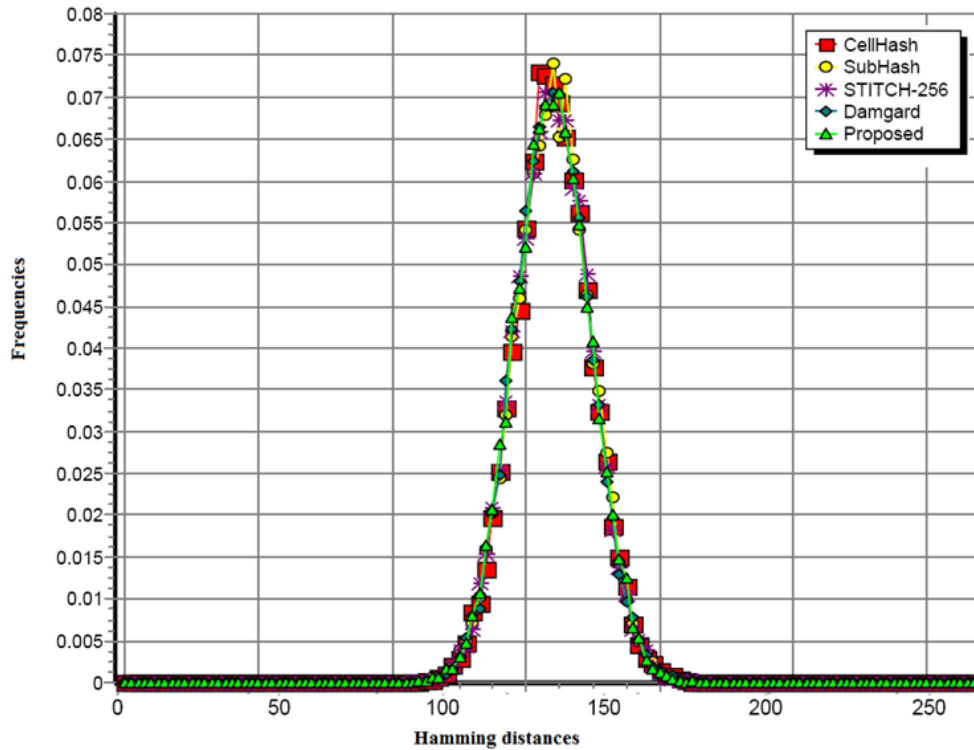

FIGURE 6.3 – Comparaison en termes de distances de Hamming expérémentales avec d'autres fonctions de hachage basées sur les ACs.

6.2 La deuxième fonction

Comme nous l'avons précédemment expliqué dans les sections 1.2.2 et 1.3.2 du chapitre 1, il n'est pas toujours possible de prouver formellement la résistance d'une fonction de hachage donnée aux attaques par collisions ou par pré-images. En revanche, plusieurs propriétés statistiques peuvent être vérifiées afin de montrer que la fonction fournit un bon niveau de sécurité cryptographique.

Dans cette section, nous effectuons quelques tests nécessaires pour quantifier et analyser la sécurité de la fonction de hachage avec clé secrète proposée. Ces tests évaluent la sécurité de la fonction à travers la sensitivité par rapport aux changements du message et de la clé secrète, la résistance aux collisions et le comportement pseudo-aléatoire.

6.2.1 Sensitivité par rapport aux changements du message

Pour tester la sensitivité de la fonction de hachage par rapport aux changements du message en entrée, nous avons effectué le test suivant : tout d'abord la valeur de la clé est fixée. Ensuite nous avons choisi un message M de 2048 bits (codé en ASCII). Nous avons effectué des modifications sur le message et/ou la clé, et nous

avons à chaque fois calculé les hachés des messages modifiés.

- Message original : "Une fonction de hachage H est un algorithme deterministe et efficace qui prend comme entree une donnee (message) binaire M de taille quelconque et produit a la sortie une empreinte digitale h de taille fixe (en general entre 128 et 512 bits). Cette emprein"
- Modification 1 : le $100^{ième}$ et le $1855^{ième}$ bits du message sont inversés.
- Modification 2 : le dixième caractère du message est remplacé par "T".
- Modification 3 : le $555^{ième}$ bit du message est inversé et le $5^{ième}$ caractère est remplacé par "R".
- Modification 4 : le $2^{ième}$ bit de la clé est inversé et le dernier caractère du message est remplacé par "A".
- Modification 5 : le $100^{ième}$ bit et le $200^{ième}$ bit de la clé sont inversé.
- Modification 6 : le premier bit du message et le premier bit de la clé sont inversés.

Les empreintes correspondantes en notation hexadécimal sont les suivantes :

- Message original : 20743E5118FBC19C41A7BC1031CBB25753AFDBB7717E6C9924500B060F7C1525
- Modification 1 : 410556656CDB89FF2B7A36043969E41B398FE48B5A0BE8BD243607BA2AE6D4A9
- Modification 2 : 28C74E5112D6CF9A7211B80345632C182A79027F4DA70C7F0EB980820BFB0E8F
- Modification 3 : 12F90EB700342998773B71F962C6A5E546A89B845DAC219D16A66218149CCE7D
- Modification 4 : 6018C9D21B31832660CFABC172BB28D01544141A79860E831B42DC9932D974FF
- Modification 5 : DAF5CE53AB5075E77C90BCF175EC44E323D55F16A6791BD82D4FC2DE63B0F53D
- Modification 6 : 7B3CE5642DAD5500743F2D036C9EB71A0E64038C5A2C83BD4CC50EBA6D4DF7A8

Les résultats des simulations indiquent que la fonction proposée est tellement sensible au changements, que tout changement minuscule du message ou de la clé va provoquer d'énormes changements dans la valeur du haché.

Shannon a montré dans son travail [Sha49] qu'il est possible de cassé de nombreux types d'algorithmes de chiffrement par analyse statistique, et donc, il propose les deux méthodes de diffusion et de confusion pour rendre les algorithmes de chiffrement résistants aux attaques statistiques.

Pour une valeur de hachage en format binaire, chaque bit est soit à 1 ou à 0. Donc l'effet de diffusion idéal devrait être que tous les changements minuscules dans les conditions initiales conduisent au changement de chaque bit du message avec une probabilité de 50%.

	$N = 250$	$N = 500$	$N = 1000$	Moyenne
$\overline{B}$	127.6853	127.8238	127.8763	127.7951
$P(\%)$	49.87	49.93	49.95	49.91
ΔB	5.212	5.8427	5.2436	5.4327
$\Delta P(\%)$	4.6542	4.4572	4.5617	4.5577
B_{min}	101	100	98	99.66
B_{max}	152	154	154	153.33

Table 6.8 – Distribution des distances de Hamming.

Pour tester la confusion et la diffusion de la fonction proposée nous avons effectué le test suivant [AAA13] :

Un message est sélectionné et la valeur de hachage du message est calculée ; puis, 1 bit du message est modifié de façon aléatoire et une nouvelle valeur de hachage est générée. Les deux valeurs de hachage sont comparés l'une avec l'autre, et les bits modifiés sont comptés, et leur nombre est appelé B_i $(i = 1..N)$. Ce test est effectué $N = 1000$ fois, et la distribution correspondante de B_i est représenté par la Figure 6.4 et le Tableau 6.8, avec :

- Le nombre minimum de bits changés $B_{min} = min(\{Bi\}, i = 1..N)$
- Le nombre maximum de bits changés $B_{max} = max(\{Bi\}, i = 1..N)$
- La moyenne de bits changés $\overline{B} = \sum_{i=1}^{N} B_i/N$
- La probabilité moyenne des bits changés $P = (\overline{B}/256) \times 100$
- La variance $\Delta B = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(B_i - \overline{B})^2}$
- Variance de $\Delta P = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(B_i/256 - P)^2 \times 100}$

Les données des tests avec N = 250 ,500 et 1000, respectivement, sont listés dans le tableau 6.8. On se basant sur l'analyse des données du tableau 6.8, nous pouvons tirer ces conclusions : le nombre moyen de bits changés B et la moyenne des probabilités P sont très proche des valeurs idéales de 128 bits et 50%, tandis que ΔB et ΔP sont très petites, ce qui indique que les capacités de diffusion et de confusion de la fonction proposée sont très fortes.

Le tableau 6.9 montre une comparaison entre notre fonction et d'autres fonctions de hachage avec clé secrète. Nous pouvons constater que notre fonction présente les résultats les plus proches des valeurs théoriques.

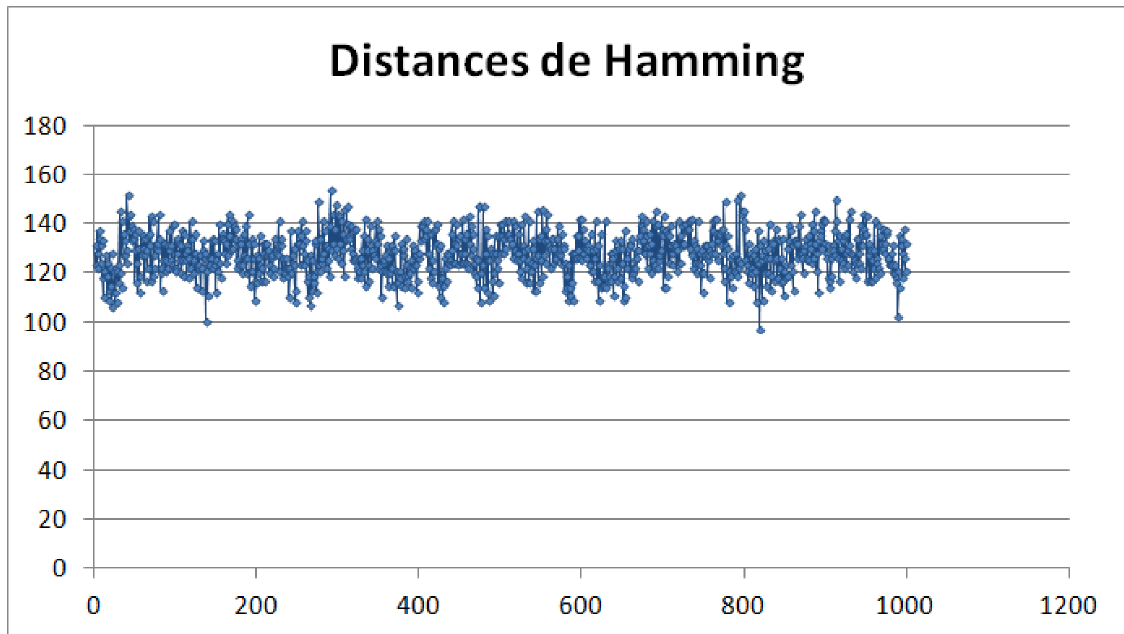

FIGURE 6.4 – Distances de Hamming.

Fonction \ Moyenne	$\overline{B}$	$P(\%)$	ΔB	$\Delta P(\%)$	B_{min}	B_{max}
Fonction de [AAA13]	127.76424	49.90793	5.8662	4.5830	90	164
Fonction de [WLXW08]	127.70	49.91	5.58	4.36	90	166
Fonction de [Z.11]	127.5876	49.685	5.5895	4.3675	92	165
Fonction de [WWX11]	128.306	50.119	5.767	4.506	88	164
Fonction proposée	127.7951	49.91	5.4327	4.5577	97	153

TABLE 6.9 – Comparaison en terme sensitivité par rapport aux changements du message.

La Figure 6.5 représente la distribution expérimentale des distances de Hamming de notre fonction. Nous pouvons voir que cette distribution est très proche de la distribution théorique d'une fonction aléatoire.

6.2.2 Distribution uniforme dans l'espace des hachés

Pour vérifier la propriété de distribution dans l'espace de hachage, le test suivant proposé par [ZWZ07] est effectué : Un message est généré aléatoirement et la valeur de son haché est calculée ; puis, 1 bit du message est modifié de façon aléatoire et une nouvelle valeur du haché est générée. Ensuite, le nombre des bits modifiés à chaque emplacement [1, 2, ..., 256] est compté. Cette procedure est répétée N fois. La figure 6.6 présente les résultats statistiques pour $N = 10000$. Le minimum, le maximum et la moyenne du nombre de bits changés sont 4905, 5127 et 5017.8,

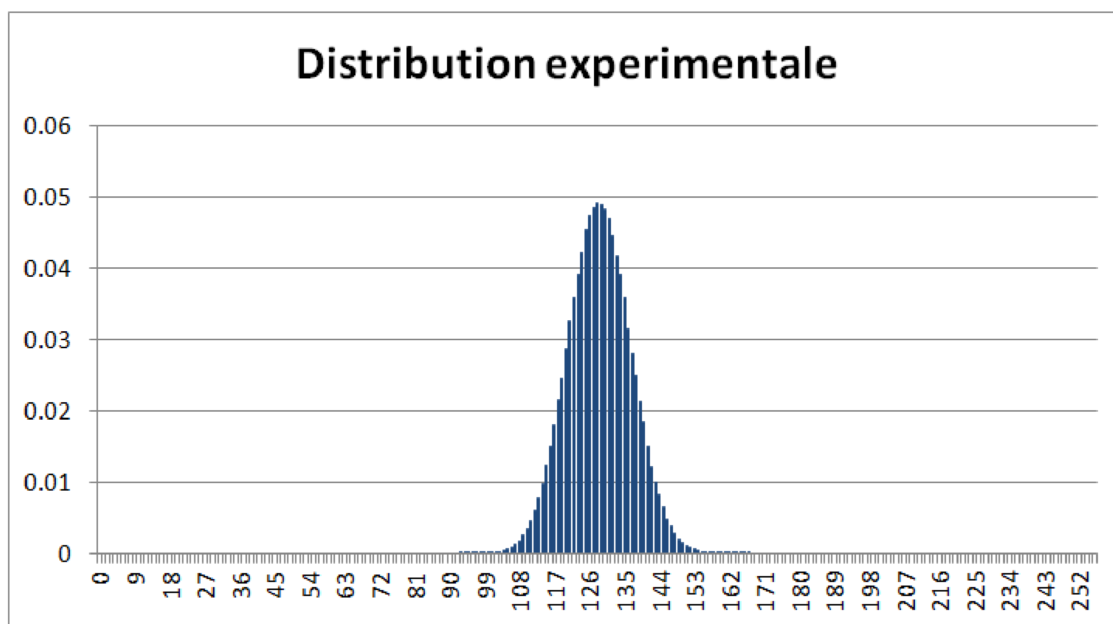

Figure 6.5 – Distribution expérimentale des distances de Hamming.

respectivement, pour $N = 10000$. On constate que la moyenne des nombres de bits inversés est 5017.8, ce qui représente une valeur très proche de la valeur idéale de 5000 (10000/2). Les nombres de bits inversés de chaque emplacement dans l'espace de hachage se concentrent aussi autour de la valeur idéale. Ce qui implique la résistance contre les attaques statistiques.

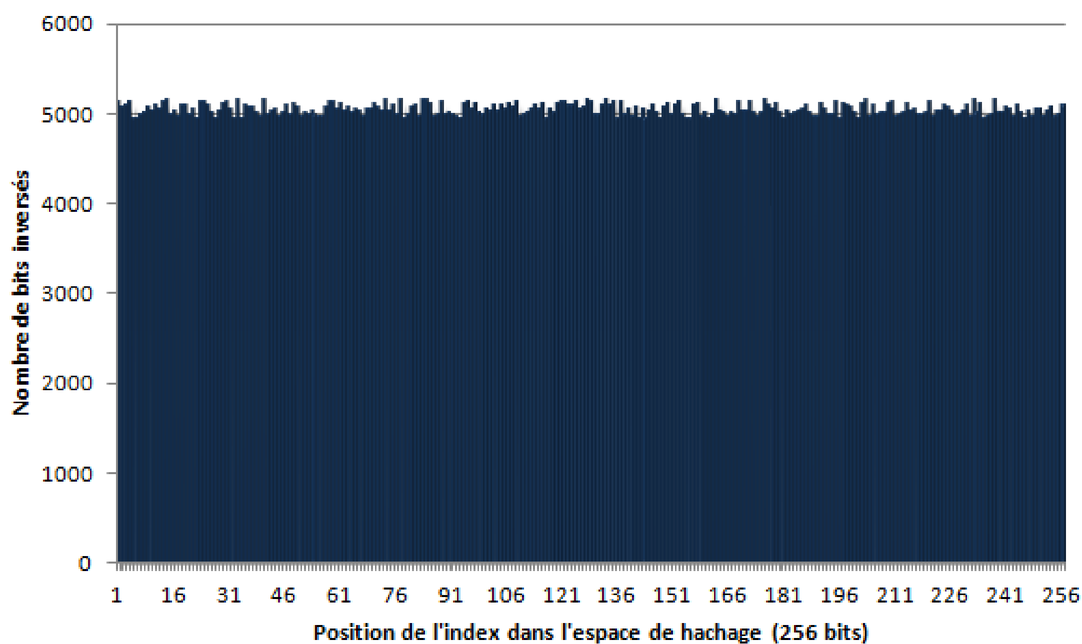

Figure 6.6 – Distribution des hachés dans l'espace de hachage.

6.2.3 Sensitivité par rapport aux changements de la clé

Pour montrer la sensitivité de la fonction proposée par rapport aux changements de la clé, on a effectué deux tests : un premier test similaire au test effectué sur les changements des messages (test d'avalanche). Dans ce test, on a compté le nombre de bits changes dans les hachés générés par le changement d'un seul bit dans la clé secrète de 256 bits. Le résultat du test est représenté par la figure 6.7.

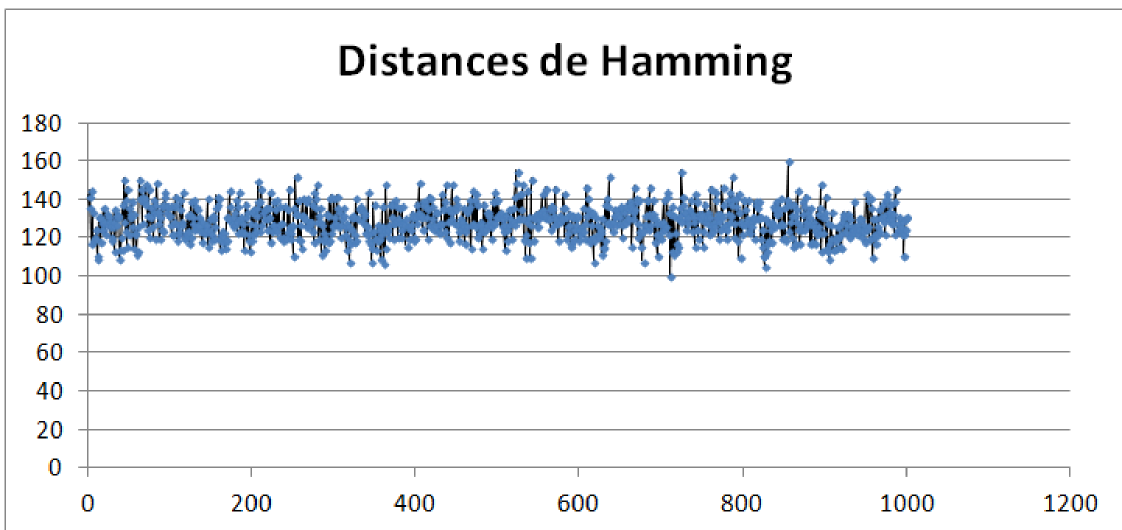

Figure 6.7 – Distances de Hamming.

Pour le deuxième test, nous avons tout d'abord fixé une clé, ensuite nous avons inversé les bits de la clé un par un. A chaque fois on calcule le haché, et on compte le nombre de bits changés par rapport au premier haché (voir la figure 6.8).

D'après les figures 6.7 et 6.8, nous pouvons constater que les distances de Hamming entre les hachés calculés avec des clés similaires sont près de 128 bits. On peut en conclure que la fonction possède une grande sensitivité aux changements de la clé.

6.2.4 Test de collision

Afin d'étudier la capacité de résistance aux collisions de la fonction proposée, nous avons effectué le test suivant [WLXW08] : d'abord, un message est choisi au hasard, et la valeur du haché est généré et stockée en format ASCII. Puis, un bit à partir d'une position aléatoire dans le message est choisie et sa valeur est inversée. Une nouvelle valeur de hachage est générée, ensuite elle est stockée au format ASCII.

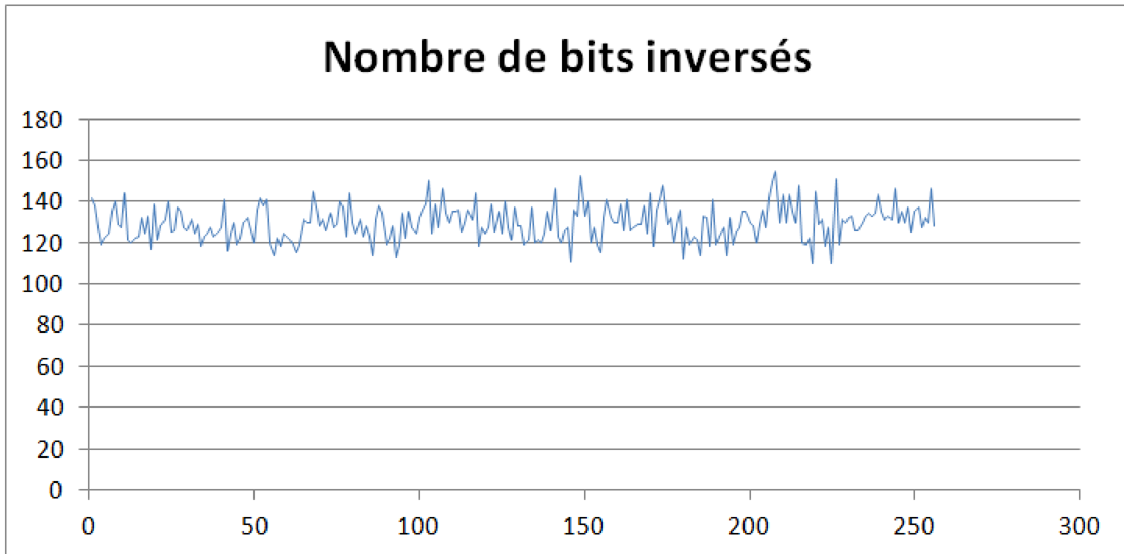

FIGURE 6.8 – Nombre de bits inversés

Les deux valeurs de hachage sont comparées, et le nombre ω de caractères ASCII avec la même valeur et dans le même emplacement dans le haché est calculé. Le nombre théorique $W_N(\omega)$ de messages similaires (dont la distance de Hamming = 1) qui produisent ω caractères ASCII avec la même valeur et dans le même emplacement, pendant N expériences indépendantes est calculé par les formules suivantes [ZWZ07] :

$$w = \sum_{i=1}^{N} f(t(e_i) - t(e_i^{'})), \quad \text{où} \quad f(x,y) = \begin{cases} 1 \ si \ x = y \ , \\ 0 \ si \ x \neq y \end{cases} \tag{6.3}$$

$$W_N(\omega) = N \times Prob(\omega) = N \frac{s!}{\omega!(s-\omega)!} \left(\frac{1}{2^k}\right)^{\omega} \left(1 - \frac{1}{2^k}\right)^{s-\omega} \tag{6.4}$$

Où e_i et e_i' sont les entrées du haché original et du nouveau haché, et la fonction $t(.)$ converti les entrées vers leurs valeurs décimales équivalentes. Ce test de collision est répété 10000 fois.

Les valeurs théoriques de l'équation 6.4 pour $N = 10000$ sont comme suit [ZWZ07] : $W_N(0) = 9600$, $W_N(1) = 600$, $W_N(2) = 2$, et $W_N(\omega) = 0$ pour $\omega = 3, 4, ..., 32$. Aussi, les valeurs expérimentales de $W_N(\omega)$ de la fonction proposée sont : $W_N(0) = 9200$, $W_N(1) = 540$, $W_N(2) = 5$, et $W_N(\omega) = 0$ pour $\omega = 3, 4, ..., 32$.

Un graphe de la distribution de nombre de caractères ASCII avec la même valeur et le même emplacement est donné dans la Figure 6.9. Les résultats montrent que la fonction proposée résiste aux collisions.

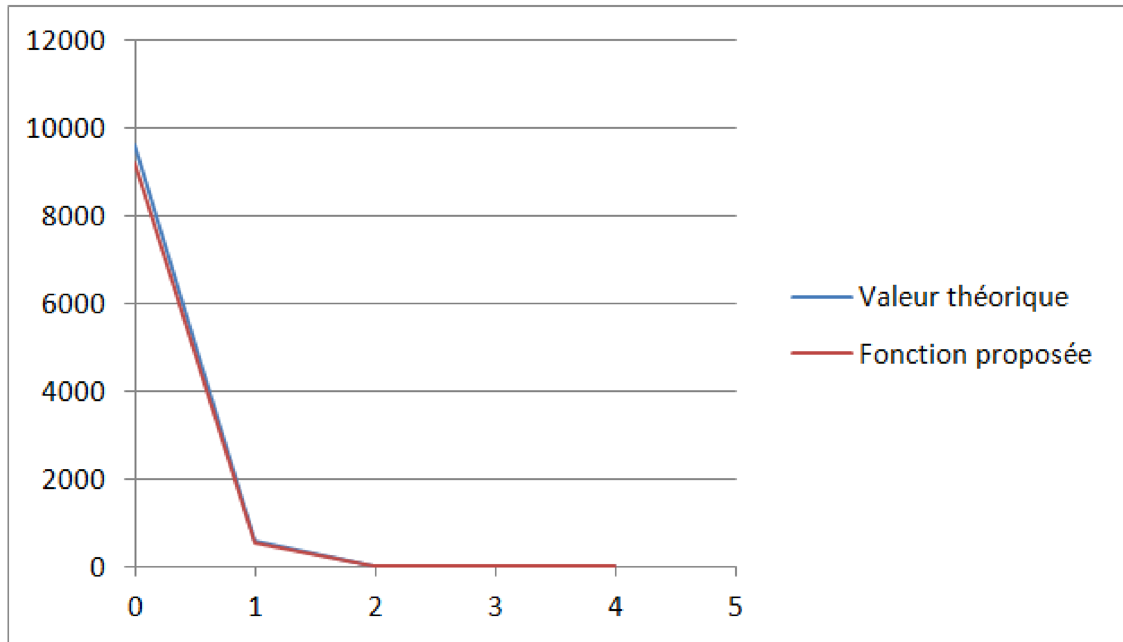

FIGURE 6.9 – Distribution de ω

6.2.5 Test du comportement pseudo-aléatoire

Pour assurer la sécurité d'un algorithme de hachage, la fonction doit avoir de bonnes propriétés statistiques telles qu'une bonne distribution, une période longue, une grande complexité ...etc. Pour tester le comportement pseudo-aléatoire de notre fonction nous avons utilisé la suite des tests de NIST (Tableau 6.11) et la suite DIEHARD (Tableau 6.10).

La suite binaire générée par la fonction de hachage a passé avec succès tous les tests de la batterie Diehard [Mar95] et 800-22 de NIST [RSN+01]. Donc, on peut en conclure que la fonction possède un bon comportement pseudo-aléatoire.

Résumé. La fonction de hachage proposée est performante en comparaison à celles avec clé secrète déjà étudiées dans la littérature. En effet, les différents résultats obtenus sur la sensibilité de l'empreinte digitale aux changements du message et de la clé secrète puis, la résistance contre les collisions, montrent l'efficacité cryptographique de la fonction proposée. Par ailleurs, l'architecture de la fonction de hachage proposée, admet un certain parallélisme en implémentations matérielle et logiciel, lui permettant d'être rapide.

Test	**p-value**	**Résultats**	**Test**	**p-value**	**Résultats**
Birthday Spacing	0.612462	Pass	Count the 1s in a Stream of Bytes	0.67485	Pass
Overlapping 5-permutation	0.647844	Pass	Count the 1s in Specific Bytes	0.335813	Pass
Rank test for 31x31 binary matrices	0.617538	Pass	Parking Lot Test	0.547596	Pass
Rank test for 32x32 binary matrices	0.842157	Pass	Minimum Distance Test	0.624972	Pass
Rank test for 6x8 binary matrices	0.497142	Pass	Random Spheres Test	0.371657	Pass
BITSTREAM TEST	0.28435	Pass	The Squeeze Test	0.48269	Pass
OPSO Test	0.459682	Pass	Overlapping Sums Test	0.518694	Pass
OQSO Test	0.724658	Pass	Runs Up and Down Test	0.762383	Pass
DNA Test	0.39241	Pass	The Craps Test	0. 71632	Pass

Table 6.10 – Résultats des tests "DIEHARD".

Test	Paramètres du test	p-value	Résultats
Frequency		0.785421	Pass
Block-frequency	(m=128)	0.928135	Pass
Runs	(M=10 000)	0.724851	Pass
Long runs of ones		0. 685421	Pass
Rank		0.347524	Pass
Spectral DFT		0.59754	Pass
No overlapping templates	(m=9, B=101001100)	0.792563	Pass
Overlapping templates	(m=9, M=1,032, N=968)	0.89547	Pass
Universal	(L=7, Q=1,280, K=141,577)	0.64585	Pass
Lempel-Ziv complexity		0.65866	Pass
Linear complexity		0.51255	Pass
Serial	P value 1	0.616847	Pass
Serial	P value 2	0.03587	Pass
Approximate entropy	(m=10)	0.36483	Pass
Cumulative sum forward		0.562781	Pass
Cumulative sum reverse		0.029832	Pass
Random excursions	x=-4	0.653749	Pass
Random excursions	x=-3	0.842672	Pass
Random excursions	x=-2	0.97452	Pass
Random excursions	x=-1	0.074286	Pass
Random excursions	x=1	0.26048	Pass
Random excursions	x=2	0.23544	Pass
Random excursions	x=3	0.134685	Pass
Random excursions	x=4	0.46249	Pass

Table 6.11 – Résultats des tests 800-22 de NIST.

6.3 Conclusion

Les résultats des tests statistiques montrent que les fonctions proposées possèdent de bonne propriétés cryptographiques telle que le comportement pseudo-aléatoire et l'effet d'avalanche. Aussi, les évaluations des performances montrent que des résultats extrêmement optimales sont atteints par les fonctions proposées , en comparaison aux normes existantes, et nous supposons que de meilleures performances peuvent être obtenus si une implémentation matérielle est adoptée en raison du parallélisme inhérent des automates cellulaires.

Conclusion

Les fonctions de hachage cryptographiques sont des primitives très polyvalentes qui permettent de réaliser de nombreuses fonctionnalités, et qui ont une multitude d'applications dans la sécurité informatique. Sans les fonctions de hachages il aurait était très difficile de réaliser des protocoles d'échanges et de transmission sécurisées qui sont la base des communications modernes à travers les réseaux. Ces fonctions ont été l'objet d'un domaine de recherche très actif dernièrement, et spécialement après le concours lancé par l'Institut américaine des normes et de la technologie (NIST) pour choisir un nouveau standard qui remplacera SHA-2. Ce concours a été lancé en réaction à la cryptanalyse des fonctions largement utilisées comme MD5 et SHA-1 dernièrement.

Les fonctions de la famille MD et SHA suivent toutes des principes de conception similaires, et la cryptanalyse d'une de ces fonctions affectera la sécurité des autres, et met en question leurs principes de désigne. Donc il est nécessaire d'envisager de nouvelles approches et stratégies de conception pour les fonctions de hachage. Les chercheurs ont proposés dernièrement une panoplie d'approches pour réaliser et implémenter des fonctions de hachages. Ces approches vont de la réutilisation des primitives cryptographiques existantes comme les algorithmes de chiffrement par blocs et par flux, en passant par les problèmes mathématiques difficiles et en terminant par les systèmes dynamiques.

Les systèmes dynamiques et particulièrement les automates cellulaires (ACs) qui sont des systèmes discrets, possèdent des propriétés qui favorisent leur utilisation comme base pour des fonctions de hachages rapides et sûres. Ils possèdent une structure parallèle intéressante pour atteindre des débits maximum. Cependant, malgré leur simplicité, prédire le comportement d'un AC en fonction de la règle utilisée est très difficile. Ce comportement chaotique et la sensibilité aux changements des conditions initiales qui sont des propriétés inhérentes aux ACs, permettent de concevoir des fonctions de compression résistantes aux attaques par collisions et par

pré-images.

Dans ce travail nous avons proposé deux fonctions de hachage cryptographiques basées sur les Automates Cellulaires. La première fonction proposée utilise deux fonctions internes : une fonction de compression basée sur un Automate Cellulaire Programmable utilisant les règles chaotiques 30, 90, 105 et 150, et contrôlé par les bits de la variable de chaînage, et une fonction de transformation construite à partir d'un automate cellulaire hybride de règles 30 et 150. La deuxième fonction proposée est une fonction avec clé secrète. Elle utilise un AC avec mémoire et un AC 2D. Les résultats des tests statistiques ont montré que les fonctions proposées possèdent de bonnes propriétés cryptographiques telles que le comportement pseudo-aléatoire, et une grande sensibilité aux changements d'entrée. En outre, elles sont simples et peuvent êtres facilement implémentée à travers le logiciel ou le matériel.

Notre proposition peut être améliorée en utilisant des ACs de voisinage supérieurs à trois. Dans ce cas la, l'ensemble des règles d'AC explose d'une façon exponentiel par rapport à la taille du voisinage. Donc pour trouver les bonnes règles nous envisageons d'utiliser une technique appelée "la programmation cellulaire", qui permet d'explorer l'espace de recherche d'une façon efficace pour trouver de bonnes règles d'automates cellulaire de voisinage supérieur à 3.

Une autre extension possible est l'utilisation d'autres types d'ACs de dimension deux (2D), qui possèdent eux aussi de bonnes propriétés favorisant leur utilisation comme base de fonctions de hachage sûres. D'autre part, les résultats obtenus dans cette thèse peuvent servir de bases pour la conception d'autres systèmes cryptographiques tels que des algorithmes de chiffrement par blocs ou par flux.

Les systèmes chaotiques sont aussi des systèmes dynamiques procédant des propriétés intéressantes pour la conception des primitives cryptographiques. Nous envisageons dans le futur, de concevoir des fonctions de hachage à base de systèmes chaotiques et d'établir une comparaison en termes de sécurité et de performances entre ces dernières et celles basées sur les ACs.

Bibliographie

[AAA13] Azman Samsudin Amir Akhavan and Afshin Akhshani. A novel parallel hash function based on 3d chaotic map. *EURASIP Journal on Advances in Signal Processing 2013*, 2013 :126, 2013.

[ADS10] Onur Kocak Ali Doganaksoy, Bars Ege and Fatih Sulak. Cryptographic randomness testing of block ciphers and hash functions. *IACR Cryptology ePrint Archive 2010*, 564, 2010.

[AFG+08] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stephane Manuel, and Nicolas Sendrier. Sha-3 proposal. *FSB*, 2008.

[AFS05] D. Augot, M. Finiasz, and N. Sendrier. A family of fast syndrome based cryptographic hash functions. In *Progress in Cryptology - MYCRYPT 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 64–83. Springer Verlag, 2005.

[BCJ+05] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of sha-0 and reduced sha-1. In *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 36–57. Springer-Verlag, 2005.

[BCK96] Mihir Bellare, Ran Canetti, , and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 1996.

[BDPA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions, the radiogatún hash function family. In *Ecrypt Hash Workshop 2007*, volume 1109, 2007.

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Nigel P. Smart,*

EUROCRYPT'08, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[BDPA09] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. *Submission to NIST (Round 2)*, 2009.

[BF15] Alaa Eddine Belfedhal and Kamel Mohamed Faraoun. Fast and efficient design of a pca-based hash function. *IJCNIS*, 7(6) :31–38, 2015.

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *Vijay Atluri, Catherine Meadows et Ari Juels, éditeurs : ACM Conference on Computer and Communications Security*, pages 203–212. ACM, 2005.

[Bou12] Christina Boura. *Analyse de fonctions de hachage cryptographiques.* PhD thesis, Université Pierre et Marie Curie - Paris VI, 2012.

[BP97] Johannes Buchmann and Sachar Paulus. A one way function based on ideal arithmetic in number fields. In *Crypto'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 385–394. Springer-Verlag, 1997.

[BPS+06] K. Bentahar, D. Page, M-J.O. Saarinen, J.H. Silverman, and N.P. Smart. Lash. In *Proceedings of Second NIST Cryptographic Hash Workshop*, 2006.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical : A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73. ACM, 1993.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with rsa and rabin. In *EUROCRYPT'96*, pages 399–416, 1996.

[BR97] Mihir Bellare and Phillip Rogaway. Collision-resistant hashing : Towards making uowhf's practical. In *Cryptgo'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 470–484. Springer-Verlag, 1997.

[BRP07] O. Billet, M.J.B. Robshaw, and T. Peyrin. On building hash functions from multivariate quadratic equations. In *Information Security and Privacy - ACISP 2007*, volume 4586 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 2007.

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer-Verlag, 2002.

[CCZ05] Zhenchuan Chai, Zhenfu Cao, and Yuan Zhou. Encryption based on reversible second-order cellular automata. In *ISPA Workshops 2005*, volume 3759 of *Lecture Notes in Computer Science*, pages 350–358. Springer-Verlag, 2005.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited : How to construct a hash function. In *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer-Verlag, 2005.

[Cha06] Donghoon Chang. Preimage attacks on cellhash, subhash and strengthened versions of cellhash and subhash. *Cryptology ePrint Archive, Report 2006*, (412) :383–404, 2006.

[CJ98] Florent Chabaud and Antoine Joux. Differential collisions in sha-0. In *CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[CJ04] Jean-Sebastien Coron and Antoine Joux. Cryptanalysis of a provably secure cryptographic hash function. *Cryptology ePrint Archive, Report 2004*, (13), 2004.

[CLG07] Denis Charles, Kristin Lauter, and Eyal Goren. Cryptographic hash functions from expander graphs. *Journal of Cryptology*, 1(22) :93–113, 2007.

[CLS06] Scott Contini, Arjen Lenstra, and Ron Steinfeld. Vsh, an efficient and provable collision- resistant hash function. In *Eurocrypt'06*, volume 4004 of *Lecture Notes in Computer Science*, pages 165–182. Springer-Verlag, 2006.

[CR06] C. De Cannière and C. Rechberger. Finding sha-1 characteristics : General results and applications. In *Advances in Cryptology - ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2006.

[CSS+05] Julio Cesar Hernandez Castroa, José Maria Sierrab, Andre Sezneca, Antonio Izquierdoa, and Arturo Ribagordaa. The strict avalanche criterion randomness test. *Mathematics and Computers in Simulation*, (68), 2005.

[Dam90] Ivan Damgard. A design principle for hash functions. In *CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1990.

[Dau05] M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family.* PhD thesis, Ruhr University Bochum, 2005.

[dBB91] Bert den Boer and Antoon Bosselaers. An attack on the last two rounds of md4. In *Joan Feigenbaum, éditeur : CRYPTO 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 1991.

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the compressin function of md5. In *T. Helleseth, editor, Advances in Cryptology - EUROCRYPT 1993*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1993.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel. Ripemd-160 : A strengthened version of ripemd. In *D. Gollmann, editor, Fast Software Encryption - FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer-Verlag, 1996.

[DC98] Joan Daemen and Craig S. K. Clapp. Fast hashing and stream encryption with panama. In *Fast Software Encryption, 5th International Workshop, FSE '98*, volume 1372 of *Lecture Notes in Computer Science*, pages 60–74. Springer-Verlag, 1998.

[DCS01] Prabir Dasgupta, Santanu Chattopadhyay, and Indranil Sengupta. Theory and application of non-group cellular automata for message authentication. *Journal of Systems Architecture : the EUROMICRO Journal archive*, 47(5) :383 – 404, 2001.

[DG01] C. Debaert and H. Gilbert. The ripemd and ripemd improved variants of md4 are not collision free. In *Fast Software Encryption - FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 2001.

[DGV91] Joan Daemen, Rene Govaerts, and Joos Vandewalle. A framework for the design of one-way hash functions including cryptanalysis of damgard's one-way function based on a cellular automaton. In *Asiacrypt'91*, volume 739 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1991.

[DGV92] Joan Daemen, Rene Govaerts, and Joos Vandewalle. A hardware design model for cryptographic algorithms. In *Second European Symposium on Research in Computer Security - ESORICS'92*, volume 648 of *Lecture Notes in Computer Science*, pages 419–434. Springer-Verlag, 1992.

[DKM06] Chang D., Gupta K.C., and Nandi M. Rc4-hash : A new hash function based on rc4. In *INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, 2006.

[DLD05] Xiao D., X. Liao, and S. Deng. One-way hash function construction based on the chaotic map with changeable-parameter. *Chaos, Solitons and Fractals*, 24(1) :65, 2005.

[DLD08] Xiao D., X. Liao, and S. Deng. One-way hash function construction based on the chaotic map with changeable-parameter. *Physics Letters A*, 372(26) :4682, 2008.

[DLW09] Xiao D., X. Liao, and Y. Wang. Parallel keyed hash function construction based on chaotic neural network. *Neurocomputing*, 72 :2288–2296, 2009.

[Dob96] Hans Dobbertin. Cryptanalysis of md4. In *FSE'96*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag, 1996.

[Dob97] Hans Dobbertin. Ripemd with two-round compress function is not collisionfree. *Journal of Cryptology*, 10(1) :51–70, 1997.

[dRMS05] A. Martin del Rey, J. Pereira Mateus, and G. Rodriguez Sanchez. A secret sharing scheme based on cellular automata. *Applied Mathematics and Computation*, 170 :1356–1364, 2005.

[DSL05] Xiao D., F.Y. Shih, and X. Liao. Hash function based on chaotic tent maps. *IEEE Transactions on Express Briefs*, 52(6) :354–357, 2005.

[DSL10] Xiao D., F.Y. Shih, and X. Liao. A chaos-based hash function with both modification detection and localization capabilities. *Communications in Nonlinear Science and Numerical Simulation*, 15(9) :2254–2261, 2010.

[FLN07] P-A. Fouque, G. Leurent, and P.Q. Nguyen. Full key-recovery attacks on hmac/nmac-md4 and nmac-md5. In *Advances in Cryptology - CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 13–30. Springer-Verlag, 2007.

[For90] R. Forre. The strict avalanche criterion : spectral properties of booleans functions and an extended definition. In *Advances in cryptology, Crypto'88*, volume 403 of *Lecture Notes in Computer Science*, pages 450–468. Springer-Verlag, 1990.

[Fuh11] Thomas Fuhr. *Conception, preuves et analyse de fonctions de hachage cryptographiques*. PhD thesis, Télécom ParisTech, 2011.

[Gua87] P. Guan. Cellular automaton public-key cryptosystem. *Complex Systems*, (1) :51–56, 1987.

[HMC89] P. D. Hortensius, R. D. McLeod, and H. C. Card. Parallel random number generation for vlsi systems using cellular automata. *IEEE Trans. on Computers*, (38) :1466–1473, 1989.

[HN02] H. Handschuh and D. Naccache. Shacal : A family of block ciphers. *Submission to the NESSIE project, 2002, http ://www.cryptonessie.org*, 2002.

[HSK03] Y-S. Her, K. Sakurai, and S-H. Kim. Attacks for finding collision in reduced versions of 3-pass and 4-pass haval. In *International Conference on Computers, Communications and Systems - ICCCS 2003*, volume 15 of *Lecture Notes in Computer Science*, pages 75–78. Springer-Verlag, 2003.

[IMPR08] S. Indesteege, F. Mendel, B. Preneel, and C. Rechberger. Collisions and other non-random properties for step-reduced sha-256. *ePrint archive, 2008. http ://eprint.iacr.org/2008/131.pdf*, 2008.

[Jeo13] Jun-Cheol Jeon. One-way hash function based on cellular automata. In *IT Convergence and Security 2012*, volume 215 of *Lecture Notes in Electrical Engineering*, pages 21–28. Springer-Verlag, 2013.

[Jou04] Antoine Joux. Multi-collisions in iterated hash functions. application to cascaded constructions. In *CRYPTO'04*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 2004.

[JP07] A. Joux and T. Peyrin. Hash functions and the (amplified) boomerang attack. In *Advances in Cryptology - CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 2007.

[Kal92] B. Kaliski. The md2 message digest algorithm. *RFC 1319, http ://www.ietf.org*, 7(3), 1992.

[Kar92] J. Kari. Cryptosystems based on reversible cellular automata. *Personal communication*, 1992.

[KCa99] A. Raouf Khan, P. P. Choudhury, and al. Text compression using two dimensional cellular automata. *Int. Journal Computers and Mathematics with applications*, 43(6) :115–127, 1999.

[KCK97] J. Klensin, R. Catoe, and P. Krumviede. Imap/pop authorize extension for simple challenge/response. *RFC 2195 (Proposed Standard)*, 1997.

[Kli06] V. Klima. Tunnels in hash functions : Md5 collisions within a minute. *ePrint archive. http ://eprint.iacr.org/2006/105.pdf*, (105), 2006.

[KM05] Lars R. Knudsen and John Erik Mathiassen. Preimage and collision attacks on md2. In *Fast Software Encryption (FSE'05)*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer-Verlag, 2005.

[Knu81] D.E. Knuth. *Seminumerical Algorithms, The Art of Computer Programming*, volume 2. 1981.

[LAB09] Zakaria LABOUDI. Evolution d'automates cellulaires par algorithmes génétiques quantiques sur un environnement parallèle, 2009.

[Lan11] Julie Langlois. Une fonction de hachage basée sur la théorie de chaos, 2011.

[Leu07] G. Leurent. Message freedom in md4 and md5 collisions : Application to apop. In *Fast Software Encryption - FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 309–328. Springer-Verlag, 2007.

[Leu08] G. Leurent. Md4 is not one-way. In *K. Nyberg, editor, Fast Software Encryption - FSE 2008*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[Leu10] Gaëtan Leurent. *Construction et Analyse de Fonctions de Hachage.* PhD thesis, Université Paris Diderot (Paris 7), 2010.

[lGa04] Álvarez G. and al. Cryptanalysis of dynamic look-up table based chaotic cryptosystems. *Physics Letters A*, 326(3-4) :211–218, 2004.

[LMS08] Patrick Lacharme, Bruno Martinn, and Patrick Solé. Pseudo-random sequences, boolean functions and cellular automata. In *Boolean Functions : Cryptography and Applications*, pages 80–95. Presses Universitaires de l'Université de Rouen, 2008.

[LSY01] R. B. Lee, Z. Shi, and X. Yang. Efficient permutation instructions for fast software cryptography. *IEEE Micro*, 21(6) :56–69, 2001.

[Luc05] Stefan Lucks. A failure-friendly design principle for hash functions. In *ASIACRYPT'05*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer-Verlag, 2005.

[Mar95] G. Marsaglia. Diehard battery of tests of randomness. *http ://www.stat.fsu.edu/pub/diehard/*, 1995.

[Mer90] Ralph C. Merkle. One way hash functions and des. In *CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1990.

[MMO85] Stephen M. Matyas, Carl H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin*, 27 :5658–5659, 1985.

[MOI90] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. 128-bit hash function (n-hash). *NTT Review*, 2 :128–132, 1990.

[Mon09] V. Monbet. Tests statistiques. *Notes de cours*, 2009.

[MOV96] Alfred Menezes, Paul Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*, chapter Hash Functions and Data Integrity, pages 321–384. CRC Press, 1996.

[MP08] Stéphane Manuel and Thomas Peyrin. Collisions on sha-0 in one hour. fast software encryption. In *15th International Workshop, FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2008.

[MS91] W. Meier and O. Staffelbach. Analysis of pseudo random sequences generated by cellular automata. In *EUROCRYPT'91*, Lecture Notes in Computer Science. Springer-Verlag, 1991.

[Mul04] Frédéric Muller. The md2 hash function is not one-way. In *ASIACRYPT'04*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer-Verlag, 2004.

[MZI98a] M. Mihaljevic, Y. Zheng, and H. Imai. A cellular automaton based fast one-way hash function suitable for hardware implementation. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC'98)*, pages 187–200, 1998.

[MZI98b] Miodrag Mihaljevic, Yuliang Zheng, and Hideki Imai. A fast cryptographic hash function based on linear cellular automata over gf(q). *Public Key Cryptography*, 1998.

[NKC94] S. Nandi, B. K. Kar, and P. P. Chaudhuri. Theory and applications of cellular automata in cryptography. *IEEE Trans. on Computers*, 43 :1346–1357, 1994.

[NRR+12] Jamil Norziana, Mahmood Ramlan, Z'aba Muhammad Reza, Udzir Nur Izura, and Zukarnaen Zuriati Ahmad. A new cryptographic hash function based on cellular automata rules 30, 134 and omega-flip network. *International Proceedings of Computer Science and Information Tech*, 27 :163, 2012.

[NSS+06] Y. Naito, Y. Sasaki, T. Shimoyama, J. Yajima, N. Kunihiro, and K. Ohta. Improved collision search for sha-0. In *Advances in Crypto-*

logy - ASIACRYPT 2006, volume 4284 of *Springer-Verlag*, pages 21–36. Lecture Notes in Computer Science, 2006.

[OPS12] Mohammad Ali Orumiehchiha, Josef Pieprzyk, and Ron Steinfeld. Cryptanalysis of rc4-based hash function. In *Proceeding AISC '12 Proceedings of the Tenth Australasian Information Security Conference*, volume 125, pages 33–38, 2012.

[oST93] National Institute of Standards and Technology. Fips 180 : Secure hash standard. *http ://csrc.nist.gov*, 1993.

[oST95] National Institute of Standards and Technology. Fips 180-1 : Secure hash standard. *http ://csrc.nist.gov*, 1995.

[oST97] National Institute of Standards and Technology. Fips publication 140-2, security requirements for cryp tographic modules. *US Gov. Printing Office*, 1997.

[oST02] National Institute of Standards and Technology. Fips 180-2 : Secure hash standard. *http ://csrc.nist.gov*, 2002.

[oST08] National Institute of Standards and Technology. Fips 180-3 : Secure hash standard. *http ://csrc.nist.gov*, 2008.

[Pat95] Jacques Patarin. Collisions and inversions for damgard's whole hash function. In *Asiacrypt'95*, volume 917 of *Lecture Notes in Computer Science*, pages 305–321. Springer-Verlag, 1995.

[Pey08] Thomas Peyrin. *Analyse de fonctions de hachage cryptographiques.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2008.

[PGV93] Bart Preneel, Rene Govaerts, and Joos Vandewalle. Hash functions based on block ciphers : A synthetic approach. In *Crypto'93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer-Verlag, 1993.

[POU06] Victor POUPET. *Automates cellulaires : temps réel et voisinages.* PhD thesis, École Normale Supérieure de Lyon, 2006.

[Pre93] Bart Preneel. Cryptographic hash functions. In *3rd Symposium on State and Progress of Research in Cryptography*, pages 161–171, 1993.

[Rab78] Michael O. Rabin. Digitalized signatures. *Foundations of Secure Computation*, pages 155–168, 1978.

[RBPV03] B. Van Rompay, A. Biryukov, B. Preneel, and J. Vandewalle. Cryptanalysis of 3-pass haval. In *C-S. Laih, editor, Advances in Cryptology - ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 228–245. Springer-Verlag, 2003.

[RC97] N. Rogier and Pascal Chauvaud. Md2 is not secure without the checksum byte. *Codes Cryptography*, 12(3) :245–251, 1997.

[RIP95] Integrity Primitives for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040) RIPE. Hash functions based on block ciphers : A synthetic approach. In *A. Bosselaers and B. Preneel, editors*, volume 1007 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Riv92a] R. Rivest. Rfc 1320 : The md4 message digest algorithm. *http : //www.ietf.org*, 1992.

[Riv92b] R. Rivest. Rfc 1321 : The md5 message digest algorithm. *http : //www.ietf.org*, 1992.

[RSN+01] A Rukhin, J Sotto, J Nechvatal, M Smid, E Barker, S Leigh, M Levenson, M Vangel, D Banks, A Heckert, J Dray, and S Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST Special Publication 800-22 NIST Gaithersburg*, 2001.

[SAN09] Renaud SANTORO. *Vers des générateurs de nombres aléatoires uniformes et gaussiens à très haut débit.* PhD thesis, Université Rennes 1, 2009.

[SBZ04] Franciszek Seredynski, Pascal Bouvry, and Albert Y. Zomaya. Cellular automata computations and secret key cryptography. *Parallel Computing*, 30 :753–766, 2004.

[Sha49] CE Shannon. Communication theory of secrecy systems. *Bell Sys. Tech. J*, 28 :656–715, 1949.

[SLdW07a] M. Stevens, A.K. Lenstra, and B. de Weger. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In *Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2007.

[SLdW07b] M. Stevens, A.K. Lenstra, and B. de Weger. Vulnerability of software integrity and code signing applications to chosen-prefix collisions for md5. *http ://www.win.tue.nl/hashclash/SoftIntCodeSign/*, 2007.

[SS01] Palash Sarkar and Paul Scellenberg. A parallel algorithm for extending cryptographic hash functions. In *Indocrypt'01*, volume 2247 of *Lecture Notes in Computer Science*, pages 40–49. Springer-Verlag, 2001.

[SWOK07] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro. ew message difference for md4. In *A. Biryukov, editor, Fast Software Encryption - FSE 2007*,

volume 4593 of *Lecture Notes in Computer Science*, pages 329–348. Springer-Verlag, 2007.

[SYA07] Y. Sasaki, G. Yamamoto, and K. Aoki. Practical password recovery on an md5 challenge and response. *ePrint archive, 2007. http ://e-print.iacr.org/2007/101.pdf*, (101), 2007.

[Ta91] Habutsu T. and al. A secret key cryptosystem by iterating a chaotic map. In *EUROCRYPT'91*, Lecture Notes in Computer Science. Springer-Verlag, 1991.

[TP00] M. Tomassini and M. Perrenoud. Stream ciphers with one- and two-dimensional cellular automata. In *M. Schoenauer at al. (Eds.) Parallel Problem Solving from Nature - PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 722–731. Springer-Verlag, 2000.

[TS00] M. Tomassini and M. Sipper. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Trans. On Computers*, 49(10) :1140–1151, 2000.

[TZ08] Jean-Pierre Tillich and Gilles Zemor. Collisions for the lps expander graph hash function. In *Eurocrypt '08*, volume 4965 of *Lecture Notes in Computer Science*, pages 254–269. Springer-Verlag, 2008.

[W13] Dai W. Crypto++. *http ://www.cryptopp.com/*, 2013.

[Wal08] J. Walker. Ent a pseudorandom number sequence test program. *http ://www.fourmilab.ch/random/*, 2008.

[WFLY04] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. *ePrint archive, http ://e-print.iacr.org/2004/199.pdf*, 2004.

[WLF+05] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions md4 and ripemd. In *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.

[WLXW08] Y Wang, X Liao, D Xiao, and K Wong. One-way hash function construction based on 2d coupled map lattices. *Inform. Sci*, 178 :1391–1406, 2008.

[WLZ12] S. Wang, D. Li, and H. Zhou. Collision analysis of a chaos-based hash function with both modification detection and localization capability. *Communications in Nonlinear Science and Numerical Simulation*, 17(2) :780–784, 2012.

[WOK08] L. Wang, K. Ohta, and N. Kunihiro. New key recovery attack on hmac / nmac-md4 and nmac-md5. In *N. Smart, editor, Advances in Cryptology - EUROCRYPT 2008*, volume 218 of *Lecture Notes in Computer Science*, pages 429–432. Springer-Verlag, 2008.

[Wol83] Stephen Wolfram. Collision analysis of a chaos-based hash function with both modification detection and localization capability. *Reviews of Modern Physics*, 55(3) :601–644, 1983.

[Wol85] Stephen Wolfram. Cryptography with cellular automata. In *Advances in Cryptology : Crypto'85*, volume 218 of *Lecture Notes in Computer Science*, pages 429–432. Springer-Verlag, 1985.

[Wol86] Stephen Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7 :123–169, 1986.

[Wol02] Stephen Wolfram. *A new kind of science.* 2002.

[Won03] Kwok-Wo Wong. A combined chaotic cryptographic and hashing scheme. *Physics Letters A*, 5-6(307) :292, 2003.

[WT86] A.F. Webster and S.E. Tavares. On the design of s-boxes. In *Advances in Cryptology : Crypto'85*, volume 218 of *Lecture Notes in Computer Science*, pages 523–534. Springer-Verlag, 1986.

[WWX11] Yong Wang, Kwok-Wo Wong, and Di Xiao. Parallel hash function construction based on coupled map lattices. *Commun Nonlinear Sci Numer Simulat*, 16 :2810–2821, 2011.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2005.

[WYY05a] X. Wang, Y.L. Yin, , and H. Yu. Finding collisions in the full sha-1. In *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer-Verlag, 2005.

[WYY05b] X. Wang, Y.L. Yin, , and H. Yu. New collision search for sha-1. In *Rump Session of Advances in Cryptology - CRYPTO 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[WYY05c] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on sha-0. In *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2005.

[WZ10] X.Y. Wang and J.F. Zhao. Cryptanalysis on a parallel keyed hash function based on chaotic neural network. *Neurocomputing*, 73(16-18) :3224–3228, 2010.

[YBC+04] H. Yoshida, A. Biryukov, C. De Cannière, J. Lano, and B. Preneel. Non-randomness of the full 4 and 5-pass haval. In *C. Blundo and S. Cimato, editors, Security and Cryptography for Networks - SCN 2004*, volume 3352 of *Lecture Notes in Computer Science*, pages 324–336. Springer-Verlag, 2004.

[YWYP06] H. Yu, X. Wang, A. Yun, and S. Park. Cryptanalysis of the full haval with 4 and 5 passes. In *M.J.B. Robshaw, editor, Fast Software Encryption - FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2006.

[Z.11] Huang Z. A more secure parallel keyed hash function based on chaotic neural network. *Communications in Nonlinear Science and Numerical Simulation*, 8(16) :3245–3256, 2011.

[ZPS92] Y. Zheng, J. Pieprzyk, and J. Seberry. Haval - a one-way hashing algorithm with variable length of output. In *Advances in Cryptology - ASIACRYPT 1992*, volume 718 of *Lecture Notes in Computer Science*, pages 83–104. Springer-Verlag, 1992.

[ZWZ07] J Zhang, X Wang, and W Zhang. Chaotic keyed hash function based on feedforward-feedback nonlinear digital filter. *Physics Letters. A*, 362(5-6) :439–448, 2007.