Université de Versailles Saint-Quentin-en-Yvelines

UFR Sciences **Laboratoire PRISM**

Analyse de fonctions de hachage cryptographiques

THÈSE

présentée et soutenue publiquement le 3 novembre 2008, à l'École normale supérieure, Paris

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines

(Spécialité informatique)

par

Thomas Peyrin

Composition du jury:

Rapporteurs :	Prof. Lars Knudsen	(Technical University of Denmark, Danemark)
	Prof. Bart Preneel	(Katholieke Universiteit Leuven, Belgique)
Directeur :	Dr. Marc Girault	(France Télécom R&D)
Examinateurs :	Prof. Jean-Sébastien Coron	(University of Luxembourg, Luxembourg)
	Prof. Pierre-Alain Fouque	(École Normale Supérieure)
	Dr. Henri Gilbert	(France Télécom R&D)
	Prof. Antoine Joux	(Univ. de Versailles Saint-Quentin-en-Yvelines)
	Dr. Guillaume Poupard	(Ministère de la Défense)
	Prof. Adi Shamir	(Weizmann Institute of Science, Israël)

Thèse effectuée au sein du laboratoire NSS, France Télécom R&D, Issy-les-Moulineaux

Remerciements

Je souhaite adresser mes premiers remerciements à Henri Gilbert, qui a accepté de m'encadrer tout au long de mon travail de recherche. Son amabilité, sa patience et sa grande disponibilité ont largement contribué à rendre ces trois années de recherche plaisantes et à améliorer ma confiance envers mes capacités, confiance ô combien nécessaire pour mener à bien des études si spécialisées. En plus de l'aspect humain, sa parfaite maîtrise de nombreux domaines de la cryptologie et ses explications claires et rigoureuses m'ont permis de mieux orienter mes approches de recherche, mais aussi de me rendre compte du long chemin qu'il me reste maintenant à parcourir. J'espère un jour être capable d'écrire une page de contenu technique sans qu'il n'y trouve mot à redire. Parallèlement, je remercie Marc Girault, qui a accepté d'être mon directeur de thèse et avec qui j'ai eu la chance d'avoir des discussions très enrichissantes malgré l'éloignement géographique.

Je tiens ensuite à remercier Lars Knudsen et Bart Preneel d'avoir accepté la lourde tâche de rapporteur, et ce, malgré les contraintes dans leur emploi du temps que ce travail implique. Je remercie également Jean-Sébastien Coron, Pierre-Alain Fouque, Antoine Joux, Guillaume Poupard et Adi Shamir de me faire l'honneur de participer au jury de cette thèse.

Mes efforts seuls n'auraient pas suffi pour accomplir tout le chemin jusqu'à la fin de cette thèse. J'ai été grandement aidé durant mon cursus par plusieurs personnes et je saisis naturellement cette occasion pour les remercier encore une fois. Merci donc tout d'abord à Gildas Avoine, Jean Monnerat et Serge Vaudenay et à toute l'équipe du laboratoire de sécurité de l'EPFL, qui ont bien voulu faire confiance à un étudiant en Chimie-Électronique pour un projet de diplôme en cryptographie. Merci aussi à Frédéric Muller et Guillaume Poupard, ainsi qu'à tout le laboratoire de cryptographie de la DCSSI, pour ces six mois de stage de Master très riches, aussi bien humainement que scientifiquement. Merci pour tout.

Ma reconnaissance va également au laboratoire de sécurité de l'AIST à Tokyo, et notamment au professeur Imai qui a accepté ma visite au Japon durant six mois de ma thèse.

Par ailleurs, les laboratoires de sécurité informatique de France Télécom R&D constituent un cadre idéal pour un jeune thésard : une excellente ambiance de travail et des chercheurs reconnus mondialement pour leurs avancées scientifiques. Merci donc à toutes les personnes que j'ai eu la chance de côtoyer dans ces locaux, et particulièrement aux membres du laboratoire de cryptographie d'Issy-les-Moulineaux : Matt Robshaw pour nos discussions sur les fonctions de hachage et pour ses talents de footballeur inépuisable, Olivier Billet pour ses blagues au degré non mesurable et pour son aide inestimable en tant qu'oracle humain de la cryptographie et de l'informatique, Côme Berbain pour sa culture sans faille de la sécurité informatique en général et pour ses biscuits faussement cachés dans son deuxième tiroir à gauche, Yannick Seurin pour être un excellent compagnon de voyage et sans qui la moitié de cette thèse présenterait des résultats erronés, Jonathan Etrog à qui je souhaite bonne chance pour la thèse à venir, Gilles Macario-Rat pour ses énigmes scientifiques en tout genre et Ryad Benadjila pour sa virtuosité à la Wii. Je remercie aussi Fabien Allard, Julien Bournelle, Jean-Michel Combes, Daniel Migault, Morgan Barbier, Jérome Cherel, Al Mahdi Chakri, Xavier Misseri, Jean-François Biasse et Tony Cheneau pour les conseils avisés et les (très) bons moments échangés. Merci également aux

managers, Sébastien Nguyen Ngoc et Thierry Baritaud, très compréhensifs en ce qui concerne les besoins si spécifiques des chercheurs.

Ma route a croisé d'autres chercheurs et je remercie donc mes coauteurs pour ces échanges si fructueux, ainsi que tous les membres du projet SAPHIR (spécialement Benoît Chevallier-Mames pour sa relecture assidue de cette thèse). J'ai aussi une pensée pour toute l'équipe de chercheurs en cryptographie de l'Université de Versailles St-Quentin-en-Yvelines, des professeurs aux thésards.

Enfin, merci à Cécile d'avoir supporté mes longues nuits de cryptanalyse, ainsi qu'à toute ma famille. Je remercie mes parents qui m'ont toujours poussé et encouragé tout au long de mes études. Je me rends compte à quel point cela fut important.

J'espère n'avoir oublié personne et je m'excuse par avance si cela est le cas.

Table des matières

Liste des tableaux

Table des figures

PREMIÈRE PARTIE

Introduction générale

La *cryptologie*, étymologiquement *la science du secret*, ne peut être vraiment considérée comme une science que depuis peu de temps. Même si l'on retrouve des utilisations de cet art jusqu'à l'Antiquité, ce n'est que récemment qu'elle est devenue un sujet de recherche académique, et qu'elle a trouvé sa motivation hors des seules applications militaires. Cette discipline se situe à présent à la frontière des mathématiques, de l'informatique et de l'électronique et l'on peut la diviser en deux sous-parties : la *cryptographie*, qui définit les mécanismes mis en oeuvre pour garantir des propriétés de sécurité, et la *cryptanalyse* qui analyse et tente de mettre en défaut ces propriétés. Le but principal de la cryptographie est de limiter l'accès à certaines informations ou certaines ressources aux seules personnes désirées. Les premières solutions apportées étaient physiques et protégeaient le support plutôt que l'information transportée (par exemple un coffre-fort ou une porte fermée à clé). Cependant, en pratique, il est difficile de garantir la sécurité d'un support. Ainsi, dans le cas d'une transmission d'informations, le besoin de protéger l'information elle-même apparut rapidement.

L'histoire de la cryptologie peut schématiquement être subdivisée en trois phases. De l'Antiquité aux grandes guerres, les méthodes utilisées étaient peu élaborées et reposaient rarement sur une théorie préalablement établie. Durant la Seconde Guerre mondiale, le milieu militaire et diplomatique se rendit compte de l'importance de la cryptologie. L'attention qui y fut portée fut grandement accrue, en particulier en ce qui concerne la cryptanalyse. De l'amélioration des techniques d'analyse découla immédiatement une nette amélioration de la qualité des nouveaux systèmes de chiffrement. Enfin, depuis l'apparition des ordinateurs et le développement de l'informatique, la cryptologie s'insère au coeur du quotidien. Aujourd'hui, des primitives cryptographiques sont présentes dans les téléphones portables, les ordinateurs, les cartes à puce, etc., pour des domaines d'application variés tels que les télécommunications, la protection des transactions bancaires, la télévision à péage, les transmissions militaires et diplomatiques, etc. Cette variété a fortement contribué à l'essor de cette science, mais en a aussi diversifié les enjeux. Au problème de confidentialité sont venus s'ajouter l'authentification, l'intégrité des données, la signature numérique, etc. Aujourd'hui, les concepteurs tentent de répondre à des problèmes aussi complexes que le vote électronique.

Historiquement, les systèmes cryptographiques ont d'abord été à clé secrète (et relevaient donc exclusivement de la *cryptographie symétrique*) : deux utilisateurs souhaitant échanger des informations confidentielles doivent au préalable posséder une clé connue d'eux seuls. Cette unique clé sera leur secret partagé et servira pour toute utilisation cryptographique, chiffrement ou authentification. À la fin des années 1970 apparut la cryptographie à clé publique (ou *cryptographie asymétrique*), qui utilise deux clés reliées entre elles pour chaque utilisateur : une clé publique (connue de tous) et une clé privée (connue seulement de l'utilisateur). Le chiffrement d'une information s'effectue avec la clé publique et le déchiffrement s'effectue avec la clé privée. Ainsi, tout un chacun peut chiffrer un message pour l'utilisateur, mais lui seul peut déchiffrer un message qui lui est destiné. Inversement, la signature d'un message se réalise avec la clé secrète et l'on vérifie cette signature à l'aide de la clé publique. Seul l'utilisateur peut calculer sa signature d'un message, mais tout un chacun peut vérifier la validité de cette signature. Dans un article fondateur, Ron Rivest, Adi Shamir, et Leonard Adleman proposèrent le système de chiffrement à clé publique RSA, qui est de loin le cryptosystème asymétrique le plus utilisé aujourd'hui.

Les performances de la cryptographie symétrique sont très bonnes, les algorithmes à clé secrète faisant partie des primitives les plus rapides. Au contraire, la cryptographie à clé publique, effectuant en général des opérations très coûteuses, n'est pas adaptée à de longs messages. De ce fait, en pratique, on réalise souvent un compromis entre ces deux familles de

technique : le problème d'échange de secret (l'établissement d'une clé symétrique) est résolu par l'utilisation d'un cryptosystème à clé publique, tandis que le message sera chiffré avec un algorithme à clé secrète.

Dans ce mémoire, nous nous intéressons aux *fonctions de hachage cryptographiques*, et plus particulièrement à leur cryptanalyse. Les fonctions de hachage cryptographiques font simplement correspondre à un message de taille arbitraire, une sortie de taille fixe, appelée *haché*. Informellement, l'idée sous-jacente est que toute modification, même infime, du message d'entrée, doit induire des modifications importantes et imprévisibles du haché en sortie. Par exemple, l'une des propriétés recherchées est qu'il doit être très difficile pour un attaquant de trouver une *collision*, c'est-à-dire deux messages distincts ayant un haché identique.

Pour des raisons historiques et bien qu'elles soient largement utilisées en dehors de ce seul domaine, ces fonctions sont considérées comme faisant partie de la cryptographie symétrique. De par leur propriété assez unique en cryptographie de ne dépendre d'aucun secret, elles diffèrent significativement des autres primitives symétriques telles que les algorithmes de chiffrement par blocs ou les algorithmes de chiffrement à flot. Grâce entre autres à leur rapidité, leur utilité est néanmoins très étendue en sécurité informatique en général : de la signature numérique à l'intégrité des données en passant par le stockage de mots de passe.

Les fonctions de hachage sont l'un des sujets les plus étudiés actuellement en cryptographie, et le *National Institute of Standards and Technology* (NIST) vient de lancer un appel à soumissions dans le but de trouver le prochain algorithme qui sera standardisé.

Dans cette première partie, nous introduisons les notations et définitions relatives aux fonctions de hachage ainsi que les principales techniques de conception de telles fonctions. Ensuite, dans la deuxième partie, nous étudions la cryptanalyse des fonctions de hachage de la famille MD-SHA, les plus utilisées en pratique. Nous rappelons en détail les précédentes avancées et nous expliquons comment nos travaux permettent d'améliorer ces attaques existantes. Enfin, dans la troisième et dernière partie, nous décrivons des attaques contre deux fonctions de hachage assez récentes, GRINDAHL et FORK-256.

Durant les trois années de recherche pour cette thèse en cryptographie, nous avons publié plusieurs articles ayant trait aux fonctions de hachage, aussi bien en ce qui concerne la partie conception [PGM06, SP07, BPR07, BCC07] que la partie cryptanalyse [MP06, MPB07, JP07a, JP07b, Pey07, MP08, GLP08, YIN08b]. Auparavant, nous nous sommes intéressés à des sujets tels que la représentation d'entiers pour des opérations algébriques sur des courbes elliptiques [AMP04], l'échange authentifié de clés pour des applications telles que le Bluetooth [PV05], ou encore la cryptanalyse d'algorithme de chiffrement à flot [MP05].

CHAPITRE 1
Les fonctions de hachage cryptographiques

Sommaire

Une fonction de hachage est une fonction prenant en entrée un élément de taille variable et renvoyant en sortie un élément de taille fixe et prédéterminée. Ces fonctions sont très utiles notamment dans le domaine des bases de données. Elles permettent de manipuler de très grandes structures tout en gardant une vitesse acceptable en ce qui concerne la recherche d'éléments. Cependant, dans ce mémoire, nous considérons uniquement une sous-classe des fonctions de hachage : les *fonctions de hachage cryptographiques*. Ces dernières ont la particularité de vérifier certaines propriétés qui rendent leur utilisation très pratique dans le domaine de la sécurité de l'information. En général, on souhaite éviter le plus possible la situation où deux entrées distinctes sont envoyées sur la même valeur de sortie par la fonction de hachage (situation appelée *collision*). La principale différence entre les fonctions de hachage classiques et les fonctions de hachage cryptographiques est que ces dernières ne doivent pas pouvoir être inversées facilement. Une fonction de hachage classique a très peu de chances de vérifier cette propriété de sécurité. Il en est de même en ce qui concerne les collisions : pour une fonction de hachage classique, on souhaite uniquement que des messages choisis au hasard n'aboutissent à une collision qu'avec une probabilité relativement faible en moyenne. Pour une fonction de hachage cryptographique, on souhaite qu'il soit difficile pour un adversaire de trouver une collision, cet adversaire pouvant choisir à loisir les messages et donc essayer de se placer dans un cas qui lui est favorable.

Nous introduisons dans ce chapitre diverses définitions et notations relatives à ces fonctions, les critères de sécurité auxquels elles doivent satisfaire, ainsi que leurs principales utilisations en pratique. Dans la suite de ce mémoire, nous utiliserons le terme *fonctions de hachage* pour désigner les fonctions de hachage cryptographiques.

1.1 Les fonctions de hachage

Une fonction de hachage H de taille de sortie n est un algorithme faisant correspondre à un message M de taille arbitraire un élément $\mathrm{H}(M)$ de taille n bits appelé *haché*. En pratique, n est

de l'ordre de plusieurs centaines de bits, typiquement 128, 160, 256 ou 512 bits.

Même si le parallélisme est en général une bonne propriété pour une primitive que l'on souhaite rapide, on ne peut pas construire une fonction de hachage de manière totalement parallèle (en traitant simultanément tous les morceaux de message). En effet, dans ce cas, la quantité de mémoire requise par l'état interne de la fonction pour le calcul du haché serait dépendante de la taille du message à hacher. Si le message est très long, une utilisation dans des environnements contraints en mémoire sera impossible.

La première grande avancée dans le domaine des fonctions de hachage fut l'algorithme de Merkle-Damgård [Mer89, Dam89], qui permet de construire des fonctions de manière itérée en traitant les morceaux de message séquentiellement. Ce procédé, que nous décrivons plus en détail dans le prochain chapitre, influença considérablement la recherche dans ce domaine, si bien qu'aujourd'hui presque toutes les fonctions de hachage reposent sur une construction en série (suivant un processus itératif).

Dans la construction de Merkle-Damgård, la fonction de hachage possède un état interne de N bits, initialisé à une certaine valeur prédéterminée (appelée *IV*). Nous définissons une fonction de hachage itérée comme étant composée de deux transformations : la compression et la fonction d'extension de domaine. Durant l'itération i, une *fonction de compression*, notée h, prend en entrée l'état interne actuel H_{i-1} de la fonction de hachage (que nous appelons *variable de chaînage*) et un nouveau bloc de message à hacher de m bits noté M_i. Cette fonction permet ainsi de hacher des messages de taille fixe et renvoie une nouvelle variable de chaînage H_i, qui représente le nouvel état interne de la fonction de hachage :

$$H_i = \mathrm{h}(H_{i-1}, M_i).$$

La fonction h est appelée fonction de compression parce qu'elle fait correspondre N bits à $N + m$ bits. La *fonction d'extension de domaine* permet de traiter des messages de taille arbitraire, et non plus fixés à m bits. Pour cela, on définit la préparation initiale du message à hacher (le *rembourrage*) et la méthode d'utilisation de la fonction de compression pour finalement traiter l'intégralité du message.

1.2 Propriétés des fonctions de hachage

Les fonctions de hachage doivent posséder plusieurs propriétés utiles en cryptographie. Les principales sont la résistance aux attaques recherchant des collisions, des préimages ou des secondes préimages.

- **collision :** trouver deux messages distincts M^1 et M^2, tels que $\mathrm{H}(M^1) = \mathrm{H}(M^2)$.
- **seconde préimage :** étant donné un message M^1 choisi aléatoirement, trouver un message distinct M^2 tel que $\mathrm{H}(M^1) = \mathrm{H}(M^2)$.
- **préimage** [‡] **:** étant donné un haché H^1 choisi aléatoirement, trouver un message M^1 tel que $\mathrm{H}(M^1) = H^1$.

Il doit être impossible pour un attaquant de trouver une collision, une préimage ou une seconde préimage. On remarque que puisque la taille d'entrée de la fonction de hachage est arbitrairement grande, des collisions existent nécessairement (si l'on choisit $2^n + 1$ messages

[‡] une autre définition possible de préimage existe : on tire aléatoirement un message et l'on donne uniquement le haché de ce message comme défi à l'attaquant. Cela évite de proposer un défi potentiellement irréalisable.

distincts, il existera obligatoirement une paire de messages aboutissant au même haché). On définit donc l'impossibilité d'un attaquant relativement à une certaine quantité d'opérations, déterminée par la meilleure attaque générique contre une fonction de hachage idéale. Ainsi, un attaquant ne doit pas être en mesure de trouver une préimage en moins de $O(2^n)$ opérations, puisque la meilleure attaque générique consiste à essayer $O(2^n)$ messages distincts pour avoir une bonne probabilité de succès d'obtenir une solution. Le raisonnement est identique pour la seconde préimage, $O(2^n)$ opérations sont nécessaires dans le cas d'une fonction de hachage idéale. En ce qui concerne la collision, l'attaque générique est moins triviale. Elle utilise le *paradoxe des anniversaires* : pour trouver deux valeurs identiques parmi a valeurs possibles, il suffit d'en tirer aléatoirement $\sqrt{a}$ pour avoir une bonne probabilité de succès. Cela s'explique par l'observation qu'en tirant $\sqrt{a}$ éléments, on forme approximativement $(\sqrt{a})^2/2 = a/2$ paires possibles, ce qui est suffisant pour obtenir une bonne probabilité de succès étant donné la taille de l'ensemble de tirage. Ainsi, nous devons utiliser $2^{n/2}$ messages pour trouver une collision pour une fonction de hachage idéale. L'attaque originale [Yuv79] requiert beaucoup de mémoire, mais des versions sans mémoire ou parallèles furent ensuite publiées [QD89, VW99], et Wagner [Wag02] généralisa le problème à plus de deux éléments.

On se rend compte facilement que la résistance aux attaques trouvant des collisions implique l'absence d'attaque de complexité strictement inférieure à $2^{n/2}$ opérations trouvant des secondes préimages (l'inverse étant faux). On ne peut néanmoins rien dire en ce qui concerne le lien entre les préimages et les collisions (des contre-exemples sont décrits dans [Handbook]).

Ces notions de sécurité peuvent aussi être appliquées aux fonctions de compression, la seule différence étant la taille fixe des messages et la possibilité pour l'attaquant de choisir ou non la variable de chaînage d'entrée. Le nombre d'opérations pour une attaque générique reste identique aux cas respectifs des fonctions de hachage.

- **collision :** étant donné un état interne H_{i-1} choisi aléatoirement, trouver deux blocs de message distincts M_i^1 et M_i^2 tels que $\mathrm{h}(H_{i-1}, M_i^1) = \mathrm{h}(H_{i-1}, M_i^2)$.
- **collision libre :** trouver deux blocs de message distincts M_i^1 et M_i^2 et un état interne H_{i-1} tels que $\mathrm{h}(H_{i-1}, M_i^1) = \mathrm{h}(H_{i-1}, M_i^2)$.
- **seconde préimage :** étant donné un état interne H_{i-1} et un bloc de message M_i^1 choisis aléatoirement, trouver un bloc de message distinct M_i^2 tel que $\mathrm{h}(H_{i-1}, M_i^1) = \mathrm{h}(H_{i-1}, M_i^2)$.
- **seconde préimage libre :** étant donné un bloc de message M_i^1 choisi aléatoirement, trouver un bloc de message distinct M_i^2 et un état interne H_{i-1} tels que $\mathrm{h}(H_{i-1}, M_i^1) = \mathrm{h}(H_{i-1}, M_i^2)$.
- **préimage :** étant donné deux états internes H_{i-1} et H_i choisis aléatoirement, trouver un bloc de message M_i^1 tel que $H_i = \mathrm{h}(H_{i-1}, M_i^1)$.
- **préimage libre :** étant donné un état interne H_i choisi aléatoirement, trouver un état interne H_{i-1} et un bloc de message M_i^1 tels que $H_i = \mathrm{h}(H_{i-1}, M_i^1)$.

La résistance aux recherches de collisions est une notion assez difficile à formaliser puisqu'en cryptographie on considère le meilleur des attaquants possibles. En effet, il existe toujours un attaquant possédant une collision pour la fonction de hachage, et renvoyant cette paire de messages. Récemment, Rogaway [Rog06] a introduit le concept de l'*ignorance humaine*, pour formaliser le fait qu'il existe effectivement toujours un tel attaquant, la difficulté étant de le trouver. Tout le problème vient du fait que, contrairement au cas de la préimage ou à celui de la seconde préimage, aucun défi n'est soumis à l'attaquant. Il doit calculer une collision

sans aucune contrainte imposée par le joueur adverse. Un contournement de ce problème fut l'introduction du concept de *famille de fonctions de hachage* : toute fonction de hachage H est paramétrée par une nouvelle entrée (par exemple l'IV) pour former une famille de fonctions. On pourra à présent défier l'attaquant en lui demandant de trouver une collision pour un membre de la famille choisi aléatoirement. Ainsi, comme pour les fonctions de compression, on peut distinguer les cas de collision et de collision libre (ou préimage et préimage libre) pour les familles de fonctions de hachage, suivant que l'adversaire peut choisir l'IV ou que celui-ci lui est imposé. Ce modèle, utile pour les preuves de sécurité, s'éloigne du cas réel puisqu'en pratique ce paramètre n'existe pas et que l'IV est fixe.

La notion de famille de fonctions de hachage apporta de nouvelles propriétés de sécurité, plus sophistiquées, mais utiles pour analyser finement un candidat. Par exemple, on peut imaginer que l'attaquant choisit des IV potentiellement différents, aussi bien pour les collisions que pour les secondes préimages. On parlera dans le premier cas de *pseudo-collision* et nous proposons, pour distinguer le second cas, l'appellation de *pseudo-seconde préimage*. Pour ces deux notions de sécurité, dont une définition est donnée ci-dessous, le nombre d'opérations requis pour une attaque générique est égal à $O(2^{n/2})$ (trouver une pseudo-seconde préimage est donc beaucoup plus facile qu'une seconde préimage). De même, les pseudo-collisions ou pseudo-secondes préimages existent aussi pour les fonctions de compression en utilisant des états internes initiaux et potentiellement différents pour chaque message. On remarque que ces notions de sécurité sont les plus fortes et les plus génériques présentées ici pour une fonction de compression, et traitent les messages et la variable de chaînage de manière identique. Ceci tend à montrer que si l'on souhaite construire une fonction de compression idéale, le message et la variable de chaînage doivent y jouer un rôle identique. Nous verrons que cela n'est que très rarement le cas en pratique.

- **pseudo-collision :** trouver deux couples distincts bloc de message/état interne (M_i^1,H_{i-1}^1) et (M_i^2,H_{i-1}^2) tels que $\mathrm{h}(H_{i-1}^1, M_i^1) = \mathrm{h}(H_{i-1}^2, M_i^2)$.
- **pseudo-seconde préimage :** étant donné un bloc de message M_i^1 choisi aléatoirement, trouver un état interne H_{i-1}^1 et un couple bloc de message/état interne $(M_i^2,H_{i-1}^2) \neq (M_i^1,H_{i-1}^1)$ tels que $\mathrm{h}(H_{i-1}^1, M_i^1) = \mathrm{h}(H_{i-1}^2, M_i^2)$.

Par ailleurs, on peut généraliser le principe de collision en stipulant que l'attaquant ne doit pas être capable d'obtenir une certaine différence entre deux hachés, quelle que soit cette différence. La collision devient alors un cas particulier où la différence est nulle. Nous appelons ce type d'attaque une *presque collision* et l'on note $\mathrm{h}(H_{i-1}, M_i^1) \simeq \mathrm{h}(H_{i-1}, M_i^2)$ lorsque les deux valeurs de haché sont très similaires. On applique identiquement ce principe aux préimages et secondes préimages et le nombre d'opérations pour une attaque générique ne change pas (sauf par exemple dans le cas où la différence en sortie ne serait pas imposée sur tous les bits). De même, on peut parler de presque collisions ou de presque préimages pour les fonctions de compression si l'on prévoit une certaine différence sur les états internes de sortie.

Rogaway et Shrimpton [RS04] ont continué d'élargir le panel de propriétés de sécurité en étudiant des scénarios où l'attaquant possède le contrôle sur le paramètre d'entrée, mais pas sur le défi (a-préimage et a-seconde préimage) ; ou sur le défi, mais pas sur le paramètre d'entrée (e-préimage et e-seconde préimage). Ils ont de plus étudié les liens entre ces différentes notions.

On pourrait continuer d'étendre ces notions, par exemple, en distinguant suivant l'ordonnancement du processus de défi : dans le cas de l'a-préimage, la situation où l'attaquant choisit le paramètre d'entrée avant de recevoir le défi est différente de celle où il choisit le paramètre d'entrée après avoir reçu le défi (ce qui correspond à une préimage libre). Enfin, le mélange des

différentes notions entre elles nous en fournit encore de nouvelles.

Plus la quantité de propriétés vérifiées est élevée et plus l'on se rapproche d'une primitive idéale. Les opinions sont partagées parmi les cryptologues en ce qui concerne la simulation d'un *oracle aléatoire* [BR93] par une fonction de hachage. Un oracle aléatoire est une primitive idéale, prenant en entrée une valeur de taille arbitraire et renvoyant une valeur de taille fixe en sortie. Ce modèle, très utilisé pour les preuves de sécurité en cryptographie, suppose que l'on dispose d'une fonction simulant cet oracle et ne pouvant pas être *distinguée* de ce composant idéal. Certains chercheurs avancent que puisqu'en pratique les oracles aléatoires (indispensables pour l'obtention de preuves de sécurité) sont remplacés par des fonctions de hachage, ces dernières ne doivent comporter aucune propriété spéciale qui pourrait les en distinguer. Récemment, cette notion fut précisée en considérant l'*indifférentiabilité* d'une fonction de hachage d'un oracle aléatoire [MRH04]. D'autres chercheurs préconisent de se concentrer uniquement sur les propriétés fondamentales des fonctions de hachage, déjà complexes à assurer, plutôt que d'essayer de concevoir une fonction idéale à tous points de vue.

1.3 Utilisations pratiques

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. Chacun peut vérifier la validité de cette signature grâce à la clé publique correspondante. Les opérations internes de ces primitives cryptographiques sont en général très coûteuses, car elles appartiennent à la cryptographie asymétrique. Ainsi, leur application à un très long message demande un temps de calcul trop grand dans certains cas pratiques. Les fonctions de hachage sont donc utilisées pour raccourcir le message à signer et améliorer les performances : on signe le haché du message plutôt que l'intégralité du message. Dans cette situation, on souhaite 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. 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.

Une autre application importante des fonctions de hachage concerne les *codes d'authentification de message* (ou MAC), qui appartiennent à la cryptographie symétrique et manipulent donc une clé secrète. Ces primitives servent à vérifier l'intégrité d'un message et à authentifier son expéditeur. Tout comme pour les signatures, il doit être impossible pour l'attaquant de retrouver la moindre information sur la clé secrète. Aussi, un attaquant doit être incapable de créer un MAC valide pour un nouveau message. Plusieurs techniques de construction existent, mais l'une des plus connues, HMAC, est fondée sur une fonction de hachage. Dans leur article original, Bellare *et al.* [BCK96] démontrèrent la sécurité d'un tel schéma sous certaines conditions concernant la fonction de hachage interne. Cette preuve fut améliorée quelques années plus

[‡] l'adversaire considéré ici est le plus fort, mais l'on peut aussi admettre qu'il ne puisse pas choisir les messages de manière adaptative, ou même qu'il ne puisse pas les choisir du tout.

tard [Bel06] en s'affranchissant de l'une des conditions, qui était peu intuitive. Actuellement, de nombreux travaux [CY06, FLN07, KBP06, WOK08] tentent de transformer les vulnérabilités de certaines fonctions de hachage en attaques retrouvant la clé secrète ou calculant des signatures valides pour l'algorithme HMAC.

Les fonctions de hachage sont utiles pour de nombreuses autres applications, telles que la protection de mots de passe : dans un serveur, au lieu de stocker tous les mots de passe d'utilisateurs, il est préférable de stocker les hachés de ces derniers. L'authentification peut toujours avoir lieu, mais, si le serveur est compromis, l'attaquant n'a accès qu'aux hachés des mots de passe. Il ne peut donc théoriquement pas retrouver les mots de passe originaux à cause de la propriété de résistance à la recherche de préimages. 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. Un autre exemple d'application pratique est la dérivation de clés cryptographiques ou encore la vérification d'intégrité de fichiers publics.

Les fonctions de hachage ont donc plusieurs rôles en cryptographie et servent souvent de « couteau suisse » grâce aux nombreuses propriétés qu'elles vérifient. Chaque utilisation ne nécessite qu'un sous-ensemble de ces propriétés, ce qui signifie qu'en général la fonction de hachage satisfait plus de propriétés de sécurité que ne l'exige son rôle réel. Néanmoins, il est plus facile pour les chercheurs de se focaliser sur un seul candidat à analyser plutôt que de créer une fonction spécifique pour chaque type d'utilisation. Cela simplifie fortement le processus de standardisation et, de plus, l'état actuel des connaissances ne permet pas de réellement saisir toutes les imbrications entre les différentes notions de sécurité d'une fonction de hachage et celles des constructions les utilisant. Par exemple, nous ne connaissons pas aujourd'hui parfaitement le danger réel pour toutes les applications connues de l'apparition d'une attaque trouvant des collisions contre la fonction de hachage. En coopération avec plusieurs membres du projet RNRT *SAPHIR* [Saphir], nous avons comblé en partie ce manque en analysant les implications entre certaines notions de sécurité des fonctions de hachage et celles de schémas de signature [BCC07].

CHAPITRE 2
L'extension de domaine

Sommaire

L'extension de domaine est l'une des deux composantes d'une fonction de hachage itérée. En raison de sa simplicité, l'algorithme de Merkle-Damgård est demeuré longtemps seul et incontesté, mais récemment, des recherches ont montré les limites de cette technique. À présent, de nombreux chercheurs tentent de proposer une nouvelle méthode, à la fois simple et sûre, qui pourrait s'imposer comme nouveau standard.

2.1 L'algorithme de Merkle-Damgård

En 1989, Ralph Merkle [Mer89] et Ivan Damgård [Dam89] proposèrent indépendamment un algorithme d'extension de domaine très simple, dont certaines propriétés de sécurité peuvent être démontrées en supposant certaines propriétés de la fonction de compression interne.

On prépare tout d'abord le message à hacher M en y ajoutant un rembourrage. La fonction de compression prenant en entrée des blocs de message de taille fixe m, le rembourrage permet de ramener la taille du message à hacher à un multiple de m. Pour cela, on rajoute tout d'abord à M un bit à 1 puis u bits à 0, où u est le plus petit nombre positif ou nul tel que la longueur finale du message soit égale à $m - v \pmod m$ (typiquement, $v = 64$ bits). Ensuite, un deuxième rembourrage intervient : on ajoute un bloc de v bits contenant la représentation en base binaire de la taille de M. La taille finale du message rembourré sera bien un multiple de m. On remarque que la taille maximale de message pouvant être hachée est limitée à 2^v bits, mais en pratique ce nombre est assez grand pour ne jamais être atteint. Le message rembourré est ensuite divisé en k blocs de message M_i de m bits chacun, qui serviront à mettre à jour la variable de chaînage H_{i-1} pour donner H_i à l'aide de la fonction de compression h :

$$H_i = \mathrm{h}(H_{i-1}, M_i).$$

La variable de chaînage initiale H_0 est fixé à la valeur d'IV et la dernière variable de chaînage H_k permet de déduire le haché final. En général, l'état interne de la fonction de hachage est de même taille n que le haché final. Dans ce cas, le haché est directement égal à H_k. Si l'état interne est plus grand que la taille de haché, on doit effectuer une certaine troncature pour obtenir la bonne taille de sortie. Il est aussi possible d'appliquer une fonction de sortie sur H_k

avant d'obtenir le haché, mais cette étape est souvent omise en pratique. Le processus entier est décrit en figure 2.1. Dans la suite, sauf mention contraire, nous considérons que la taille de l'état interne de la fonction de hachage est égale à celle du haché : $n = N$.

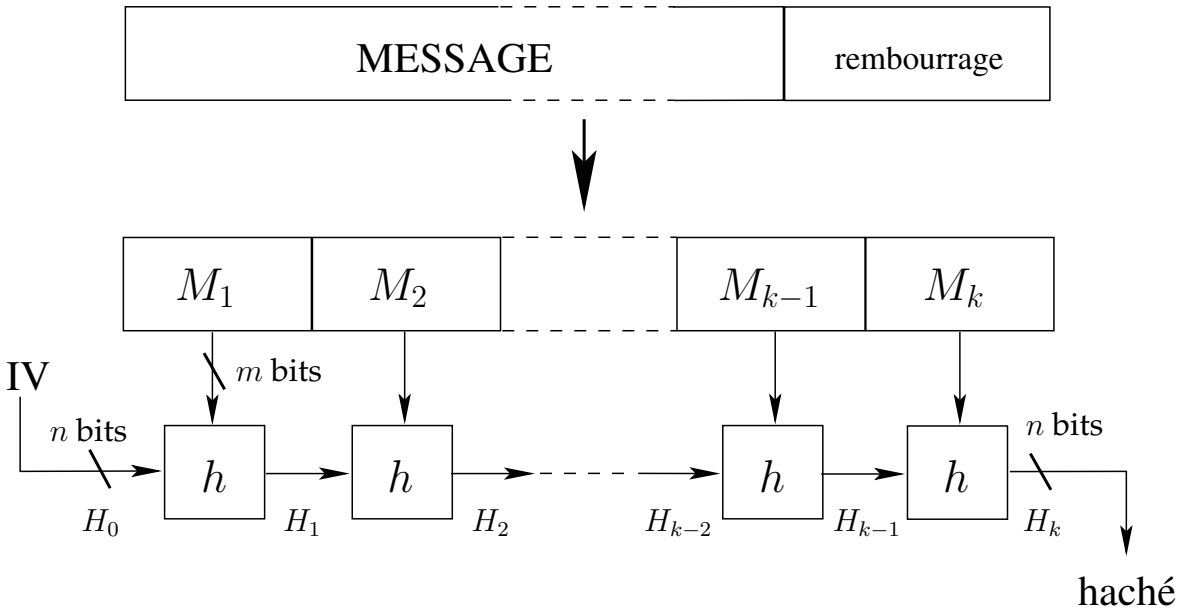

FIG. 2.1 – L'algorithme de Merkle-Damgård [Mer89, Dam89]. Après rembourrage, le message est divisé en blocs de m bits. Chacun de ces blocs M_i opère la mise à jour par la fonction de compression h de la variable de chaînage H_{i-1} pour donner H_i.

Le succès de cet algorithme d'extension de domaine vient de sa preuve de sécurité très simple et très utile. Plus précisément, on peut prouver que toute fonction de compression résistante aux attaques trouvant des pseudo-collisions fournit par cette méthode une fonction de hachage résistante aux attaques trouvant des collisions. On s'en convainc facilement en remarquant qu'une collision sur la fonction de hachage entraîne l'existence d'une pseudo-collision sur la fonction de compression à une certaine itération. Le cas où l'attaquant trouve un message M tel que $\mathrm{h}(\mathrm{IV}, M) = \mathrm{IV}$ est pris en compte par le deuxième rembourrage, qui empêche qu'un message rembourré soit le suffixe d'un autre message rembourré. Il évite aussi certaines attaques triviales, notamment en ce qui concerne les pseudo-collisions : en choisissant $\mathrm{IV}' = \mathrm{h}(\mathrm{IV}, M^1)$, si l'on omet le deuxième rembourrage, nous calculons immédiatement la pseudo-collision $\mathrm{h}(\mathrm{IV}, M^1||M^2) = \mathrm{h}(\mathrm{IV}', M^2)$.

Cette preuve est quelquefois mal interprétée parmi les chercheurs : il est admis par certains auteurs qu'une attaque contre la fonction de compression permettant de trouver des pseudo-collisions rend vulnérable la fonction de hachage totale. Ceci est a priori faux puisqu'aucun algorithme générique ne permet de trouver une collision pour la fonction de hachage à partir de pseudo-collisions pour la fonction de compression interne. Il est néanmoins vrai que la preuve de sécurité n'est plus valide dans ce cas. Il serait d'ailleurs intéressant de trouver une hypothèse nécessaire et suffisante pour cette preuve : il semble trop fort d'empêcher l'attaquant de trouver des pseudo-collisions sur la fonction de compression (sauf à prouver le contraire en établissant une attaque générique), mais, d'autre part, ne considérer que la résistance à la recherche de collisions est trop faible.

Le cas de la préimage est plus simple. La preuve de Merkle-Damgård s'applique de la même manière : une fonction de compression résistante à la recherche de préimages libres aboutira à une fonction de hachage résistante à la recherche de préimages. Cependant, l'hypothèse est ici nécessaire puisqu'un algorithme générique [Pre93] permet de transformer des préimages libres pour la fonction de compression en préimages pour la fonction de hachage.

De manière simplifiée, on choisit $2^{(n+s)/2}$ blocs de message qui nous fournissent $2^{(n+s)/2}$ variables de chaînage à partir de l'IV que l'on stocke dans une première liste. On inverse ensuite $2^{(n-s)/2}$ fois la fonction de compression à partir de la valeur de haché du défi (calcul de préimages libres) pour obtenir encore $2^{(n-s)/2}$ variables de chaînage que l'on stocke dans une deuxième liste. On applique enfin une technique de *rencontre au milieu* : on trouve une valeur de variable de chaînage présente dans les deux listes (en les triant par exemple), ce qui nous donne une préimage. Nous avons une bonne probabilité de succès puisque le paradoxe des anniversaires s'applique dans notre cas : nous avons bien $2^{(n-s)/2} \times 2^{(n+s)/2} = 2^n$ couples possibles. Enfin, la complexité finale est de $O(2^{(n+s)/2})$ opérations si l'on est en mesure de calculer des préimages libres en $O(2^s)$ opérations. On remarque que si $s < n$ (la fonction de compression n'est pas résistante à la recherche de préimages libres), alors $2^{(n+s)/2} < 2^n$ et nous avons bien une meilleure attaque que celle générique pour la recherche de préimages pour la fonction de hachage.

2.2 Les vulnérabilités de l'algorithme de Merkle-Damgård

L'un des premiers problèmes concernant la sécurité de l'algorithme de Merkle-Damgård fut découvert rapidement. Une fonction de hachage H simulant un oracle aléatoire doit fournir des constructions de MAC sûres en utilisant par exemple $\mathrm{MAC}_K(M) = \mathrm{H}(K||M)$, où K représente la clé secrète et M le message. Or, cette construction présente des vulnérabilités si l'on utilise une fonction de hachage fondée sur le principe de Merkle-Damgård. En effet, supposons que l'on demande le MAC pour la clé secrète K du message M_1. On obtient $\mathrm{MAC}_K(M_1) = \mathrm{H}(K||M_1)$. Puisque l'on connaît l'état interne de la fonction de hachage à la dernière itération (c'est le haché), on peut parfaitement calculer $\mathrm{H}(K||M_1||M_2)$, ce qui est égal à $\mathrm{MAC}_K(M_1||M_2)$. Nous avons ainsi engendré un nouveau MAC valide, sans pour autant connaître la clé secrète. Cette vulnérabilité sur l'algorithme d'extension de domaine, appelée attaque par *extension de longueur*, fut corrigée bien plus tard par Coron *et al.* [CDM05].

De nouveaux types d'attaques ont également été mis en évidence. Tout d'abord, Joux introduisit le concept de *multicollisions* [Jou04] : une k-multicollision est un ensemble de k messages aboutissant tous à la même valeur de sortie (une collision est donc une 2-multicollision). Pour une fonction de hachage simulant un oracle aléatoire, on devrait exécuter $O(2^{n\cdot(k-1)/k})$ opérations pour avoir une bonne probabilité d'obtenir une k-multicollision. Dans le cas d'une fonction de hachage fondée sur le principe de Merkle-Damgård, cela est beaucoup plus simple : on trouve tout d'abord une collision sur la fonction de compression à partir de l'IV (avec $O(2^{n/2})$ opérations). Ensuite, à partir de la nouvelle variable de chaînage, on trouve une nouvelle collision interne (encore avec $O(2^{n/2})$ opérations). On continue, de la même manière, jusqu'à forcer k collisions internes. Ces k collisions vont finalement nous donner 2^k chemins différents pour aboutir à la variable de chaînage finale et nous obtenons donc une 2^k-multicollision en exécutant seulement $O(k \times 2^{n/2})$ opérations. Cette technique est décrite en figure 2.2. Si la taille de l'état interne est égale à celle du haché, aucun algorithme d'extension de domaine ne corrige cette vulnérabilité à ce jour excepté le schéma de Maurer *et al.* [MT07].

Ces travaux de Joux ont aussi montré que la concaténation des hachés de deux fonctions

de hachage indépendantes n'améliore pas la résistance en collision autant que l'exigerait l'augmentation de la taille du haché final : en engendrant une $2^{n/2}$-multicollision sur l'un des hachés, on obtient une bonne probabilité de trouver une collision pour le deuxième haché parmi ces candidats. Pour une taille de haché final de $2n$, nous trouvons donc des collisions à l'aide de seulement $n/2 \times 2^{n/2}$ opérations approximativement.

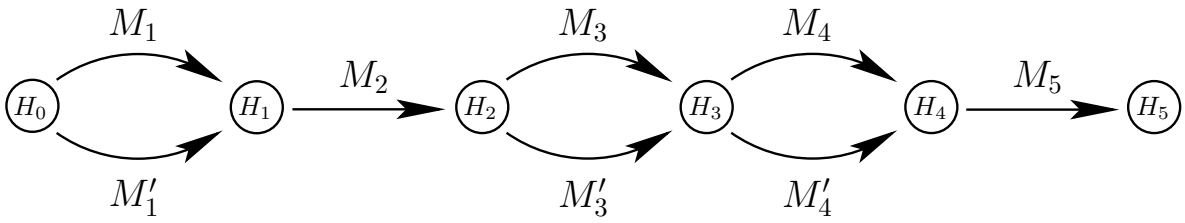

FIG. 2.2 – Exemple de l'attaque par multicollisions de Joux [Jou04] pour l'algorithme d'extension de domaine de Merkle-Damgård. Chaque flèche représente une transition d'une variable de chaînage H_i à une variable de chaînage H_{i+1} par l'application de la fonction de compression h avec le bloc de message M_i. Dans le cas où l'on réalise une collision sur h, on note M'_i le second bloc de message. Dans cet exemple, nous avons 3 collisions internes, ce qui nous fournit une 8-multicollision pour le haché final en seulement $3 \times 2^{n/2}$ opérations approximativement (au lieu de $2^{7n/8}$ opérations dans le cas idéal).

De nouvelles vulnérabilités furent ensuite publiées. Par exemple, Dean [Dea99] puis Kelsey et Schneier [KS05] montrèrent que l'on peut trouver une seconde préimage pour une fonction de hachage fondée sur le principe de Merkle-Damgård en moins de 2^n opérations. Cette attaque, qui peut être vue comme une généralisation du travail précédent de Joux, utilise néanmoins de très longs messages. Plus le message du défi est long, meilleure sera la complexité finale de l'attaque.

On peut enfin noter le travail de Kelsey *et al.* [KK06], qui décrit *l'attaque par rassemblement*, permettant de s'engager à l'avance et de manière frauduleuse sur une valeur de haché. Ces travaux furent étendus plus tard par Dunkelman *et al.* [DP07].

2.3 Les nouveaux algorithmes

Beaucoup de nouveaux candidats apparaissent pour tenter de remplacer l'algorithme d'extension de domaine de Merkle-Damgård. On peut citer par exemple les récentes propositions HAIFA [BD06], EMD [BR06], ROX [ANP07], etc. Néanmoins, même si chacune de ces propositions possède certaines qualités, aucune d'entre elles ne corrige les problèmes de multicollisions ; hormis celle de Maurer *et al.* [MT07] qui permet de concevoir un oracle aléatoire de taille d'entrée variable à partir d'un oracle aléatoire de taille d'entrée fixe au prix d'une construction peu pratique.

L'une des solutions possibles à ce problème serait d'augmenter la taille de l'état interne de la fonction de hachage (à taille de haché constante), pour rendre trop coûteuse la recherche de collisions internes. Pour tenter d'obtenir une fonction de hachage idéale, nous devrons même satisfaire $N \geqslant 2n$. Cette possibilité, déjà pressentie par Joux [Jou04] puis formalisée par Lucks [Luc05], attire de plus en plus les concepteurs de nouveaux schémas, comme en témoignent les dernières fonctions de hachage publiées.

Enfin, les fonctions éponges [BDP06, BDP08] semblent assez prometteuses pour construire des fonctions de hachage. L'idée est d'utiliser un état interne très grand pour éviter toute possibilité de trouver une collision, tout en rendant très légère la fonction de compression. La résistance à la recherche de préimages est renforcée par une fonction de sortie très robuste. Cependant, nous avons montré dans [Pey07] et nous expliquons dans la dernière partie de ce mémoire que pour la fonction de hachage GRINDAHL, très proche des fonctions éponges, la résistance à la recherche de collisions n'est pas satisfaite. Nous avons aussi publié d'autres vulnérabilités des fonctions éponges quant à la construction de MAC [GLP08, Pey08].

CHAPITRE 3
Fonctions de compression

Sommaire

La fonction de compression est la principale composante d'une fonction de hachage itérée. En pratique, c'est aussi souvent la partie la plus vulnérable et donc la plus difficile à construire. Historiquement, on peut distinguer trois manières de procéder pour créer une telle primitive : à partir de rien, en utilisant un algorithme de chiffrement par blocs, ou en se fondant sur un problème difficile.

3.1 Fonctions de compression ad hoc

Les fonctions de compression ad hoc sont en pratique les plus rapides puisqu'elles utilisent des opérations très peu coûteuses. Le désavantage est que la sécurité de ces fonctions est en général conjecturée et n'est pas fondée sur des preuves réductionnistes, car le principe de construction vise avant tout la performance. Les plus connues font partie de la famille MD-SHA [RFCmd4, RFCmd5, RIPE95, DBP96, ZPS92, N-sha0, N-sha1, N-sha2, N-sha2b], initiée par Ron Rivest en 1990. Nous ne décrivons pas en détail ces fonctions, car elles le seront dans la deuxième partie de ce mémoire. On peut tout de même remarquer que même si le fonctionnement interne de ces fonctions ne repose sur aucune primitive cryptographique déjà connue, elles définissent en fait le plus souvent un algorithme de chiffrement par blocs ad hoc E qui est imbriqué dans un schéma de type Davies-Meyer (ce schéma sera décrit dans la prochaine section) :

$$H_i = \mathrm{h}(H_{i-1}, M_i) = E_{M_i}(H_{i-1}) \oplus H_{i-1}.$$

où $E_x(y)$ représente le chiffrement du message y par la clé x. D'ailleurs, des algorithmes de chiffrement par blocs déduits d'une fonction de compression ad hoc ont parfois été proposés. Par exemple, SHACAL [HN00, HN02] est extrait de la fonction de compression de SHA-1 [N-sha1].

De nombreuses fonctions de compression ne font pas partie de la famille MD-SHA. On peut citer par exemple TIGER [AB96], WHIRLPOOL [BR00], SMASH [Knu05], FORK-256 [HCS05,

HCS06, HCS07], LAKE [AMP08], MAME [YW007], etc. Cependant, beaucoup de ces fonctions présentent des vulnérabilités [MR07, PRR05, MLP07, MPB07, Saa07a].

Les fonctions éponges [BDP08], comme RADIOGATÚN [BDP06] ou GRINDAHL [KRT07], utilisent elles aussi des fonctions de compression ad hoc spécialement adaptées à la très grande taille de leur état interne.

Les résultats présentés dans ce mémoire concernent la cryptanalyse de ces fonctions de compression ad hoc. Nous décrivons certaines vulnérabilités pour les fonctions de hachage FORK-256 [MPB07] et GRINDAHL [Pey07] (et plus généralement les fonctions éponges [GLP08, Pey08]). Nous améliorons ensuite les attaques connues en ce qui concerne la recherche de collisions pour SHA-0 [MP08] et SHA-1 [JP07a, JP07b, JP07c, YIN08b]. Enfin, nous avons montré avec Frédéric Muller [MP06] que de nombreux schémas fondés sur des *T-fonctions* [KS02, KS04] ne peuvent pas être considérés comme sûrs. Cependant, ces derniers travaux ne seront pas décrit en détail dans cette thèse.

3.2 Fonctions de compression fondées sur un algorithme de chiffrement par blocs

Les fonctions de compression fondées sur un algorithme de chiffrement par blocs paraissent intéressantes pour les concepteurs. En effet, nous connaissons de nombreux candidats efficaces et sûrs, tels que l'AES, et cette approche débouche parfois sur des preuves de sécurité. Plus précisément, certaines propriétés de la fonction de hachage (résistance à la recherche de collisions, de préimages ou de secondes préimages) peuvent être démontrées en supposant la primitive interne idéale.

Cela fut le cas pour certains des premiers schémas considérés par Preneel *et al.* [PGV93]. Ils étudièrent tous les candidats simples pour construire une fonction de compression à partir d'un algorithme de chiffrement par blocs. Plus précisément, ils analysèrent toutes les constructions de la forme :

$$H_i = \mathrm{h}(H_{i-1}, M_i) = E_{V_1}(V_2) \oplus V_3$$

où V_1, V_2 et V_3 sont des combinaisons linéaires des deux entrées H_{i-1}, M_i. Il y a donc $(2^2)^3 = 64$ candidats au total, dont seulement 12 furent conjecturés sûrs (une vulnérabilité fut décrite pour chacun des autres candidats). Ce n'est que plusieurs années plus tard que Black *et al.* [BRS02] démontrèrent la validité de cette conjecture, dans le modèle du *chiffrement idéal* ou modèle de la *boîte noire* où l'on considère la primitive interne comme une permutation parfaitement aléatoire, voir [Sha49, Win84]. Les plus connus de ces schémas sont ceux de Matyas-Meyer-Oseas, de Davies-Meyer, et de Miyaguchi-Preneel. Ils sont explicités dans la figure 3.1. Chacun d'eux possède un taux d'efficacité égal à 1, défini par le rapport entre le nombre de blocs de message traités et le nombre d'appels à un algorithme de chiffrement par blocs. Néanmoins, il faut noter que pour le schéma de Davies-Meyer il est facile de trouver des *points fixes*, i.e. des valeurs de la variable de chaînage interne restant identiques après application de la fonction de compression : $\mathrm{h}(H_{i-1}, M_i) = H_{i-1}$. Il suffit de déchiffrer le chiffré nul pour E avec une clé égale à un message M_i tiré aléatoirement. Cela fournit une variable de chaînage d'entrée vérifiant

$$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1} = H_{i-1}.$$

Cette vulnérabilité n'est pas très pénalisante puisque la fonction de compression est en principe résistante à la recherche de préimages. Ainsi, il sera très difficile en pratique pour un attaquant d'atteindre un tel point fixe durant l'exécution.

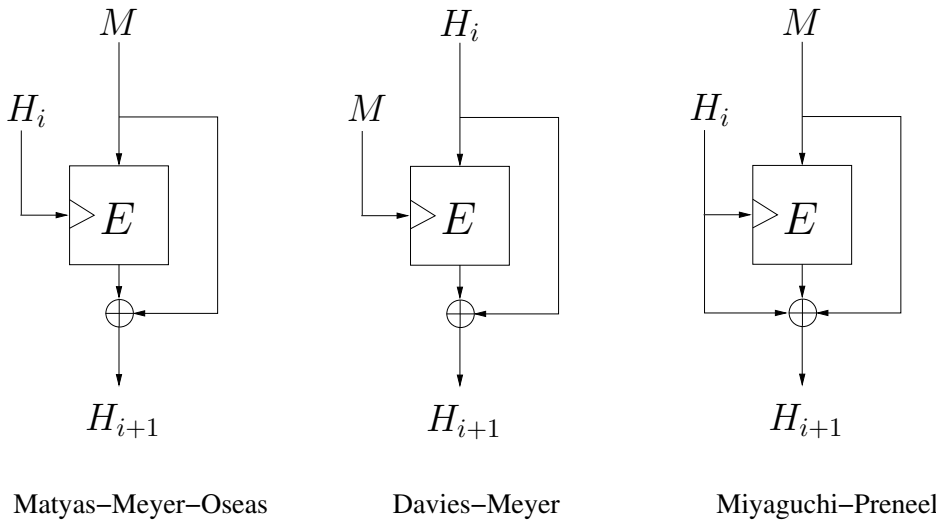

FIG. 3.1 – Les trois schémas les plus connus de fonctions de compression de taille simple fondées sur des algorithmes de chiffrement par blocs. M représente le bloc de message à hacher, H_i et H_{i+1} représentent respectivement l'ancienne et la nouvelle variable de chaînage. Enfin, E est un algorithme de chiffrement par blocs dont l'entrée relative à la clé est marquée par une encoche.

Dans le cas où la taille de bloc de l'algorithme de chiffrement par blocs est égale à celle de la variable de chaînage de la fonction de compression, toutes ces études semblent clore le sujet. Cependant, d'un point de vue pratique, cette configuration n'est pas très utile. En effet, nous disposons actuellement d'algorithmes dont la taille de bloc est inférieure ou égale à 128 bits (dans le cas de l'AES, nous avons 128 bits par bloc). Or, si le haché de la fonction de hachage finale est aussi d'une taille de 128 bits, la recherche générique de collisions ne nécessitera que $O(2^{64})$ opérations. Cette quantité de calculs est actuellement à portée d'une grappe d'ordinateurs puissants.

Ainsi, un problème plus complexe doit être étudié : comment produire des hachés de taille $2n$ (ou plus) lorsque l'on dispose d'un algorithme de chiffrement par blocs opérant sur des blocs de n bits. Les fonctions les plus connues tentant de répondre à ce problème sont MDC-2 (explicité en figure 3.2) et MDC-4 [CPM90]. Elles sont cependant très loin d'offrir une sécurité parfaite proportionnée à leur taille de sortie [Pre93]. De nombreuses propositions de nouveaux schémas furent publiées [PBG89, QG89, BPS90, LM92, LWH93, NLS05], mais peu de ces candidats restèrent indemnes de toute attaque [LM92, KL94, KM05]. Aussi, Bart Preneel et Lars Knudsen [KP96, KP97, KP02] proposèrent des constructions étayées par des arguments de sécurité forts et utilisant la théorie des codes pour définir les combinaisons linéaires d'entrée et de sortie pour l'algorithme de chiffrement par blocs. Cependant, en plus de quelques légères vulnérabilités pour certaines instances [Wat08], ces schémas présentent le désavantage de ne pas offrir une sécurité idéale relativement à la taille de sortie du haché : un haché de taille beaucoup plus grande que $2n$ bits est nécessaire pour prétendre à une sécurité équivalente à celle d'une fonction de hachage idéale dont le haché mesure $2n$ bits. Plus récemment, Hi-

rose [Hir04, Hir06] eut l'idée d'utiliser des algorithmes de chiffrement par blocs dont la taille de clé est double (comme cela est le cas pour AES-256). Cette proposition est accompagnée d'une preuve de sécurité relative à la résistance à la recherche de collisions et de (secondes) préimages dans le modèle du chiffrement idéal.

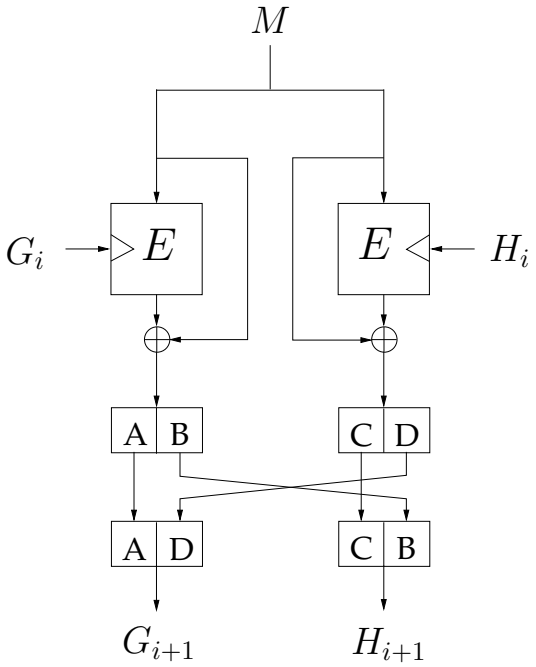

FIG. 3.2 – La fonction de compression de MDC-2 [CPM90]. M représente le bloc de message à hacher, G_i et H_i sont les deux blocs de variable de chaînage d'entrée, G_{i+1} et H_{i+1} dénotent les deux blocs de variable de chaînage de sortie. E est un algorithme de chiffrement par blocs dont l'entrée relative à la clé est marquée par une encoche. Enfin, les notations A, B, C et D désignent des demi-blocs et servent à décrire le mélange réalisé entre la branche de droite et celle de gauche.

En collaboration avec Henri Gilbert, Frédéric Muller et Matt Robshaw, nous avons étudié ce problème dans un article publié à la conférence ASIACRYPT 2006 [PGM06], en essayant de définir un cadre de travail regroupant toutes les attaques connues. Nous avons aussi tenté de généraliser l'analyse en étudiant comment construire une fonction de compression idéale en utilisant un certain nombre de fonctions de compression elles-mêmes idéales mais plus petites. Nous sommes arrivés à la conclusion quelque peu contre-intuitive que dans le cas d'une construction parallèle simple et pour une taille de haché final de 256 bits, il faut utiliser au moins cinq appels à AES pour hacher un bloc de message de 128 bits. Cette borne inférieure montre que les tentatives antérieures des chercheurs de résoudre ce problème en utilisant seulement un ou deux appels à la primitive interne (pour obtenir un bon taux d'efficacité) étaient condamnées à l'échec. Nous avons par ailleurs proposé dans [PGM06] des candidats potentiels, explicités dans la figure 3.3. Aucune vulnérabilité n'a pour l'instant été trouvée pour ces schémas, pour lesquels nous avons apporté de nouveaux arguments de sécurité dans un article publié avec Yannick Seurin à la conférence FSE 2007 [SP07].

Nous conjecturons qu'il existe pour ce problème un compromis entre la taille de description

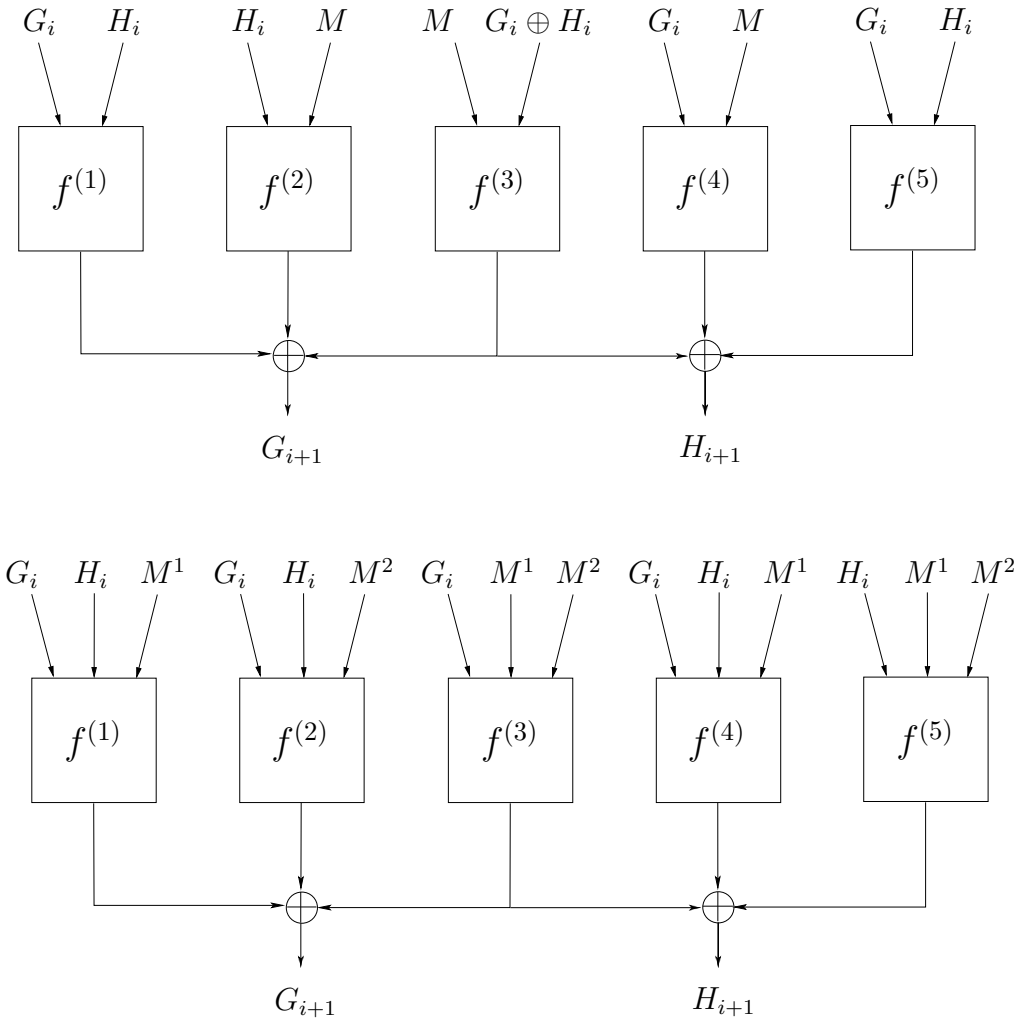

FIG. 3.3 – Deux nouvelles propositions de fonction de compression de taille double fondée sur un algorithme de chiffrement par blocs [PGM06]. M, ou M^1 et M^2, représentent les blocs de message à hacher, G_i et H_i sont les deux blocs de variable de chaînage d'entrée, G_{i+1} et H_{i+1} désignent les deux blocs de variable de chaînage de sortie. Les fonctions $f^{(i)}$ représentent des fonctions de compression indépendantes de taille simple, pouvant par exemple être instanciées en pratique par des algorithmes de chiffrement par blocs indépendants en mode Davies-Meyer. Celles utilisées pour le premier schéma possèdent deux entrées de taille n tandis que celles utilisées pour le deuxième ont trois entrées de n bits (par exemple un algorithme de chiffrement par blocs à taille de clé double en mode Davies-Meyer). Ces schémas ne présentent aucune vulnérabilité connue et l'on remarque que le deuxième schéma possède un taux d'efficacité deux fois meilleur que le premier.

de l'algorithme et son taux d'efficacité. Nos travaux tendent à montrer que plus on s'autorise un schéma complexe (et donc, exécutant de nombreux appels à la primitive interne), plus on a de chances de trouver un candidat sûr et présentant un très bon taux d'efficacité.

Il existe une assez forte ressemblance entre les fonctions de hachage et les algorithmes de chiffrement par blocs actuels. Cette ressemblance est sans doute 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. Cependant, la différence fondamentale est qu'il n'y a aucun secret dans une fonction de hachage. Ainsi, l'attaquant contrôle entièrement l'exécution, contrairement au cas des algorithmes de chiffrements par blocs. Comme l'a récemment remarqué Knudsen [Knu08], on serait donc tenté de penser qu'une fonction de hachage ne devrait pas être plus rapide qu'un algorithme de chiffrement par blocs. Cela n'est pas le cas aujourd'hui, SHA-1 étant beaucoup plus rapide que l'AES. Cependant, il est encore trop tôt pour pleinement mesurer la difficulté de construire un schéma résistant à la recherche de collisions, propriété généralement non exigée des algorithmes de chiffrement par blocs.

3.3 Fonctions de compression fondées sur une structure algébrique

Le dernier groupe de fonctions de compression, celles fondées sur une structure algébrique, est situé à l'opposé de celui des fonctions ad hoc. 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 préimages, de secondes préimages). Depuis les récentes attaques dévastatrices contre les membres de la famille MD-SHA, ces fonctions ont connu un regain d'intérêt certain. 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. De plus, ces fonctions possèdent une forte structure algébrique et cela peut poser problème puisque les fonctions de hachage sont aussi utilisées pour casser de telles structures dans certaines primitives cryptographiques.

Aujourd'hui, pour construire des fonctions de compression, on peut utiliser des problèmes aussi variés que la factorisation [CLS06], le décodage de syndrome [AFS05], ou la recherche du plus court vecteur dans un réseau [GGH96, BPS06]. En collaboration avec Olivier Billet et Matt Robshaw, nous avons proposé une fonction de hachage fondée sur le problème de la résolution de systèmes d'équations quadratiques multivariées dans un corps fini [BPR07], et fournissant une preuve de sécurité en ce qui concerne la résistance à la recherche de préimages et de secondes préimages. Une analyse de sécurité fut conduite plus tard par Aumasson *et al.* [AM07], sans pour autant aboutir à une attaque contre le schéma complet.

Certains schémas de cette famille ne sont pas indemnes de toute vulnérabilité [Saa07b, CMP08, Saa06], mais cette approche semble de plus en plus intéressante à mesure que les vitesses d'exécution des autres familles de schémas diminuent. Cependant, étant donné les performances actuelles, il semble peu probable qu'une telle fonction soit sélectionnée comme algorithme vainqueur de l'appel à soumissions du NIST. On peut enfin noter la récente proposition d'Adi Shamir [Sha08] pour construire un MAC, présentant l'avantage de ne demander qu'une quantité de mémoire très faible et rendant le schéma attractif pour des environnements très contraints.

DEUXIÈME PARTIE

Cryptanalyse de la famille SHA

Nous présentons dans cette section les fonctions de hachage de la famille *Message-Digest* (MD) ou *Secure Hash Algorithm* (SHA), de loin les plus implantées en pratique et les plus étudiées. Ces primitives, très rapides, définissent en fait une fonction de compression qui sera utilisée avec l'algorithme d'extension de domaine de Merkle-Damgård. Elles ont toutes pour caractéristique commune d'utiliser un algorithme de chiffrement par blocs dédié E, à l'intérieur d'une construction de type Davies-Meyer :

$$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$$

où H_{i-1} représente la variable de chaînage d'entrée, H_i celle de sortie et E est fondé sur un schéma de Feistel asymétrique généralisé.

Le premier représentant de cette lignée, MD4 [RFCmd4] [‡], a été conçu en 1990 par Rivest pour les laboratoires RSA. Cette fonction produit des hachés de 128 bits et est spécialement optimisée pour les architectures 32 bits. À la suite de la découverte de vulnérabilités potentielles de MD4, une version améliorée, MD5 [RFCmd5], fut proposée l'année suivante. En 1992, Zheng *et al.* publièrent l'algorithme HAVAL [ZPS92], très proche de la famille MD, mais ayant l'avantage de posséder un paramètre de sécurité et une longueur de sortie paramétrable. Parallèlement, le projet européen *RACE Integrity Primitives Evaluation* (RIPE) recommanda la fonction de hachage RIPEMD-0 [RIPE95] [††], constituée quasiment de deux MD4 mis en parallèle, puis renforcée en 1996 pour donner RIPEMD-128 et RIPEMD-160 [DBP96]. Le NIST ne tarda pas à réagir en standardisant en 1993 SHA-0 [N-sha0], fruit du travail de la *National Security Agency* (NSA). SHA-0, dont la conception est également fortement inspirée par celle des fonctions de la famille MD, est une fonction de hachage produisant des hachés de 160 bits. Cette version fut très légèrement corrigée en 1995, pour des raisons restées confidentielles, et a donné naissance à SHA-1 [N-sha1] [‡‡]. Dans un souci d'anticipation des futures cryptanalyses et de l'augmentation supposée des capacités de calcul, le NIST a publié récemment les fonctions SHA-256 et SHA-512 [N-sha2], des versions améliorées des anciens membres de la famille SHA produisant des hachés de taille 256, 384 ou 512 bits (puis 224 bits [N-sha2b]).

Après plusieurs avancées sur des versions réduites de MD4 [BB91], de MD5 [BB93] ou encore de RIPEMD-0 [Dob97], une première cryptanalyse complète d'un membre de cette lignée permettant de produire des collisions pour MD4 fut publiée par Dobbertin [Dob96a] en 1996. En 1998, Chabaud et Joux [CJ98] établirent la première attaque théorique fournissant une collision pour SHA-0 en moins de 2^{80} opérations (pour 160 bits de sortie). En 2004, Biham et Chen [BC04] introduisirent l'idée de bits neutres, qui mena plus tard au calcul de la première collision sur SHA-0 [BCJ05] avec 4 blocs de message. Durant l'été 2004 [WFL04], la communauté académique découvrit l'existence d'un travail indépendant d'une équipe de chercheurs chinois étudiant la résistance à la recherche de collisions de nombreuses fonctions de la famille MD ou SHA : les premières attaques produisant une collision pour MD5 [WY05], SHA-1 [WYY05b], RIPEMD-0 [WLF05] et certaines versions d'HAVAL [YWY06] ; en plus d'une amélioration significative des attaques existantes pour MD4 [WLF05] et SHA-0 [WYY05d]. Depuis ces importantes avancées, une grande quantité d'articles a permis de mieux comprendre

[‡] MD2, conçu pour les processeurs 8 bits, date de 1989, mais ne repose pas sur les mêmes principes de construction que les autres membres de la famille MD.

[††] Les notations RIPEMD-0, RIPEMD-128 et RIPEMD-160 furent introduites après la création des versions renforcées de RIPEMD.

[‡‡] De manière similaire à RIPEMD, les notations SHA-0 et SHA-1 furent introduites après la création de SHA-1 pour distinguer la version originale de la version corrigée de SHA.

Algorithme	Taille de sortie	Complexité	Référence
MD4	128	2	[SWO07]
MD5	128	2^{23}	[Kli06]
HAVAL 3 tours	128	2^{6}	[YWY06]
HAVAL 3 tours	160 → 256	2^{29}	[VBP03]
HAVAL 4 tours	128 → 256	2^{36}	[YWY06]
HAVAL 5 tours	128 → 256	2^{123}	[YWY06]
RIPEMD-0	128	2^{16}	[WLF05]
RIPEMD-128	128 ou 256		
RIPEMD-160	160 ou 320		
SHA-0	160	2^{33}	[MP08]
SHA-1	160	2^{61}	[MRR07]
SHA-256	256 ou 224		
SHA-512	512 ou 384		

TAB. 3.1 – Meilleures attaques par collision contre les membres de la famille MD-SHA, avec les tailles de sortie (en bits), les complexités des attaques en nombre d'appels à la fonction de compression et les références respectives.

ou d'améliorer encore ces attaques. Le tableau 3.1 donne les attaques produisant des collisions de meilleure complexité actuellement connues et les références correspondantes. Actuellement, un des axes majeurs de recherche concerne l'investigation des implications réelles de ces attaques pour la sécurité des applications cryptographiques pratiques [LW05, CY06, KBP06, Leu07, SYA07, SLW07a, FLN07, WOK08].

Les fonctions de la famille MD-SHA encore considérées comme sûres sont en nombre limité : seules RIPEMD-128, RIPEMD-160, SHA-256 et SHA-512 ont résisté pour l'instant à l'épreuve du temps. Néanmoins, ces avancées inattendues en cryptanalyse ont jeté un doute sur la sécurité de ce type de construction et ceci explique en grande partie la décision du NIST d'organiser un appel à soumissions pour de nouvelles fonctions de hachage, dans le but de standardiser le futur SHA-3 [N-sha3], comme cela fut le cas de l'*Advanced Encryption Standard* (AES) [N-aes] pour les algorithmes de chiffrement par blocs. La date finale de décision du candidat vainqueur est prévue pour le milieu de l'année 2012.

Le travail de cryptanalyse n'est cependant toujours pas terminé puisque l'amélioration des attaques connues permet d'évaluer précisément les limites de ce type de méthodes pour chaque schéma. De plus, même si la fonction SHA-1 est théoriquement cassée, le temps pour calculer une collision est pour l'instant hors de portée des ordinateurs actuels et de nombreuses équipes de recherche tentent de trouver un moyen pour calculer la première collision pour SHA-1, la fonction de hachage actuellement la plus utilisée en pratique.

CHAPITRE 4

Présentation des fonctions de la famille MD-SHA

Sommaire

Toutes les fonctions de hachage décrites dans cette première partie utilisent le procédé de Merkle-Damgård comme algorithme d'extension de domaine, avec préalablement un rembourrage approprié. Nous ne décrirons donc ici que les fonctions de compression, faisant correspondre pour le mot de message M la variable de chaînage de sortie H' à celle d'entrée H. Sauf indication contraire, tous les mots considérés seront des mots de $w = 32$ bits chacun, ce qui explique la rapidité de ces fonctions pour les applications logicielles puisque toutes les opérations utilisées sont disponibles sur les microprocesseurs actuels. Nous notons n le nombre de bits de sortie de la fonction h, m le nombre de mots de message traités à chaque itération de la fonction h, et r le nombre de mots de l'état interne, que l'on appelle *registres*. Sauf dans le cas de RIPEMD où deux branches parallèles sont utilisées (où r représente alors le nombre de registres dans chaque branche), nous aurons $r \times 32 = n$ ce qui signifie que la taille de l'état interne est égale à celle de la variable de chaînage. La variable de chaînage H (respectivement H') peut alors naturellement être divisée en r mots de 32 bits $h_0, \ldots, h_{r-1}$ (respectivement $h'_0, \ldots, h'_{r-1}$).

Les r registres internes sont mis à jour par un schéma de Feistel asymétrique généralisé, chaque registre correspondant à un brin. À chaque étape i, un registre est mis à jour à l'aide d'une fonction f_i dépendante de i, avant le décalage des registres propre au schéma de Feistel. Aussi, certains registres pourront potentiellement subir une rotation avant le décalage. Il n'existe en fait que t fonctions f_i différentes, ce qui définit la notion de tour : pour toutes les étapes du tour j, la même fonction f_j sera utilisée. Soit t le nombre de tours et u le nombre d'étapes par tour. Le nombre total d'étapes de la fonction de compression est égal à $s = t \times u$.

Un mot dépendant du message est requis à l'entrée de chaque étape i, ce qui nous oblige à définir l'expansion de message : à partir du message M constitué de m mots, nous construisons le message étendu W comportant s mots. Il existe deux types d'expansion. Celle de la famille MD ou RIPEMD utilise des permutations des mots de M pour chaque tour. Ainsi, chaque mot de message sera utilisé une et une seule fois par tour. Dans le cas de la famille SHA, une formule de récurrence est définie pour calculer les mots de W, l'initialisation étant fournie par les mots de M.

À chaque étape, seulement un registre sera mis à jour et nous l'appelons *registre cible* (sauf dans le cas de SHA-256 et SHA-512 où nous avons deux registres cibles par étape). Nous pouvons ne considérer que les registres de l'état initial et les registres cibles pour la description de la fonction puisque l'état interne à chaque étape peut être intégralement reconstruit grâce à la seule connaissance de 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} correspondant à la valeur du registre cible mis à jour durant l'étape i pour $0 \leqslant i < s$ (dans le cas de SHA-256 ou de SHA-512 nous aurons deux vecteurs de registres cibles, et il en est de même pour RIPEMD puisque nous devrons maintenir un vecteur de registres cibles pour chaque branche). L'initialisation insérera les registres de la variable de chaînage d'entrée dans les r premiers registres de l'état interne. Cette description simplifiée des algorithmes est utile à la fois pour une meilleure compréhension des schémas et une meilleure vision des évènements dans les registres pour le cryptanalyste [Dau05].

Toutes les fonctions de la famille MD-SHA utilisent la construction de Davies-Meyer, où la variable de chaînage d'entrée est rebouclée sur les registres internes finaux pour donner la variable de chaînage de sortie, et ce, pour éviter toute possibilité d'inversion de la fonction de compression (ce qui aboutirait directement à une attaque par préimage sur la fonction de

hachage complète). Cette technique de rebouclage est connue en anglais sous le nom de *feedforward*. On peut donc voir la fonction de compression comme un algorithme de chiffrement par blocs E (possédant une structure de schéma de Feistel asymétrique généralisé) placé dans une construction de Davies-Meyer. Cet algorithme chiffre la variable de chaînage h avec comme clé le message M, la phase de préparation des clés de E étant la fonction d'expansion de message de la fonction de hachage. Ce formalisme donna même naissance au chiffrement par blocs SHACAL [HN00, HN02], utilisant l'algorithme interne de la fonction de compression de SHA-1. Le schéma de Davies-Meyer a été étudié par Preneel *et al.* [PGV93] parmi d'autres schémas, puis prouvé sûr par Black *et al.* [BRS02] dans le modèle de la boîte noire où le chiffrement par blocs interne E est censé être une primitive parfaite.

Le principal avantage des fonctions de hachage dédiées est leur rapidité dans un environnement logiciel. En effet, seules des opérations très simples et très bien supportées par les microprocesseurs actuels sont utilisées pour la diffusion (additions modulaires, fonctions booléennes bit à bit, rotations et décalages). En ce qui concerne la confusion, l'utilisation de fonctions booléennes et d'additions modulaires semble cryptographiquement robuste. Enfin, seuls quelques registres sont modifiés à chaque étape, il aurait pu en être autrement si les fonctions de la famille MD-SHA n'avaient pas été destinées à des applications logicielles.

Une fois les notations ci-dessus introduites, explicitées dans la figure 4.1 et résumées dans le tableau 4.1, il est très facile de décrire les différents membres de la famille MD-SHA. En fait, il ne reste plus qu'à définir pour chaque cas la méthode d'expansion de message et à préciser par quelle fonction le registre A_{i+1} est mis à jour à chaque étape i. Dans la suite, nous noterons $A^{\lll x}$ (respectivement $A^{\ggg x}$) la rotation de x positions vers la gauche (respectivement vers la droite) des bits du mot A.

Algorithme	n	Sécurité	w	r	m	t	s	Référence
MD4	128	128	32	4	16	3	48	[RFCmd4]
MD5	128	128	32	4	16	4	64	[RFCmd5]
RIPEMD-0	128	128	32	4	16	3	48	[RIPE95]
RIPEMD-128	128	128	32	4	16	4	64	[DBP96]
RIPEMD-128 v. 256 bits	256	128	32	4	16	4	64	[DBP96]
RIPEMD-160	160	160	32	5	16	5	80	[DBP96]
RIPEMD-160 v. 320 bits	320	160	32	5	16	5	80	[DBP96]
SHA-0	160	160	32	5	16	4	80	[N-sha0]
SHA-1	160	160	32	5	16	4	80	[N-sha1]
SHA-256	256	256	32	8	16	1	64	[N-sha2]
SHA-256 v. 224 bits	224	224	32	8	16	1	64	[N-sha2b]
SHA-512	512	512	64	8	16	1	64	[N-sha2]
SHA-512 v. 384 bits	384	384	64	8	16	1	64	[N-sha2]

TAB. 4.1 – Paramètres des fonctions de compression des membres de la famille MD-SHA, avec les tailles de sortie n (en bits), le nombre de bits de sécurité esperés, la taille w (en bits) des mots utilisés, le nombre r de registres internes (par branche dans le cas des fonctions de la famille RIPEMD), le nombre m de mots de message traités par appel à la fonction de compression, le nombre t de tours et le nombre total s d'étapes. Nous indiquons aussi les références des caractéristiques complètes de ces algorithmes.

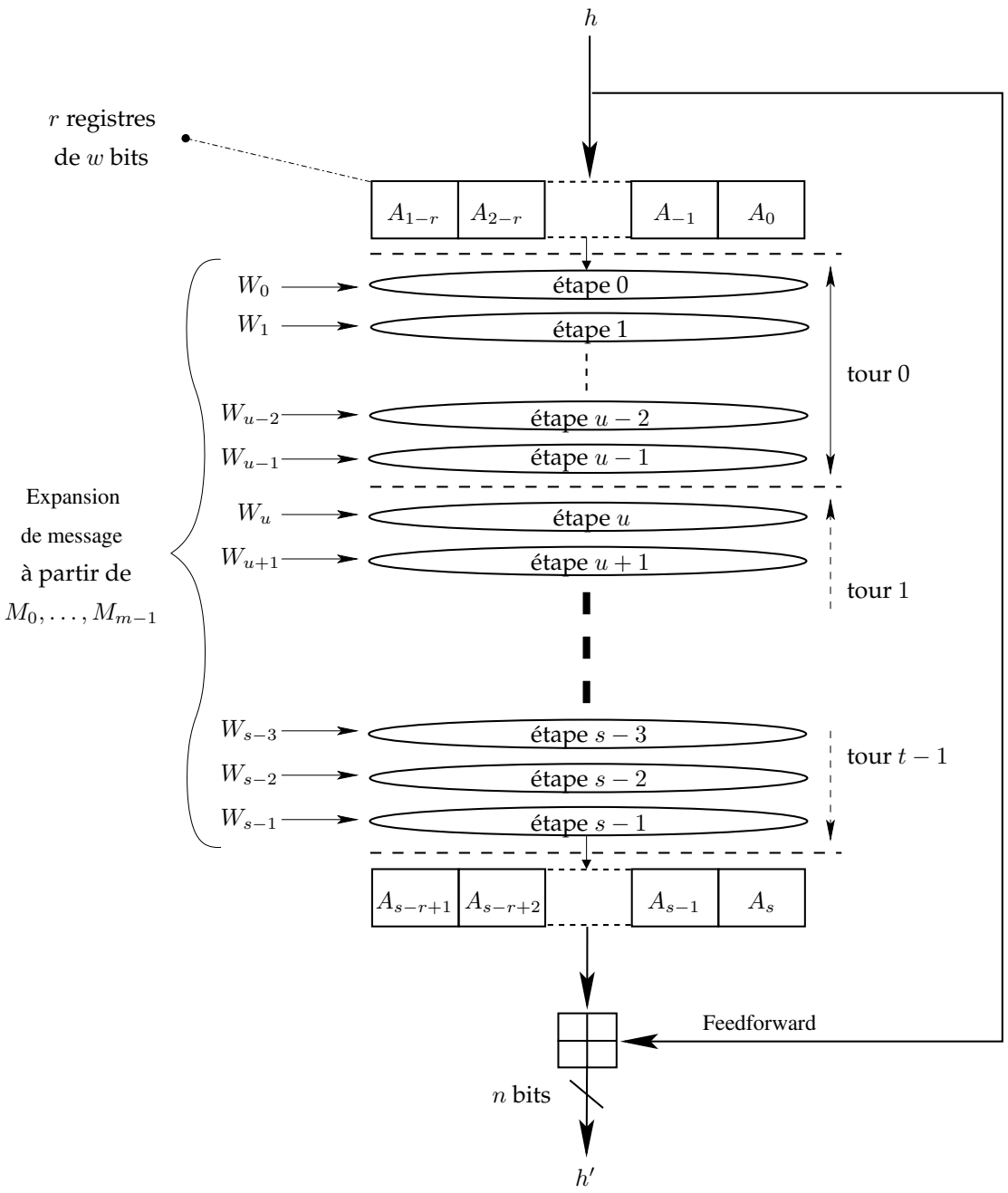

FIG. 4.1 – Structure générique des fonctions de compression pour la famille MD-SHA. Lors de l'étape i, la valeur A_{i+1} est produite.

4.1 MD4

4.1.1 Description

Inventée par Rivest en 1990 pour les laboratoires RSA [RFCmd4], MD4 est la plus ancienne fonction de la famille MD-SHA, c'est donc naturellement aussi la plus simple. La fonction de compression produit des hachés de $n = 128$ bits pour un état interne de $r = 4$ registres de $w = 32$ bits chacun, initialisé par la variable de chaînage d'entrée :

$$A_{-3} = h_0 \qquad A_{-2} = h_3 \qquad A_{-1} = h_2 \qquad A_0 = h_1 \ .$$

À chaque appel, $m = 16$ mots de message seront traités, avec $t = 3$ tours de $u = 16$ étapes chacun (c'est à dire $s = 48$ étapes en tout). L'expansion de message est très simple. Pour chaque tour j une permutation π_j de l'ordre des mots de message est définie (π_0 étant l'application identité). Ainsi, nous avons pour la k-ième étape du tour j, avec $0 \leqslant j \leqslant 2$ et $0 \leqslant k \leqslant 15$:

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

Les permutations π_j sont définies dans la section A.1 de l'appendice. 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.

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

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

où les K_j sont des constantes prédéfinies pour chaque tour, les s_i sont des valeurs de rotation prédéfinies pour chaque étape, et les fonctions Φ_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée (voir section A.1 de l'appendice). Du fait du rebouclage, à la fin des 48 étapes, les mots de la sortie de la fonction de compression sont calculés par :

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

Une description visuelle d'une étape est donnée dans la figure 4.2 et les caractéristiques complètes de la fonction peuvent être trouvées dans [RFCmd4].

4.1.2 Sécurité actuelle

Des vulnérabilités pour la fonction de compression furent très rapidement identifiées juste après la publication de MD4. En effet, dès 1991, Den Boer et Bosselaers [BB91] attaquèrent une version réduite à deux tours au lieu des trois tours que compte MD4 (c'est l'une des raisons de la conception de MD5, dont l'une des différences majeures avec MD4 est qu'elle comporte 4 tours). Après plusieurs avancées, Dobbertin [Dob96a] publia en 1996 la première attaque permettant de trouver des collisions pour la fonction complète dont la complexité en temps est équivalente à 2^{20} appels à la fonction de compression. Huit ans plus tard, en plus d'un grand nombre d'autres cryptanalyses de membres de la famille MD-SHA, Wang *et al.* [WFL04, WLF05] améliorèrent l'attaque pour obtenir une complexité de moins de 2^8 appels à la fonction.

n	128
m	16
w	32
r	4
t	3
u	16
s	48

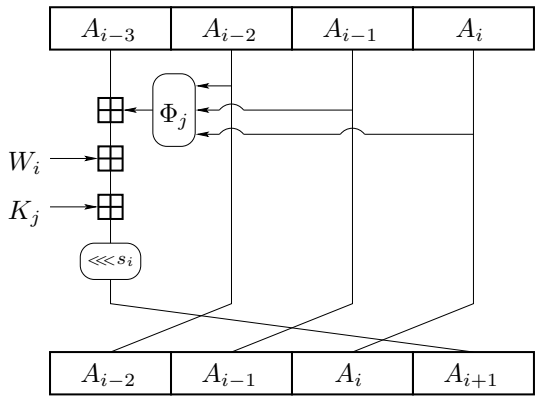

FIG. 4.2 – Une étape de la fonction de compression de MD4.

Les améliorations qui suivirent furent telles qu'aujourd'hui la complexité pour trouver une collision pour MD4 est à peu près égale à celle pour vérifier cette collision [SWO07].

Néanmoins, cette fonction continue d'être étudiée, représentant une bonne entrée en matière dans la cryptanalyse des fonctions de hachage de la famille MD-SHA. De plus, de nouvelles voies de recherche consistent à exploiter ces types d'attaques pour mettre en défaut la sécurité de certains protocoles dans lesquels une fonction de hachage est utilisée (tels que HMAC [FLN07, WOK08] ou APOP [Leu07]). Enfin, cette fonction semble être le point de départ idéal pour essayer d'évaluer les fonctions de la famille MD-SHA quant à leur résistance en préimage [Leu08] ou seconde préimage [WLF05]. Il est aujourd'hui évident que toute utilisation de MD4 comme primitive cryptographique est à proscrire.

4.2 MD5

4.2.1 Description

En réponse aux attaques sur des versions réduites de MD4, une nouvelle version plus complexe fut créée. Comparée à celle de MD4, la fonction de compression de MD5 possède un tour de plus, de nouvelles fonctions booléennes, une diffusion accrue dans la fonction d'étape, des constantes définies pour chaque étape, etc. Les paramètres généraux restent inchangés : 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 :

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

À chaque appel, $m = 16$ mots de message seront traités, durant $t = 4$ tours de $u = 16$ étapes chacun (c'est à dire $s = 64$ 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 étant l'application identité). Ainsi, nous avons pour la k-ième étape du tour j, avec $0 \leqslant j \leqslant 3$ et $0 \leqslant k \leqslant 15$:

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

Les permutations π_j sont définies dans la section A.2 de l'appendice. 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 identique à celle de MD4 à ceci près que les permutations ont été légèrement modifiées.

Durant 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 :

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

où les K_i sont des constantes prédéfinies pour chaque étape, les s_i sont des valeurs de rotation prédéfinies pour chaque étape, et les fonctions Φ_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée (voir section A.2 de l'appendice). À la fin des 64 étapes, les mots de la sortie de la fonction de compression sont calculés par :

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

Une description visuelle d'une étape est donnée dans la figure 4.3 et les caractéristiques complètes de la fonction peuvent être trouvées dans [RFCmd5].

n	128
m	16
w	32
r	4
t	4
u	16
s	64

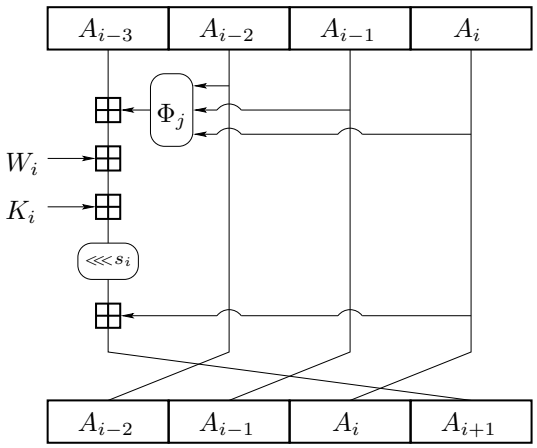

FIG. 4.3 – Une étape de la fonction de compression de MD5.

4.2.2 Sécurité actuelle

La fonction de hachage MD5 résista un peu plus longtemps que MD4 aux efforts des cryptanalystes et, pour cette raison, elle réussit à mieux pénétrer le monde industriel et fut implantée dans de très nombreuses applications. Une pseudo-collision fut tout d'abord découverte en 1993 [BB93] pour la fonction de compression de MD5, puis une attaque trouvant des collisions contre la fonction de hachage complète (mais avec une valeur d'initialisation différente de celle prédéfinie) fut présentée par Hans Dobbertin [Dob96b] en 1996. C'est grâce au travail de Wang et *al.* [WFL04, WY05] en 2004 que put être calculée la première collision pour MD5, avec une complexité en temps équivalente à 2^{39} appels à la fonction de compression. Beaucoup de progrès ont été faits 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 standard.

Comme pour MD4, beaucoup d'équipes de recherche continuent d'analyser MD5, et ce, pour plusieurs raisons. Tout d'abord, une meilleure compréhension des fonctions de hachage de la famille MD-SHA est nécessaire. De plus, MD5 est encore utilisée dans certaines implantations, ce qui pose la question des conséquences de ces attaques. Par exemple, de nombreux travaux [Leu07, SYA07, FLN07, WOK08] analysent la possibilité de compromettre la sécurité de HMAC ou d'APOP lorsque MD5 est utilisé comme brique interne. De nouvelles vulnérabilités ont récemment été identifiées concernant l'utilisation de MD5 pour vérifier l'intégrité de programmes exécutables [SLW07b] ou pour la signature de certificats *X.509* [LW05, SLW07a]. Même si la résistance de la fonction de compression quant à la recherche de préimages n'est pour l'instant toujours pas mise en défaut, MD5 ne doit plus être implantée dans des applications cryptographiques.

4.3 HAVAL

4.3.1 Description

La fonction de hachage HAVAL [ZPS92], très inspirée de MD4, a été inventée par Zheng *et al.*. L'une des nouveautés est que le nombre de tours et la taille des hachés peuvent être modulés, donnant ainsi naissance à toute une série de fonctions HAVAL permettant de couvrir un éventail de besoins le plus large possible. 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 relativement simple de troncature de la sortie originale de 256 bits, voir [ZPS92]) pour un état interne de $r = 8$ registres de $w = 32$ bits chacun, initialisé par la variable de chaînage d'entrée. À chaque appel, $m = 32$ mots de message seront traités, avec t tours de $u = 32$ étapes chacun (c'est à dire $s = t \times 32$ étapes en tout). Suivant la version utilisée, le nombre de tours t peut être égal à 3, 4 ou 5. Cette deuxième possibilité de modularité 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 le réimplanter complètement. Cela permet de mieux anticiper et gérer les avancées potentielles en cryptanalyse. Une telle fonctionnalité est requise pour tout candidat à l'appel à soumissions du NIST. Les caractéristiques complètes de la fonction HAVAL peuvent être trouvées dans [ZPS92].

4.3.2 Sécurité actuelle

Comme pour MD4 ou MD5, les cryptologues analysèrent tout d'abord des versions réduites d'HAVAL [KP00, PSC02, HSK03], ou tentèrent de trouver des comportements non aléatoires dans la fonction [YBC04]. La première attaque contre une version complète d'HAVAL (version 3 tours et n'importe quelle taille de sortie) est due à Van Rompay *et al* [VBP03] et permet de trouver des collisions avec une complexité de 2^{29} appels à la fonction de compression. Un an plus tard, Wang *et al* [WFL04] réduisirent la complexité à seulement 2^6 appels pour la version 3 tours avec 128 bits de sortie, puis les versions 4 et 5 tours succombèrent en 2006 [YWY06] (2^{36} et 2^{123} appels à la fonction de compression respectivement). Pour ces raisons, HAVAL ne doit plus être implantée dans des applications cryptographiques.

4.4 RIPEMD-0

4.4.1 Description

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 Evaluation* (RIPE) sur les primitives permettant de garantir l'intégrité [RIPE95]. 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 $r = 4$ registres de $w = 32$ 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 :

$$\begin{array}{llll} A_{-3}^L = h_0 & A_{-2}^L = h_1 & A_{-1}^L = h_2 & A_0^L = h_3 \\ A_{-3}^R = h_0 & A_{-2}^R = h_1 & A_{-1}^R = h_2 & A_0^R = h_3\,. \end{array}$$

À chaque appel, $m = 16$ mots de message sont traités, durant $t = 3$ tours de $u = 16$ étapes chacun dans chaque branche (c'est à dire $s = 48$ 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 (π_0 étant l'application identité). Ainsi, nous avons pour la k-ième étape du tour j, avec $0 \leqslant j \leqslant 2$ et $0 \leqslant k \leqslant 15$:

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

Les permutations π_j sont définies dans la section A.3 de l'appendice. 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 identique à celle de MD4 à ceci près que les permutations ont été légèrement modifiées.

Durant chaque étape i, les registres cibles A_{i+1}^L et A_{i+1}^R sont mis à jour à l'aide des fonctions f_j^L et f_j^R, dépendantes du tour j auquel appartient i :

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

où les K_j^L et K_j^R 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 Φ_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée (voir section A.3 de l'appendice). 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_j^L et K_j^R. À la fin des 48 étapes des deux branches, on applique le rebouclage pour calculer les mots de la sortie de la fonction de compression :

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

Une description visuelle d'une étape est donnée dans la figure 4.4 et les caractéristiques complètes de la fonction peuvent être trouvées dans [RIPE95].

n	128
m	16
w	32
r	4
t	3
u	16
s	48

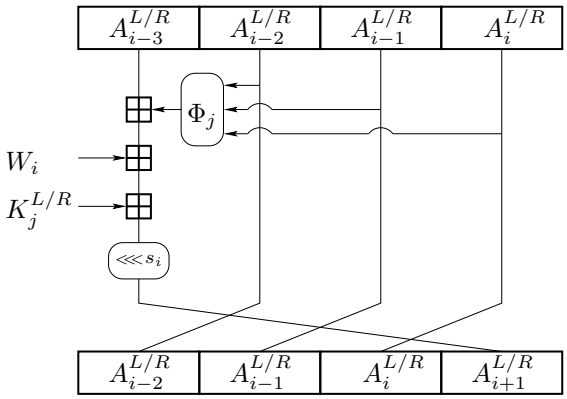

FIG. 4.4 – Une étape pour une branche de la fonction de compression de RIPEMD-0.

4.4.2 Sécurité actuelle

Comme pour ses prédécesseurs, des versions réduites de RIPEMD-0 furent 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 fut publiée en 2004 par Wang *et al.* [WFL04, WLF05], et présente une complexité de 2^{16} appels à la fonction de compression. Comme pour MD4 ou MD5, l'utilisation de RIPEMD-0 dans une application cryptographique est à proscrire.

4.5 RIPEMD-128

4.5.1 Description

En 1996, Hans Dobbertin, Antoon Bosselaers et Bart Preneel [DBP96] proposèrent une version renforcée de RIPEMD-0 pour contrer les premières cryptanalyses de MD4 et de RIPEMD-0 qui apparaissaient. De plus, une version 256 bits a aussi été définie, mais pour une sécurité équivalant à une fonction de 128 bits. Cette nouvelle primitive comporte toujours 2 branches de $r = 4$ registres de $w = 32$ bits. On note A^L_i les registres cibles de la branche de gauche et A^R_i ceux de la branche de droite. On initialise chaque branche par la variable de chaînage d'entrée :

$$A^L_{-3} = h_0 \qquad A^L_{-2} = h_1 \qquad A^L_{-1} = h_2 \qquad A^L_0 = h_3$$
$$A^R_{-3} = h_0 \qquad A^R_{-2} = h_1 \qquad A^R_{-1} = h_2 \qquad A^R_0 = h_3\,.$$

À chaque appel, $m = 16$ mots de message seront traités, avec maintenant $t = 4$ tours de $u = 16$ étapes chacun dans chaque branche (c'est à dire $s = 64$ étapes en tout). 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 π_j^L sont définies. Ainsi, nous avons pour la k-ième étape du tour j, avec $0 \leqslant j \leqslant 3$ et $0 \leqslant k \leqslant 15$:

$$\begin{aligned} W^L_{j\times 16+k} &= M_{\pi_j^L(k)} \\ W^R_{j\times 16+k} &= M_{\pi_j^R(k)}. \end{aligned}$$

Les permutations π_j^L et π_j^R sont définies dans la section A.4 de l'appendice. 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.

Durant chaque étape i, les registres cibles A^L_{i+1} et A^R_{i+1} sont mis à jour par les fonctions f_j^L et f_j^R, dépendantes du tour j auquel appartient i :

$$\begin{aligned} A^L_{i+1} &= f_j^L(A^L_i, A^L_{i-1}, A^L_{i-2}, A^L_{i-3}, W^L_i, s^L_i) \\ &= (A^L_{i-3} + \Phi_j^L(A^L_i, A^L_{i-1}, A^L_{i-2}) + W^L_i + K^L_j)^{\lll s^L_i}, \\ A^R_{i+1} &= f_j^R(A^R_i, A^R_{i-1}, A^R_{i-2}, A^R_{i-3}, W^R_i, s^R_i) \\ &= (A^R_{i-3} + \Phi_j^R(A^R_i, A^R_{i-1}, A^R_{i-2}) + W^R_i + K^R_j)^{\lll s^R_i}, \end{aligned}$$

où les K_j^L et K_j^R sont des constantes prédéfinies pour chaque tour et pour chaque branche, les s_i^L et s_i^R sont des valeurs de rotation prédéfinies pour chaque étape et pour chaque branche, et les fonctions Φ_j^L et Φ_j^R sont des fonctions booléennes définies pour chaque tour et pour chaque branche et prenant 3 mots de 32 bits en entrée (voir section A.4 de l'appendice). On peut noter que les fonctions booléennes Φ_j^L et Φ_j^R (et les constantes de rotation s_i^L et s_i^R) utilisées dans chaque branche sont différentes et ceci constitue la deuxième grande distinction 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 le rebouclage pour obtenir la bonne taille de sortie, un mélange s'opérant en inversant certains registres à la fin de chaque tour. Une description visuelle d'une étape est donnée dans la figure 4.5 et les caractéristiques complètes de la fonction RIPEMD-128 peuvent être trouvées dans [DBP96].

4.5.2 Sécurité actuelle

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 du cryptanalyste.

n	128
m	16
w	32
r	4
t	4
u	16
s	64

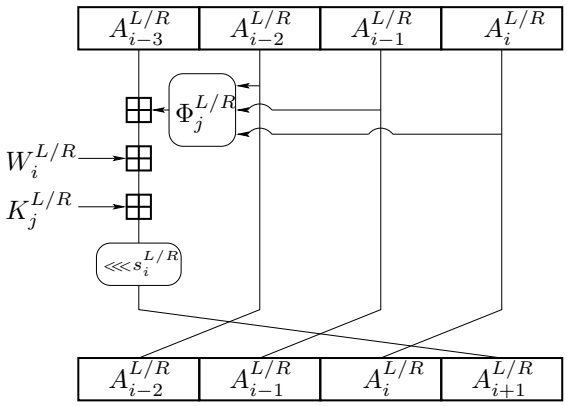

FIG. 4.5 – Une étape pour une branche de la fonction de compression de RIPEMD-128.

4.6 RIPEMD-160

4.6.1 Description

RIPEMD-160 [DBP96] est une fonction de hachage de 160 bits qui fut publiée en même temps que RIPEMD-128 et qui représente une version plus robuste en raison de sa taille de sortie plus grande, et aussi grâce à sa fonction de compression un peu plus complexe : un tour et un registre interne par branche sont rajoutés. Comme pour RIPEMD-128, une version doublant la taille de sortie est définie, permettant des hachés de 320 bits pour une sécurité de 160 bits. Comme pour tous les membres de la famille RIPEMD, deux branches parallèles sont utilisées, mais chacune possédant cette fois $r = 5$ registres de $w = 32$ 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 :

$$A_{-4}^L = (h_0)^{\ggg 10} \quad A_{-3}^L = (h_4)^{\ggg 10} \quad A_{-2}^L = (h_3)^{\ggg 10} \quad A_{-1}^L = h_2 \quad A_0^L = h_1$$
$$A_{-4}^R = (h_0)^{\ggg 10} \quad A_{-3}^R = (h_4)^{\ggg 10} \quad A_{-2}^R = (h_3)^{\ggg 10} \quad A_{-1}^R = h_2 \quad A_0^R = h_1 \,.$$

À chaque appel, $m = 16$ mots de message seront traités, avec $t = 5$ tours de $u = 16$ étapes chacun dans chaque branche (c'est à dire $s = 80$ étapes en tout). Comme pour RIPEMD-128, à chaque tour j et pour chaque branche, des permutations de l'ordre des mots de message π_j^R et π_j^L sont définies. Ainsi, nous avons pour la k-ième étape du tour j, avec $0 \leqslant j \leqslant 4$ et $0 \leqslant k \leqslant 15$:

$$W_{j\times 16+k}^L = M_{\pi_j^L(k)}$$
$$W_{j\times 16+k}^R = M_{\pi_j^R(k)}.$$

Les permutations π_j^L et π_j^R sont définies dans la section A.5 de l'appendice. Comme dans RIPEMD-128, chaque mot de M sera utilisé une fois pour chaque tour dans chaque branche et l'ordonnancement des mots de message est différent dans les deux branches.

Durant 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 :

$$\begin{aligned} A^L_{i+1} &= f^L_j(A^L_i, A^L_{i-1}, A^L_{i-2}, A^L_{i-3}, A^L_{i-4}, W^L_i, s^L_i) \\ &= ((A^L_{i-4})^{\lll 10} + \Phi^L_j(A^L_i, A^L_{i-1}, (A^L_{i-2})^{\lll 10}) + W^L_i + K^L_j)^{\lll s^L_i} + (A^L_{i-3})^{\lll 10}, \\ A^R_{i+1} &= f^R_j(A^R_i, A^R_{i-1}, A^R_{i-2}, A^R_{i-3}, A^R_{i-4}, W^R_i, s^R_i) \\ &= ((A^R_{i-4})^{\lll 10} + \Phi^R_j(A^R_i, A^R_{i-1}, (A^R_{i-2})^{\lll 10}) + W^R_i + K^R_j)^{\lll s^R_i} + (A^R_{i-3})^{\lll 10}, \end{aligned}$$

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 Φ^L_j et Φ^R_j sont des fonctions booléennes définies pour chaque tour et pour chaque branche, et prenant 3 mots de 32 bits en entrée (voir section A.5 de l'appendice). En raison du rebouclage, à la fin des 80 étapes des deux branches, les mots de la sortie de la fonction de compression sont calculés par :

$$\begin{aligned} h'_0 &= A^L_{79} + (A^R_{78})^{\lll 10} + A_0 & h'_1 &= (A^L_{78})^{\lll 10} + (A^R_{77})^{\lll 10} + A_{-1} \\ h'_2 &= (A^L_{77})^{\lll 10} + (A^R_{76})^{\lll 10} + (A_{-2})^{\lll 10} & h'_3 &= (A^L_{76})^{\lll 10} + A^R_{80} + (A_{-3})^{\lll 10} \\ h'_4 &= A^L_{80} + A^R_{79} + (A_{-4})^{\lll 10}\,. \end{aligned}$$

Quant à la version 320 bits, les deux branches sont gardées séparées durant le rebouclage pour obtenir la bonne taille de sortie, un mélange s'opérant en inversant certains registres à la fin de chaque tour. Une description visuelle d'une étape est donnée dans la figure 4.6 et les caractéristiques complètes de la fonction RIPEMD-160 peuvent être trouvées dans [DBP96].

n	160
m	16
w	32
r	5
t	5
u	16
s	80

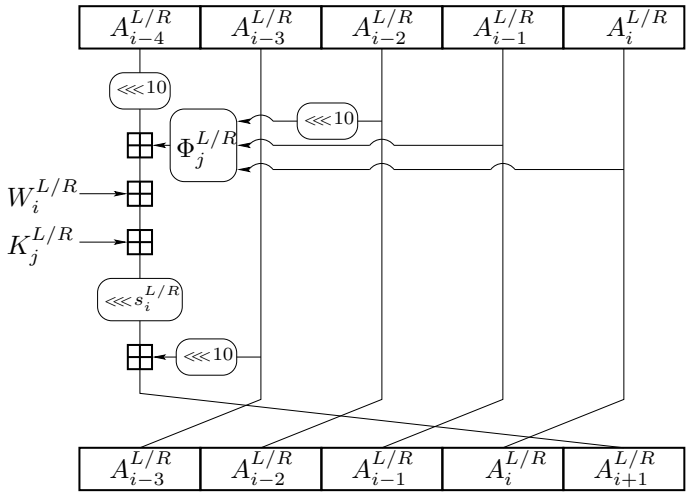

FIG. 4.6 – Une étape pour une branche de la fonction de compression de RIPEMD-160.

4.6.2 Sécurité actuelle

Avec son état interne plus grand et un tour de plus, RIPEMD-160 semble un peu plus robuste que RIPEMD-128. Pour les mêmes raisons que dans le cas de RIPEMD-128, aucune des versions de RIPEMD-160 n'a été attaquée jusqu'à présent.

4.7 SHA-0

4.7.1 Description

Publié en 1993 [N-sha0], SHA-0 est le premier membre de la famille Secure Hash Standard, les fonctions de hachage standardisées par le NIST. Très inspirée de celles de la famille MD, la fonction de compression de SHA-0 n'en diffère quasiment que par l'utilisation d'une expansion de message novatrice : 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. Pour permettre une longévité suffisante de l'algorithme quant à l'augmentation de la puissance de calcul, 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 :

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

À chaque appel, $m = 16$ mots de message seront traités, avec $t = 4$ tours de $u = 20$ étapes chacun (c'est à dire $s = 80$ étapes en tout). L'expansion de message n'utilise donc plus de permutation. Les 16 premiers mots du message étendu sont égaux aux 16 mots de message d'entrée de la fonction de compression. Le reste des W_i sont calculés par une formule de récurrence :

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

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

$$\begin{aligned} A_{i+1} &= f_j(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, W_i) \\ &= (A_i)^{\lll 5} + \Phi_j(A_{i-1}, (A_{i-2})^{\ggg 2}, (A_{i-3})^{\ggg 2}) + (A_{i-4})^{\ggg 2} + W_i + K_j, \end{aligned}$$

où les K_j sont des constantes prédéfinies pour chaque tour et les fonctions Φ_j sont des fonctions booléennes définies pour chaque tour et prenant 3 mots de 32 bits en entrée (voir section A.6 de l'appendice). À la fin des 80 étapes, les mots de la sortie de la fonction de compression sont calculés par :

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

Une description visuelle d'une étape est donnée dans la figure 4.7 et les caractéristiques complètes de la fonction peuvent être trouvées dans [N-sha0].

4.7.2 Sécurité actuelle

La fonction de hachage SHA-0 fut rapidement mise à jour en 1995 pour donner naissance à SHA-1 [N-sha1], sans qu'aucun détail soit donné par la NSA quant aux vulnérabilités éventuelles. Un premier élément de réponse fut apporté par Chabaud et Joux en 1998 [CJ98]. En effet, cette première attaque théorique calculant des collisions pour SHA-0, d'une complexité de 2^{61} appels à la fonction de compression, ne s'applique pas à la version améliorée SHA-1. On

n	160
m	16
w	32
r	5
t	4
u	20
s	80

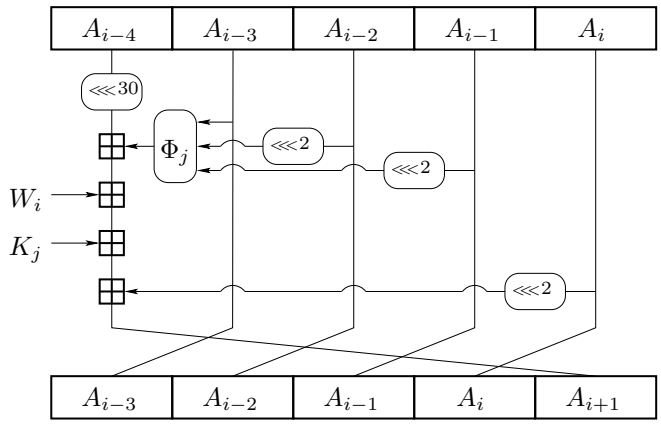

FIG. 4.7 – Une étape de la fonction de compression de SHA-0 ou de SHA-1.

peut donc penser que la NSA avait déjà identifié l'un des problèmes de SHA-0, à savoir sa diffusion. Par comparaison à la famille MD ou RIPEMD, le principe d'une formule de récurrence pour calculer les mots du message étendu semble une bonne idée. En termes de diffusion, de bien meilleures propriétés sont obtenues. Par exemple, une perturbation sur un mot du message d'entrée M (modifiant directement l'un des premiers mots de W) influera très rapidement sur 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 des familles MD et RIPEMD. Cependant, dans le cas de SHA-0, cette diffusion ne s'opère que sur une seule position de bit. Ainsi, si l'on se cantonne 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, il fallut attendre plusieurs années avant de voir progressivement apparaître de nouvelles améliorations. Tout d'abord, Biham et Chen [BC04] introduisirent en 2004 la notion de bits neutres pour accélérer la recherche de collisions. Grâce à cette nouvelle technique, ils calculèrent des presque collisions pour la fonction de compression. Ensuite, en 2005, en utilisant plusieurs blocs de message contenant des différences, Biham *et al.* [BCJ05] publièrent la première collision pour SHA-0 et apportèrent ainsi une preuve pratique du bien-fondé des hypothèses des précédents travaux. La complexité étant de l'ordre de 2^{51} appels à la fonction de compression, un très grand nombre d'ordinateurs furent nécessaires pour trouver la collision en seulement trois semaines. Cette complexité fut améliorée tout d'abord par les travaux de Wang *et al.* [WYY05d], calculant des collisions pour une complexité annoncée d'approximativement 2^{39} appels à la fonction de compression, puis par Naito *et al.* [NSS06]. Finalement, nous présenterons dans cette thèse la meilleure attaque connue à ce jour [MP08], qui calcule des collisions en seulement une heure de calcul sur un ordinateur personnel (2^{33} appels à la fonction de compression). Même si la résistance de la fonction de compression quant à la recherche de préimages n'est pour l'instant toujours pas mise en défaut, SHA-0 ne doit plus être implantée dans des applications cryptographiques.

4.8 SHA-1

4.8.1 Description

SHA-1 est la version corrigée de SHA-0, et fut publiée en 1995 par le NIST [N-sha1]. L'expansion de message mise à part, la description de SHA-1 est absolument identique à celle de SHA-0. La seule et unique différence consiste en une rotation dans la formule de récurrence de l'expansion de message, ce qui implique une meilleure diffusion :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})^{\lll 1}, & \text{pour } 16 \leqslant i \leqslant 79 \end{cases}$$

Comme pour SHA-0, la description visuelle d'une étape est donnée dans la figure 4.7. Les caractéristiques complètes de la fonction peuvent être trouvées dans [N-sha1].

4.8.2 Sécurité actuelle

SHA-1 sembla résister relativement bien aux attaques contre sa version antérieure SHA-0, seules des vulnérabilités sur des versions très réduites furent publiées [BCJ05]. Ceci étant dû au fait que la diffusion des mots de message y est bien meilleure grâce à la rotation ajoutée. Pour ces raisons, et à cause des problèmes de sécurité rencontrés par les membres de la famille MD ou RIPEMD, SHA-1 est la fonction la plus implantée et la plus utilisée actuellement. Seulement, Wang *et al.* [WYY05b] surprirent toute la communauté de la recherche académique en cryptographie en exposant la première attaque théorique contre SHA-1, en 2^{69} appels à la fonction de compression (améliorée plus tard pour une complexité annoncée de 2^{63} appels à la fonction de compression [WYY05a, WYY05c]). Cette attaque, assez complexe et difficile d'accès, était d'autant plus surprenante qu'elle était le fruit de plusieurs années de recherche sans vraiment utiliser d'outils informatiques. Plusieurs travaux s'ensuivirent pour essayer de mieux comprendre, théoriser, ou automatiser ces nouvelles attaques [CR06, JP07a]. À présent, la recherche de la première collision réelle est l'un des principaux objectifs et un calcul distribué pour réaliser ce but a récemment été démarré [MRR07] (pour une complexité annoncée inférieure à 2^{61} appels à la fonction de compression). Nous présentons dans cette thèse la meilleure attaque pratique [JP07c] qui calcule une collision sur une version réduite à 70 tours de SHA-1 en moins de 4 jours sur un ordinateur personnel. Ainsi, même si aucune collision sur la version complète n'a encore été trouvée et même si la résistance en préimage n'est pour l'instant toujours pas mise en défaut, 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 pour de nouvelles fonctions de hachage, dans le but de standardiser le futur SHA-3 [N-sha3].

4.9 SHA-256

4.9.1 Description

Publiée en 2002 [N-sha2], SHA-256 fait partie des derniers membres en date de la famille MD-SHA. Outre sa taille de sortie, 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 ou 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 [N-sha2b] introduite 2 ans plus tard. On maintient donc un état interne de $r = 8$ registres de $w = 32$ bits chacun, initialisé par la variable de chaînage d'entrée :

$$\begin{array}{llll} A_{-3} = h_3 & A_{-2} = h_2 & A_{-1} = h_1 & A_0 = h_0 \\ B_{-3} = h_7 & B_{-2} = h_6 & B_{-1} = h_5 & B_0 = h_4\,. \end{array}$$

À chaque appel, $m = 16$ mots de message seront traités, avec $s = 64$ étapes (on peut noter $t = 1$ et $u = 64$ dans notre formalisme). L'expansion de message est beaucoup plus complexe que dans les autres versions de SHA, mais utilise toujours une formule de récurrence :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{pour } 16 \leqslant i \leqslant 63 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_0(x) = (x^{\ggg 7}) \oplus (x^{\ggg 18}) \oplus (x^{\gg 3}) \\ \sigma_1(x) = (x^{\ggg 17}) \oplus (x^{\ggg 19}) \oplus (x^{\gg 10}). \end{cases}$$

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 :

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

où les K_i sont des constantes prédéfinies pour chaque étape et les fonctions MAJ et IF sont des fonctions booléennes définies en section A de l'appendice. Les fonctions Σ_0 et Σ_1 sont définies par :

$$\begin{aligned} \Sigma_0(x) &= (x^{\ggg 2}) \oplus (x^{\ggg 13}) \oplus (x^{\gg 22}) \\ \Sigma_1(x) &= (x^{\ggg 6}) \oplus (x^{\ggg 11}) \oplus (x^{\gg 25}). \end{aligned}$$

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

$$\begin{array}{llll} h'_0 = A_{64} + A_0 & h'_1 = A_{63} + A_{-1} & h'_2 = A_{62} + A_{-2} & h'_3 = A_{61} + A_{-3} \\ h'_4 = B_{64} + B_0 & h'_5 = B_{63} + B_{-1} & h'_6 = B_{62} + B_{-2} & h'_7 = B_{61} + B_{-3}\,. \end{array}$$

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 description visuelle d'une étape est donnée dans la figure 4.8 et les caractéristiques complètes de la fonction 256 bits peuvent être trouvées dans [N-sha2] et dans [N-sha2b] pour la fonction 224 bits.

n	256
m	16
w	32
r	8
t	1
u	64
s	64

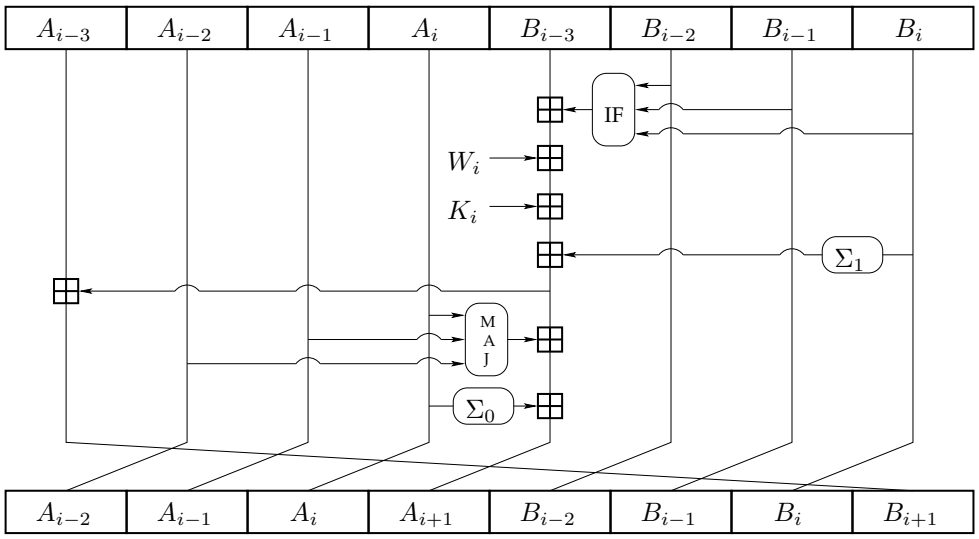

FIG. 4.8 – Une étape de la fonction de compression de SHA-256.

4.9.2 Sécurité actuelle

Pour l'instant, aucune attaque n'a été publiée pour la version complète de SHA-256. Les essais des cryptologues ne portant que sur des versions très réduites [GH03, MPR06, NB08, IMP08, SS08]. Ceci peut s'expliquer par la nette amélioration de sécurité apportée par SHA-256 par rapport à ses ancêtres de la famille SHA. La mise à jour de deux registres par étape permet une bien meilleure diffusion, mais surtout l'expansion de message n'est plus linéaire dans $\mathbb{F}_2$ comme cela était le cas dans SHA-0 ou SHA-1. En effet, l'utilisation d'additions modulaires rend le processus non linéaire dans $\mathbb{F}_2$ et complique grandement le contrôle du message étendu pour un attaquant. Au contraire, la perte de la notion de tour (et de l'emploi de fonctions booléennes différentes à chaque tour) semble faciliter le travail de l'attaquant. Mais cette évolution parait logique puisque l'attention a été portée sur l'intégration des mots de message qui fut largement sous-estimée par le passé, comparée au traitement assez robuste de la variable de chaînage.

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 tumultueux. L'implantation de SHA-256 dans des applications cryptographiques n'est pas à proscrire, mais par mesure de précaution le NIST anticipe de futures avancées en organisant un appel à soumission pour de nouvelles fonctions de hachage, dans le but de standardiser le futur SHA-3 [N-sha3]. Cela permettra à des fonctions de hachage très différentes de MD-SHA de pouvoir peut-être s'imposer comme nouveau standard.

4.10 SHA-512

4.10.1 Description

SHA-512 fut publiée en même temps [N-sha2] que SHA-256 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 nou-

velle architecture. Les autres différences par rapport à SHA-256 concernent la taille de sortie, qui est doublée pour obtenir une fonction viable sur le très 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 [N-sha2] fut 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 :

$$\begin{array}{llll} A_{-3} = h_3 & A_{-2} = h_2 & A_{-1} = h_1 & A_0 = h_0 \\ B_{-3} = h_7 & B_{-2} = h_6 & B_{-1} = h_5 & B_0 = h_4\,. \end{array}$$

À chaque appel, $m = 16$ mots de message sont traités, avec $s = 80$ étapes ($t = 1$ et $u = 80$ dans notre formalisme). L'expansion de message, toujours de type récursif, est similaire à celle de SHA-256 excepté les fonctions σ_0 et σ_1 :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{pour } 16 \leqslant i \leqslant 79 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_0(x) = (x^{\ggg 1}) \oplus (x^{\ggg 8}) \oplus (x^{\gg 7}) \\ \sigma_1(x) = (x^{\ggg 19}) \oplus (x^{\ggg 61}) \oplus (x^{\gg 6}). \end{cases}$$

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 :

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

où les K_i sont des constantes prédéfinies pour chaque étape et les fonctions MAJ et IF sont des fonctions booléennes définies en section A de l'appendice. Les fonctions Σ_0 et Σ_1 sont différentes de celles qui sont utilisées dans SHA-256 et sont définies par :

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

Enfin, du fait du rebouclage, à la fin des 80 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$$\begin{array}{llll} h'_0 = A_{80} + A_0 & h'_1 = A_{79} + A_{-1} & h'_2 = A_{78} + A_{-2} & h'_3 = A_{77} + A_{-3} \\ h'_4 = B_{76} + B_0 & h'_5 = B_{75} + B_{-1} & h'_6 = B_{74} + B_{-2} & h'_7 = B_{73} + B_{-3}\,. \end{array}$$

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'_7 sont retirés).

Une description visuelle d'une étape est donnée dans la figure 4.9 et les caractéristiques complètes des deux versions de la fonction peuvent être trouvées dans [N-sha2].

n	512
m	16
w	64
r	8
t	1
u	80
s	80

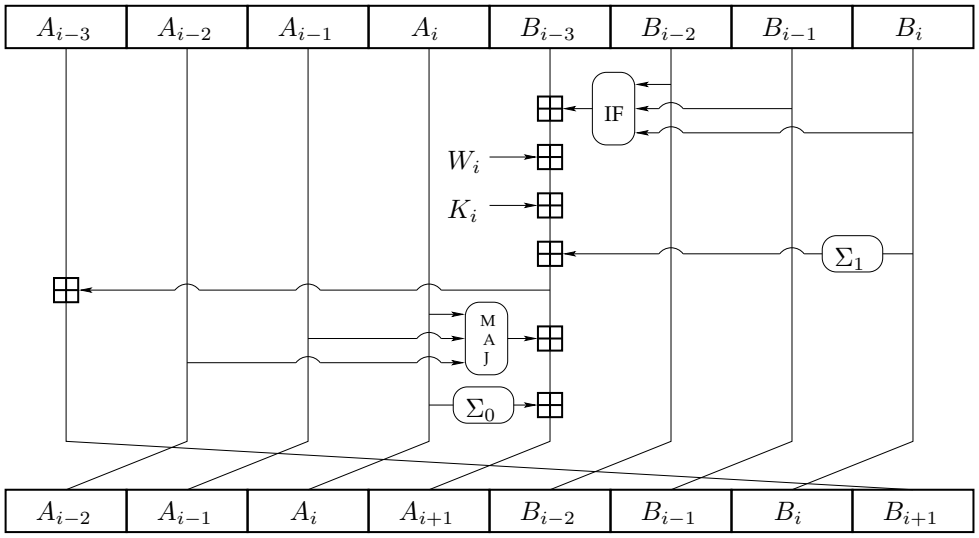

FIG. 4.9 – Une étape de la fonction de compression de SHA-512.

4.10.2 Sécurité actuelle

Les remarques quant à 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.

CHAPITRE 5

Historique de la cryptanalyse des fonctions de la famille SHA

Sommaire

Nous étudions dans ce chapitre les différentes cryptanalyses qui sont apparues contre les fonctions de la famille SHA. Bien que les attaques contre les autres membres de la famille MD-SHA soient très similaires étant donné le noyau de construction commun, nous ne les décrirons pas ici. Nous considérerons des techniques génériques de cryptanalyse, mais aussi et surtout les spécificités de SHA-0 et SHA-1. Sauf indication contraire, nous utilisons donc dans ce chapitre les paramètres de SHA-0 et SHA-1.

5.1 Structure générale de la cryptanalyse d'une fonction de hachage

Tout d'abord, il nous faut expliciter les différentes manières possibles d'aborder la cryptanalyse. Nous cherchons à attaquer une fonction de hachage, ce qui peut être interprété de différentes manières. Ici, nous nous intéresserons exclusivement à des attaques où l'on cherche à mettre en défaut la résistance en collision. Les fonctions de hachage de la famille MD-SHA utilisant le procédé de Merkle-Damgård comme algorithme d'extension de domaine, toute la sécurité repose sur la fonction de compression. Comme expliqué précédemment, il doit ainsi

être impossible pour un attaquant de calculer une pseudo-collision sur la fonction de compression, et plus généralement il doit être impossible pour lui de calculer deux messages qui aboutissent à deux hachés dont la différence est prévue.

Introduisons le principal outil utilisé par les cryptanalystes des fonctions de hachage : le *chemin différentiel* ou *caractéristique différentielle* définit les différences qui seront introduites dans le message et la variable de chaînage et celles attendues dans les registres internes à chaque étape durant la recherche de collisions. Plus généralement, le chemin différentiel peut aussi indiquer les valeurs de certains bits ou fixer certaines conditions entre certains bits en plus des différences. Ainsi, cet outil est la base de travail lorsque l'on souhaite partir d'un masque de différences choisi (un ensemble de conditions imposées sur la différence) dans la variable de chaînage et aboutir à un autre masque de différences choisi en sortie de la fonction. Le rebouclage de la variable de chaînage (feedforward) n'est pour l'instant pas considéré ici, le chemin différentiel ne concernera que le chiffrement par blocs interne.

Il existe plusieurs façons de conceptualiser un chemin différentiel, aucune n'est parfaite. D'abord, les différences considérées sont multiples : pour compliquer la cryptanalyse et éviter de simples attaques algébriques, les opérations de base utilisées dans la famille MD-SHA sont volontairement incompatibles. Par exemple, une différence sur une addition modulaire (dans $\mathbb{Z}_{2^w}$) ne se comportera pas de la même façon qu'une différence binaire (dans $\mathbb{F}_2^w$). Pour clarifier, nous introduisons les types de différences utilisées :

- La différence *binaire* entre deux nombres $X \in \{0,1\}^w$ et $Y \in \{0,1\}^w$ est définie par le vecteur $\Delta^{\oplus}(X,Y) \in \{0,1\}^w$ calculé par $\Delta^{\oplus}(X,Y) = X \oplus Y$.
- La différence *additive* (ou *modulaire*) entre deux nombres $X \in \{0,1\}^w$ et $Y \in \{0,1\}^w$ est définie par $\partial(X,Y) = X - Y$. Nous avons $-2^w < \partial(X,Y) < 2^w$.
- La différence *binaire signée* entre deux nombres $X \in \{0,1\}^w$ et $Y \in \{0,1\}^w$ utilise un ensemble de trois chiffres $\{-1,0,1\}$. La paire (X,Y) a une différence binaire signée $\Delta^{\pm}(X,Y)$ $= (X^0 - Y^0, X^1 - Y^1, \ldots, X^{w-1} - Y^{w-1})$, c'est-à-dire le i-ième chiffre de la différence est le résultat de la soustraction des i-ièmes bits correspondants de X and Y.

Pour représenter un chemin différentiel, nous utilisons la notation de [CR06], explicitée dans le tableau 5.1. Relativement simple et claire pour utiliser la différence binaire ou binaire signée, elle présente néanmoins le désavantage de ne pas prendre en compte les conditions entre deux bits de positions différentes. En fait, cette représentation permet d'aller plus loin qu'une différence binaire signée puisque n'importe quel type de condition est pris en compte pour un couple de bits sur la même position. Ainsi, on note $\bigtriangledown(X,Y)$ une caractéristique différentielle pour un couple de mots de w bits X et Y. Par exemple, si $w = 8$, la différentielle

$$\bigtriangledown(X,Y) = \{(X,Y) \mid X^7 \wedge\ Y^7 = 0,\ X^i = Y^i \text{ pour } 2 \leqslant i \leqslant 5,\ X^1 \neq Y^1,\ X^0 = Y^0 = 0\}$$

sera représentée par $\bigtriangledown(X,Y) =$ `[7 ?---x0]`.

Nous notons $\bigtriangledown X$ la différence attendue sur un mot X et $X^{\bigtriangledown}$ le couple de mots (X, X^*). Un tel couple *satisfait* la différence attendue si nous avons $X^{\bigtriangledown} \in \bigtriangledown X$, c'est-à-dire que la différence entre les mots X et X^* est l'une des différences valides de $\bigtriangledown X$. Une caractéristique différentielle pour une itération de la fonction de compression de SHA-0 ou de SHA-1 sera donc une collection $\bigtriangledown A_{-4}, \ldots, \bigtriangledown A_{80}$ et $\bigtriangledown W_0, \ldots, \bigtriangledown W_{79}$. Une paire de messages est i-valide pour un chemin différentiel (ou satisfait un chemin différentiel jusqu'à l'étape i) si

$$\begin{cases} A_{-4}^{\bigtriangledown} \in \bigtriangledown A_{-4}, \ldots, A_{i+1}^{\bigtriangledown} \in \bigtriangledown A_{i+1} \\ W_0^{\bigtriangledown} \in \bigtriangledown W_0, \ldots, W_{79}^{\bigtriangledown} \in \bigtriangledown W_{79} \end{cases}$$

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓
–	✓	-	-	✓
x	-	✓	✓	-
0	✓	-	-	-
u	-	✓	-	-
n	-	-	✓	-
1	-	-	-	✓
#	-	-	-	-

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
3	✓	✓	-	-
5	✓	-	✓	-
7	✓	✓	✓	-
A	-	✓	-	✓
B	✓	✓	-	✓
C	-	-	✓	✓
D	✓	-	✓	✓
E	-	✓	✓	✓

TAB. 5.1 – Notations utilisées dans [CR06] pour représenter un chemin différentiel : x désigne un bit du premier message et x^* désigne le même bit pour le deuxième message du couple.

et une paire de messages est valide si elle est 79-valide dans le cas de SHA-0 et SHA-1.

Supposons qu'un chemin différentiel nous soit donné. Il est possible d'évaluer l'effort en termes de calculs pour trouver une paire de messages vérifiant ce chemin à l'aide d'un algorithme très simple de type *profondeur en premier*, fixant les mots de message un par un en commençant par M_0 [CR06]. Pour cela, il nous faut calculer le nombre moyen de noeuds visités dans l'arbre de recherche. Commençons par introduire quelques définitions, provenant de l'article original de De Cannière et Rechberger [CR06] :

Définition 1 *Le degré de liberté $L(i)$ d'un chemin différentiel à l'étape i est égal au nombre de façons de choisir $W_i^{\bigtriangledown} \in \bigtriangledown W_i$ sans violer de conditions linéaires imposées par le calcul du message étendu, étant donné des valeurs fixées de $W_j^{\bigtriangledown}$ pour $0 \leqslant j < i$.*

Puisque le message étendu est totalement déterminé par les 16 premiers mots dans le cas de SHA-0 ou SHA-1, nous avons $L(i) = 1$ pour $i \geqslant 16$.

Définition 2 *La probabilité incontrôlée $P_i(i)$ d'un chemin différentiel à l'étape i est la probabilité que la sortie $A_{i+1}^{\bigtriangledown}$ de l'étape i soit valide pour ce chemin sachant que les mots d'entrée de cette étape sont aussi valides pour ce chemin :*

$$P_i(i) = P(A_{i+1}^{\bigtriangledown} \in \bigtriangledown A_{i+1} \mid A_{i-j}^{\bigtriangledown} \in \bigtriangledown A_{i-j} \text{ pour } 0 \leqslant j < 5 \text{ et } W_i^{\bigtriangledown} \in \bigtriangledown W_i).$$

Cette définition nous permet de facilement exprimer le nombre $N_e(i)$ de noeuds visités à chaque étape de la fonction de compression lors de la recherche d'une paire de messages valide. Le nombre moyen de fils d'un noeud à l'étape i est égal à $L(i) \cdot P_i(i)$. Ainsi, lorsque l'on calcule le nombre de noeuds à parcourir en moyenne en remontant les étapes, pour une étape i nous avons $N_e(i) = N_e(i+1) \cdot L^{-1}(i) \cdot P_i^{-1}(i)$. La recherche étant finie une fois arrivée à l'étape $s - 1 = 79$, on obtient alors la relation récursive suivante [‡] :

[‡]Dans l'article original [CR06], les auteurs ont essayé d'introduire une notion supplémentaire (la *probabilité*

$$N_e(i) = \begin{cases} 1 & \text{si } i = s-1, \\ N_e(i+1) \cdot L^{-1}(i) \cdot P_i^{-1}(i) & \text{si } i < s-1, \end{cases}$$

et le nombre total de noeuds parcourus est égal à

$$N_T = \sum_{i=0}^{s-1} N_e(i).$$

Afin de faciliter la compréhension de ces différentes valeurs, nous donnons dans la figure 5.1 et 5.2 des exemples de chemins différentiels peu élaborés pour SHA-0. Pour plus de lisibilité, les probabilités $P_i(i)$, les degrés de liberté $L(i)$ et les nombres de noeuds $N_e(i)$ y sont donnés en base logarithmique 2. Le premier chemin représente une caractéristique très simple où nous n'imposons que la condition suivante : nous souhaitons obtenir une collision (aucune différence dans les 5 derniers registres mis à jour) à partir de la véritable valeur d'initialisation de la première variable de chaînage pour SHA-0 ou SHA-1. On remarque que l'on doit alors parcourir plus de 2^{160} noeuds pour trouver une collision. Dans la deuxième figure, le nombre de noeuds à parcourir est très largement réduit grâce à quelques conditions imposées sur les registres internes et sur les mots de message étendu. La méthode de recherche de ces conditions sera explicitée par la suite. On peut toutefois déjà observer que les degrés de liberté peuvent être utilisés pour vérifier les conditions dans les registres et ainsi potentiellement réduire le coût total de l'attaque. De plus, le nombre de noeuds à visiter est nécessairement croissant de l'étape 79 à l'étape 15 puisqu'aucun degré de liberté n'est alors disponible, et $N_e(16)$ sera en pratique presque toujours le terme maximal parmi les $N_e(i)$ (ainsi $N_T \simeq N_e(16)$). Les probabilités faibles sont donc moins contrariantes sur les 16 premières étapes que sur le reste de la fonction de compression, car les mots de messages sont dans ce cas entièrement contrôlés par l'attaquant.

La condition $N_e(s-1) = 1$ dans notre méthode nous indique que nous calculons le nombre de noeuds à visiter pour trouver une seule paire de messages vérifiant le chemin différentiel. Il est également possible de généraliser ce calcul pour trouver a paires valides distinctes en posant $N_e(s-1) = a$. Ceci nous permet aussi de mieux cerner la signification de $N_e^{-1}(0)$ qui représente le nombre total de paires de messages valides que l'on peut engendrer avec le chemin différentiel considéré. Nous pouvons aussi considérer que cela représente le nombre de degrés de liberté non utilisés pour rechercher une paire de messages.

Il faut noter que la visite d'un noeud ne correspond pas forcément au calcul complet d'une itération de la fonction de compression. Il est donc prématuré de dire qu'étant donné un chemin différentiel, la complexité en temps pour trouver une paire de messages valide pour ce chemin est exactement égale à N_T appels à la fonction de compression. Il faudra au préalable mesurer le coût moyen de parcours d'un noeud, soit de manière théorique, soit de manière expérimentale.

Une fois le chemin différentiel établi, il reste à l'attaquant à rechercher une paire de messages vérifiant ce chemin, ce qui constitue en général l'essentiel de la complexité de l'attaque. Ceci peut être fait par la méthode très simple décrite précédemment, mais de bien meilleures techniques sont possibles. Cette partie d'accélération de recherche est quelquefois imbriquée

contrôlée) pour prendre en compte certains cas particuliers d'arbres de recherche. Leur expression du nombre de noeuds à parcourir dépend à la fois de la probabilité contrôlée et de la probabilité incontrôlée. Cependant, dans les situations que l'on considère, les deux expressions sont équivalentes et aboutissent à la même complexité finale de l'attaque.

i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010110111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	????????????????????????????????	64	0.00	0.00
01 :	????????????????????????????????	????????????????????????????????	64	0.00	0.00
	...	...			
12 :	????????????????????????????????	????????????????????????????????	64	0.00	0.00
13 :	????????????????????????????????	????????????????????????????????	64	0.00	0.00
14 :	????????????????????????????????	????????????????????????????????	64	0.00	32.00
15 :	????????????????????????????????	????????????????????????????????	64	0.00	96.00
16 :	????????????????????????????????	????????????????????????????????	0	0.00	160.00
17 :	????????????????????????????????	????????????????????????????????	0	0.00	160.00
	...	...			
75 :	????????????????????????????????	????????????????????????????????	0	-32.00	160.00
76 :	--------------------------------	????????????????????????????????	0	-32.00	128.00
77 :	--------------------------------	????????????????????????????????	0	-32.00	96.00
78 :	--------------------------------	????????????????????????????????	0	-32.00	64.00
79 :	--------------------------------	????????????????????????????????	0	-32.00	32.00
80 :	--------------------------------				

FIG. 5.1 – Un premier chemin différentiel peu élaboré pour SHA-0.

i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$L(i)$	$P_i(i)$	$N_e(i)$
	...	...			
00 :	01100111010001010010001100000001	--------------------------------	32	0.00	-255.00
01 :	????????????????????????????????	--------------------------------	32	0.00	-223.00
	...	...			
07 :	????????????????????????????????	x----------------------x------	32	0.00	-31.00
08 :	????????????????????????????????	-------------------------------x-	32	0.00	1.00
09 :	????????????????????????????????	x------------------------------	32	0.00	33.00
10 :	????????????????????????????????	x------------------------------	32	0.00	65.00
11 :	????????????????????????????????	x------------------------------	32	-32.00	97.00
12 :	--------------------------------	--------------------------------	32	-32.00	97.00
13 :	--------------------------------	--------------------------------	32	-32.00	97.00
14 :	--------------------------------	-------------------------------x-	32	-32.00	97.00
15 :	-------------------------------x-	----------------------x------	32	-32.00	97.00
16 :	--------------------------------	--------------------------------	0	-2.00	97.00
17 :	-------------------------------x-	x----------------------x----x-	0	-3.00	95.00
18 :	-------------------------------x-	x----------------------x------	0	-4.00	92.00
	...	...			
43 :	--------------------------------	-------------------------------x-	0	-2.00	59.00
44 :	--------------------------------	x------------------------------	0	-1.00	57.00
45 :	--------------------------------	x-------------------------------x-	0	-2.00	56.00
46 :	-------------------------------x-	x----------------------x------	0	-1.00	54.00
47 :	--------------------------------	-------------------------------x-	0	-2.00	53.00
	...	...			
76 :	--------------------------------	--------------------------------	0	0.00	0.00
77 :	--------------------------------	--------------------------------	0	0.00	0.00
78 :	--------------------------------	--------------------------------	0	0.00	0.00
79 :	--------------------------------	--------------------------------	0	0.00	0.00
80 :	--------------------------------				

FIG. 5.2 – Un chemin différentiel plus élaboré pour SHA-0.

dans l'établissement du chemin différentiel et permet très souvent une importante réduction de la complexité de l'attaque. Cependant, la rapidité vient au détriment de la facilité de descrip-

tion de l'attaque et les techniques sont parfois tellement sophistiquées que des erreurs sur des travaux publiés peuvent rester plusieurs mois sans être détectées [LL05, YS05]. Notamment, nous avons découvert plusieurs nouvelles erreurs dans les travaux de Wang *et al.* [WYY05b, WYY05a] durant la rédaction de cette thèse. Nous verrons aussi que l'accélération de recherche a des limites, généralement imposées par la consommation trop importante des degrés de liberté. Ceci montre toute la complexité de l'approche du problème, où l'attaquant se doit parfois de choisir entre un chemin différentiel ayant une relativement faible valeur N_T, mais peu de degrés de liberté (ce qui est souvent la meilleure stratégie), et un autre chemin ayant une valeur N_T plus importante, mais permettant potentiellement une meilleure accélération grâce à un plus grand nombre de degrés de liberté.

Comme nous allons le voir, le cheminement historique des attaques contre les fonctions de hachage de la famille MD-SHA est marqué par des améliorations ponctuelles d'un ou des deux outils de la cryptanalyse d'une fonction de compression : l'établissement d'un chemin différentiel et l'accélération de la recherche de messages. L'amélioration de la compréhension des techniques donne la chance aux cryptanalystes de mieux mesurer la complexité de l'approche générale et l'on peut supposer que l'on verra apparaître dans le futur des méthodes qui permettent de choisir de façon automatisée la meilleure stratégie possible.

5.2 Premières attaques

La première attaque que nous allons détailler fut publiée en 1998 par Chabaud et Joux contre SHA-0 et représente l'une des bases de la cryptanalyse moderne des fonctions de hachage. Relativement simple, cette attaque constitue néanmoins le squelette de toutes les attaques contre la famille SHA décrites ultérieurement dans cette thèse. Cette méthode utilise la différence binaire comme base de raisonnement pour simplifier le schéma, et considère ensuite la différence binaire signée pour améliorer la complexité. Ainsi, une différence, notée `x` dans notre formalisme, a un sens et peut être une différence montante `n` ou descendante `u`. Comme expliqué précédemment, SHA-0 comporte des opérations incompatibles, ce qui complique grandement l'analyse. La première étape de l'attaque est de commencer par simplifier l'analyse de l'algorithme en partant d'un schéma modifié, puis de se rapprocher graduellement du schéma original en corrigeant l'attaque si nécessaire.

5.2.1 Les collisions locales

La différence binaire est très facile à étudier et peut constituer un premier point d'appui pour le cryptanalyste. Ainsi, nous partons d'un schéma entièrement linéaire dans $\mathbb{F}_2^{32}$ en modifiant les composantes non linéaires : les additions modulaires sont transformées en « ou exclusif » bit à bit et les fonctions booléennes MAJ et IF sont remplacées par la fonction XOR, définie en appendice. Ce processus, appelé *linéarisation* de la fonction, aboutit donc à la formule de mise à jour suivante pour l'étape i du tour k :

$$A_{i+1} \quad = \quad (A_i)^{\lll 5} \oplus A_{i-1} \oplus (A_{i-2})^{\ggg 2} \oplus (A_{i-3})^{\ggg 2} \oplus (A_{i-4})^{\ggg 2} \oplus W_i \oplus K_k.$$

La deuxième idée de l'attaque est d'utiliser des *collisions locales*. Si l'on introduit une différence a dans un mot du message, cette perturbation va ensuite s'immiscer dans le prochain registre mis à jour. Si aucune différence n'est introduite dans les prochains mots du message, a va se propager dans les registres mis à jour suivants et créer de nouvelles différences qui vont à

leur tour se propager, etc. Bien évidemment, ce comportement de diffusion des différences doit être rapide dans une fonction de hachage pour éviter des attaques où l'on n'introduirait que très peu de différences. Essayons à présent de compenser cette diffusion rapide en introduisant dès que possible les corrections adéquates dans les mots de message qui suivent l'introduction de a. Supposons qu'aucune différence n'est présente dans les registres $A_{i-4}, \ldots, A_i$ et que l'on introduit une première différence binaire sur le mot W_i au niveau du bit j. À l'étape i, le registre A_{i+1} sera mis à jour et la différence sur W_i^j va s'y propager. Le schéma étant totalement linéarisé (les additions modulaires sont transformées en « ou exclusif » bit à bit), la différence se propagera au bit A_{i+1}^j seulement. À l'étape suivante, la différence sur le bit A_{i+1}^j va, en l'absence de toute autre modification des entrées de cette étape, se propager au prochain registre mis à jour A_{i+2} ; mais si l'on introduit une nouvelle différence sur W_{i+1} au niveau du bit $j+5$ (à cause de la rotation de 5 positions sur A_{i+1}) alors les différences s'annuleront et aucune différence ne sera présente dans le registre A_{i+2}. On continue de la même manière aux étapes suivantes en corrigeant A_{i+3}, A_{i+4}, A_{i+5} et A_{i+6} par W_{i+2}^j, W_{i+3}^{j-2}, W_{i+4}^{j-2} et W_{i+5}^{j-2} respectivement. Une fois le registre A_{i+6} corrigé à l'étape $i+5$, la différence initiale dans le registre A_{i+1} ne s'exprimera plus par la suite. Nous avons donc créé une collision dans l'état interne sur ces quelques étapes en corrigeant exactement la diffusion de la perturbation introduite. Un tel motif est appelé une *collision locale* et est décrit dans la figure 5.3. On peut remarquer que si la perturbation se trouve en position j, les corrections se feront sur les positions $j+5$, j et $j-2$ des mots de message étendu.

i	∇A_i	∇W_i
	...	...
$i-2$	----------	----------
$i-1$	----------	----------
i	----------	----------x----------
$i+1$	----------x----------	----------x----------
$i+2$	----------	----------x----------
$i+3$	----------	----------x--------
$i+4$	----------	----------x--------
$i+5$	----------	----------x--------
$i+6$	----------	----------
$i+7$	----------	----------
	...	...

FIG. 5.3 – Une collision locale à la position de bit $j = 10$.

Créer une collision locale est simple, mais cela ne suffit pas à construire une collision sur l'ensemble de la fonction puisque l'attaquant n'a pas un contrôle total sur les mots de message étendu, contrôle qui pourrait sembler nécessaire pour réaliser les corrections de la perturbation. À cause de l'expansion de message, créer une collision locale à une certaine étape impliquera sûrement l'introduction de différences non contrôlées à d'autres moments du calcul du haché, et ces nouvelles perturbations auront une très faible probabilité d'être elles-mêmes corrigées. Il nous faut donc précisément prévoir l'introduction des perturbations et des corrections dans tout le message étendu, de telle sorte que l'expansion de message entière dessine un motif de collisions locales imbriquées, où chaque perturbation introduite est parfaitement corrigée. Pour cela, on peut utiliser deux propriétés de l'expansion de message de SHA-0 : sa linéarité et le fait qu'il n'y ait aucune rotation dans le calcul des mots de message étendu. Ainsi, les bits des mots de message étendu en position j évoluent indépendamment de ceux en une position $k \neq j$. On

peut donc grandement simplifier l'analyse puisqu'un raisonnement sur une seule position de bit est à présent possible : si nous introduisons les perturbations sur la position j uniquement, et si ces perturbations vérifient l'équation de récurrence d'expansion de message pour SHA-0

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}$$

alors nous aurons un motif de différences sur les mots de message étendu qui représente une succession de perturbations et les corrections appropriées sur les bits en positions $j+5$, j et $j-2$, et le tout vérifiant l'expansion de message. On appelle *masque de perturbation* ou *vecteur de perturbation* le vecteur qui définit les étapes où des perturbations sont introduites (chaque perturbation correspondant à une collision locale entière, avec les corrections adéquates). À partir d'un masque de perturbation, le chemin différentiel correspondant sur les registres internes et sur les mots de message étendu peut être entièrement déduit.

5.2.2 Conditions sur le masque de perturbation

Un vecteur de perturbation vérifie l'expansion de message et par conséquent, nous avons seulement 2^{16} candidats différents (l'expansion de message sur une position de bit est totalement définie par les 16 premières étapes). Néanmoins, les candidats ne seront pas tous valides, car aucune *perturbation tronquée* ne doit être présente. Une perturbation tronquée est une perturbation qui apparaît virtuellement dans une étape antérieure à la première étape de la fonction de compression. Le vecteur de perturbation peut en effet prendre en compte le cas où une perturbation est insérée avant la première étape et appliquer ainsi les corrections nécessaires (dont certaines seront présentes dans les premières étapes de la fonction de compression). Aucune différence n'est pourtant présente dans la variable de chaînage d'entrée et des corrections non voulues introduiront de nouvelles différences non corrigées. Cette situation doit être évitée. Pour cela, on peut calculer à l'envers la formule d'expansion de message pour le vecteur de perturbation et vérifier qu'aucune perturbation n'apparaît dans les 5 premières étapes virtuelles antérieures à la première étape de la fonction de compression. Nous pouvons définir le masque de perturbation comme un vecteur de 85 bits $MP_{-5}, \ldots, MP_{79}$ où MP_i impose l'insertion d'une perturbation à l'étape i et nous avons :

Contrainte 1 *Le vecteur de perturbation ne doit comporter aucune perturbation tronquée :*

$$MP_{-5} = \ldots = MP_{-1} = 0.$$

En plus de cette première contrainte, une condition supplémentaire doit être satisfaite par les masques de perturbation. En effet, nous cherchons une collision à la fin des 80 étapes, ce qui nous impose de n'avoir aucune perturbation à partir de l'étape 75 puisqu'une collision locale entière nécessite en tout 6 étapes. Dans le cas contraire, une telle perturbation ne pourrait être corrigée totalement avant la fin du calcul de la fonction de compression, et au moins une différence serait présente en sortie. Nous avons donc :

Contrainte 2 *Le vecteur de perturbation ne doit comporter aucune perturbation non entièrement corrigée :*

$$MP_{75} = \ldots = MP_{79} = 0.$$

Seulement 128 masques de perturbation sur les 2^{16} initiaux vérifient les deux contraintes et en utilisant le chemin différentiel défini par l'un de ces masques, le calcul d'une collision

est immédiat, car le schéma simplifié que nous considérons est totalement linéaire (chacun des 128 vecteurs conduit à un chemin différentiel avec $N_e(i) = 1$ pour tout i, puisque nous avons $P(i) = 1$ à chaque étape).

5.2.3 Attaque de la vraie fonction de compression de la famille SHA

Nous avons utilisé la différence binaire pour trouver une cryptanalyse d'une version linéarisée de SHA-0. Cependant, la fonction de compression n'est pas $\mathbb{F}_2^{32}$-linéaire et il nous faut à présent réintroduire les éléments réels, tels que l'addition modulaire et les fonctions booléennes MAJ et IF. Dans l'article original de Chabaud et Joux, la réintroduction est très progressive, mais nous étudions ici directement la fonction de compression réelle de SHA-0. La différence binaire signée sera considérée à la place du « ou exclusif », et nous allons voir que plusieurs nouveaux effets sont maintenant à prendre en compte. En fait, nous passons d'une attaque déterministe pour le schéma simplifié à une attaque probabiliste pour le schéma réel : les éléments que nous avions remplacés se comporteront de façon linéaire avec une certaine probabilité.

Examinons tout d'abord les fonctions booléennes. Deux tours sur les quatre utilisent la fonction XOR et ne nécessitent donc pas d'analyse particulière (tout se comporte comme prévu avec probabilité 1). Les fonctions IF et MAJ nécessitent toutefois une analyse plus approfondie. Nous allons distinguer plusieurs cas différents suivant le nombre de différences en entrée de la fonction booléenne sur une position de bit (trois bits en entrée donc huit cas au total), ce nombre étant identique que l'on considère les différences binaires ou celles binaires signées. Il est évident que lorsqu'aucune différence n'est présente en entrée, les fonctions booléennes IF et MAJ se comportent comme la fonction XOR avec probabilité 1 puisqu'aucune différence ne sera créée. Nous pouvons donc exclure ce cas et il nous reste sept situations à étudier :

Situation 1 *Seul le premier bit d'entrée de la fonction booléenne contient une différence.*

Situation 2 *Seul le deuxième bit d'entrée de la fonction booléenne contient une différence.*

Situation 3 *Seuls le premier et le deuxième bit d'entrée de la fonction booléenne contiennent une différence.*

Situation 4 *Seul le troisième bit d'entrée de la fonction booléenne contient une différence.*

Situation 5 *Seuls le premier et le troisième bit d'entrée de la fonction booléenne contiennent une différence.*

Situation 6 *Seuls le deuxième et le troisième bit d'entrée de la fonction booléenne contiennent une différence.*

Situation 7 *Tous les bits d'entrée de la fonction booléenne contiennent une différence.*

Pour toutes ces situations, il nous faut analyser si la propagation des différences se fait correctement (comme dans le cas du « ou exclusif ») pour les fonctions booléennes IF et MAJ. De plus, si une différence doit apparaître en sortie de la fonction booléenne, il faut prendre en compte son signe pour corriger de manière appropriée durant l'addition avec une différence de signe opposé (même pour la fonction booléenne XOR). Nous avons ainsi deux types d'effet à étudier : la propagation et l'addition des différences binaires signées.

On peut se rendre compte qu'il est impossible d'avoir une différence à la fois sur le premier et sur le deuxième bit d'entrée de la fonction booléenne (ou sur le premier et sur le troisième bit), car ils sont situés à des positions différentes et nous avons confiné les perturbations à une seule position de bit. Ainsi, les situations 3, 5 et 7 sont impossibles et seules les situations 1, 2, 4 et 6 sont à considérer [‡]. Par symétrie, les situations 2 et 4 sont équivalentes pour la fonction IF ; les situations 1, 2 et 4 sont équivalentes pour les fonctions MAJ et XOR comme les situations 3, 5, 6. Une table des valeurs permet de se rendre compte que dans les situations 1, 2 et 4 la fonction booléenne MAJ se comporte comme la fonction XOR avec probabilité $1/2$ à chaque étape. La situation 6 se comporte comme un XOR si et seulement si les deux différences ont un sens opposé. Ce qui veut dire que si l'on prend la précaution que ces deux perturbations consécutives insérées aient un sens opposé, alors la fonction booléenne MAJ se comportera comme la fonction XOR avec probabilité 1 dans la situation 6 (et avec probabilité nulle sinon). Une probabilité nulle équivaut à un chemin différentiel impossible et pour éviter cette situation nous devons donc rajouter une nouvelle contrainte. Cette contrainte est néanmoins particulière, car elle concerne le sens des perturbations et non le masque des perturbations en lui-même. Elle traduira simplement le fait que durant les étapes utilisant la fonction booléenne MAJ, nous ne devons pas observer de perturbations consécutives de même sens (situation 6) :

Contrainte sur signes 1 *Le vecteur de perturbation ne doit pas comporter de perturbations consécutives de même sens entre les étapes* 36 *et* 56 *:*

$$(MP_i \wedge MP_{i+1}) \wedge (W_i^j \oplus W_{i+1}^j \oplus 1) = 0 \text{ pour } 36 \leqslant i \leqslant 55.$$

Pour la fonction booléenne IF, comme pour la fonction MAJ, tout se comporte comme une fonction XOR dans les situations 1, 2 et 4 avec une probabilité $1/2$. La situation 6 est plus problématique, car la probabilité d'un comportement linéaire est nulle : une différence sera nécessairement renvoyée dans le cas de la fonction IF alors qu'aucune différence n'est observée en sortie dans le cas de la fonction XOR. Cette situation doit donc être évitée sous peine d'obtenir un chemin différentiel impossible à vérifier. Pour cela, il faut empêcher d'avoir deux perturbations consécutives dans le premier tour, où la fonction IF est utilisée. Plus précisément, il faut éviter d'avoir deux perturbations entre les étapes -4 et 16 puisque le problème se situe trois et quatre étapes après l'introduction de la perturbation. Cela nous rajoute ainsi une troisième contrainte sur le vecteur de perturbation :

Contrainte 3 *Le vecteur de perturbation ne doit pas comporter de perturbations consécutives entre les étapes* -4 *et* 15 *:*

$$MP_i \wedge MP_{i+1} = 0 \text{ pour } -4 \leqslant i \leqslant 15.$$

Considérons à présent la réintroduction de l'addition. Pour qu'une addition modulaire se comporte comme un « ou exclusif », il faut qu'aucune retenue ne soit impliquée. En d'autres termes, il faut qu'une perturbation introduite ne se propage pas sur les bits adjacents à la position initiale par une modification de la valeur de la retenue sortante. Une simple table des valeurs indique que cela se produit avec une probabilité $1/2$. Une condition supplémentaire est qu'une correction corrige exactement le bit visé, sans introduire de nouvelles différences dans les bits adjacents. Ici encore, la différence binaire signée devient utile. Si la perturbation

[‡] Même si les perturbations ne sont pas confinées à une seule position de bit dans le cas de SHA-1, les situations 3, 5 et 7 n'apparaîtront que très rarement en pratique.

que nous avons introduite est une différence montante n (respectivement descendante u) alors nous devons obtenir une différence montante (respectivement descendante) dans le premier registre mis à jour (autrement la retenue a été modifiée et les bits adjacents comporteront aussi une différence), ce qui se produit avec probabilité $1/2$. Les corrections devront se faire dans le sens inverse de celui dans lequel s'exprime la différence concernée. Dans le cas de la première et de la dernière correction, où la différence s'exprime simplement comme un terme de l'addition, il suffira d'utiliser une différence de sens inverse à la perturbation pour corriger avec probabilité 1. Les autres corrections sont plus complexes, car la perturbation s'exprime alors dans la fonction booléenne et peut changer de sens suivant la valeur des autres bits d'entrée de la fonction. Reprenons les situations étudiées précédemment, sachant que l'on ne garde que les cas où les fonctions booléennes se comportent de façon linéaire. Pour la fonction XOR, la correction ne se fera dans le bon sens qu'avec probabilité $1/2$ pour les situations 1, 2 et 4. Dans la situation 6, les différences s'annulant, aucune correction ne sera nécessaire. Pour la fonction MAJ dans les situations 1, 2 et 4, si la différence est bien diffusée de façon linéaire, son sens ne changera pas et l'on peut la corriger avec probabilité 1 en utilisant une différence de sens inverse à la perturbation. La situation 6 est identique au XOR, les différences s'annulant. Pour la fonction IF, nous savons que la situation 6 est évitée. Dans les situations 2 et 4, si la différence est bien diffusée de façon linéaire, son sens ne changera pas et l'on peut ainsi la corriger avec probabilité 1 en utilisant une différence de sens inverse à la perturbation. Dans la situation 1, suivant la valeur des deux derniers bits d'entrée, le sens de la différence peut changer et nous corrigeons alors correctement avec probabilité $1/2$.

Il y a cependant une exception à ce raisonnement concernant l'opération d'addition : si la position du bit considéré est $j = 31$ (le bit de poids fort), alors la différence binaire signée n'a plus de sens puisque la retenue sera de toute façon absorbée par l'application du modulo de l'addition modulaire. À cette position, on peut donc corriger[‡] sans se soucier du sens de la différence, et ceci avec probabilité 1. Il est de ce fait naturel de se servir de cette propriété pour augmenter le plus possible les probabilités de réussite. Si nous introduisons les perturbations sur le bit j, la majorité des corrections se feront sur le bit $j - 2$. On maximisera ainsi la réussite en posant $j - 2 = 31$, ce qui nous donne $j = 33 = 1 \pmod{32}$. Dans le cas de SHA-0, nous utilisons finalement la position $j = 1$ pour introduire les perturbations.

Il existe une autre représentation des phénomènes analysés ci-dessus, car toutes les probabilités considérées reflètent des conditions sur les registres internes ou sur les mots de message étendu. Une collision locale a une certaine probabilité de réussite suivant son étape de démarrage et sa position de bit, et ceci peut être retranscrit en conditions. Nous résumons dans le tableau 5.2 ces conditions et les probabilités de réussite pour chaque étape d'une collision locale dans les situations rencontrées pour SHA-0 (les situations 1, 2, 4 et 6). Dans les tableaux B.1 et B.2 en appendice est donnée une analyse plus complète, comportant toutes les situations. On appelle alors *conditions suffisantes* les conditions sur les bits de registres internes qui, une fois remplies, fournissent à l'attaquant une collision avec probabilité égale à 1. Il devra donc tester 2^b paires de messages, où b représente le nombre de conditions suffisantes, avant d'atteindre son but.

Pour illustrer ce qui précède de façon plus visuelle, la figure 5.4 représente un exemple de collision locale signée suivant la fonction booléenne considérée. Nous notons $C(i)$ le nombre de conditions présentes sur A_i, sachant que toute condition portant sur n bits différents aura un poids $1/n$ pour chacun d'eux. Par exemple, si nous avons la condition $A_i^j \oplus A_{i+1}^j = 0$,

[‡]Même si $j = 31$, la probabilité ne peut être égale à 1 pour l'introduction de la perturbation, car le signe de la différence doit obligatoirement être connu par l'attaquant pour le reste de la collision locale.

étape	type	propagation	addition	probabilité
i	introduction		$A_{i+1}^j = a$ et $W_i^j = a$	$1/2$
$i+1$	correction		$W_{i+1}^{j+5} = \overline{a}$	1
$i+2$ (IF)	corr. situation 1	$A_{i-1}^{j+2} \neq A_i^{j+2}$	$A_{i-1}^{j+2} = W_{i+2}^j \oplus \overline{a}$	$1/4$ ($1/2$ si $j = 31$)
$i+2$ (MAJ)	corr. situation 1	$A_{i-1}^{j+2} \neq A_i^{j+2}$	$W_{i+2}^j = \overline{a}$	$1/2$
$i+2$ (XOR)	corr. situation 1		$A_{i-1}^{j+2} \oplus A_i^{j+2} = W_{i+2}^j \oplus \overline{a}$	$1/2$ (1 si $j = 31$)
$i+3$	corr. situation 2 (-,a,-)	$A_{i+2}^{j-2} = 1$	$W_{i+3}^{j-2} = \overline{a}$	$1/2$
IF	corr. situation 6 (-,a,p_2)			0
$i+3$	corr. situation 2 (-,a,-)	$A_{i+2}^{j-2} \neq A_i^j$	$W_{i+3}^{j-2} = \overline{a}$	$1/2$
MAJ	corr. situation 6 (-,a,p_2)	$p_2 = \overline{a}$		1
$i+3$	corr. situation 2 (-,a,-)		$A_{i+2}^{j-2} \oplus A_i^j = W_{i+3}^{j-2} \oplus \overline{a}$	$1/2$ (1 si $j = 1$)
XOR	corr. situation 6 (-,a,p_2)			1
$i+4$	corr. situation 4 (-,-,a)	$A_{i+3}^{j-2} = 0$	$W_{i+4}^{j-2} = \overline{a}$	$1/2$
IF	corr. situation 6 (-,p_2,a)			0
$i+4$	corr. situation 4 (-,-,a)	$A_{i+3}^{j-2} \neq A_{i+2}^j$	$W_{i+4}^{j-2} = \overline{a}$	$1/2$
MAJ	corr. situation 6 (-,p_2,a)	$p_2 = \overline{a}$		1
$i+4$	corr. situation 4 (-,-,a)		$A_{i+3}^{j-2} \oplus A_{i+2}^j = W_{i+4}^{j-2} \oplus \overline{a}$	$1/2$ (1 si $j = 1$)
XOR	corr. situation 6 (-,p_2,a)			1
$i+5$	correction		$W_{i+5}^{j-2} = \overline{a}$	1

TAB. 5.2 – Conditions à vérifier pour la réussite d'une collision locale pour SHA-0 (ou dans la plupart des cas de SHA-1, voir tableaux B.1 et B.2 en appendice), avec une perturbation introduite à l'étape i et à la position j. Le signe de la perturbation est donné par la contrainte $W_i^j = a$. La première colonne désigne l'étape considérée et la deuxième précise le type d'action appliquée et la situation le cas échéant. La troisième colonne (respectivement la quatrième) donne les conditions pour que la propagation (respectivement l'addition) des différences binaires signées se comporte de façon linéaire. Enfin, la dernière colonne donne la probabilité de réussite.

nous ajoutons $1/2$ à $C(i)$ et $C(i+1)$. Nous donnerons dans la suite pour chaque étape soit les probabilités $P_i(i)$, soit le nombre de conditions $C(i)$ à vérifier (l'analyse étant équivalente).

5.2.4 Rechercher une paire de messages valide

Une fois l'évaluation qualitative d'un vecteur de perturbation introduite, on peut rechercher exhaustivement parmi les 2^{16} candidats et garder celui minimisant le nombre de conditions ou celui minimisant $N_e(i)$ pour une certaine étape i. Tout le problème, à présent, est de choisir le bon i. La partie recherche de vecteurs de perturbation représente en fait une phase de précalcul et le vrai coût de l'attaque est le coût durant la recherche d'une paire de messages valide pour le chemin différentiel qui découle du vecteur de perturbation. Le i à considérer peut varier suivant les méthodes de recherche. Il faut aussi noter que certaines conditions doivent être vérifiées sur le message étendu. En effet, en plus du masque de différences binaires déjà préétabli, quelques bits du message étendu doivent vérifier certaines relations comme l'on peut le constater dans le tableau 5.2. Plusieurs techniques sont possibles pour maximiser la probabilité de succès, comme par exemple fixer de façon appropriée tous les prédécesseurs de ces bits pour

fonction booléenne **IF**				
i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$P_i(i)$	$C(i)$
$i-1$	----------	----------	0.00	1.5
i	----------	----------n----------	-1.00	0.5
$i+1$	----------n----------	----------u----------	0.00	1.0
$i+2$	----------	----------u----------	-2.00	1.0
$i+3$	----------	----------u--------	-1.00	1.0
$i+4$	----------	----------u--------	-1.00	0.0
$i+5$	----------	----------u--------	0.00	0.0
$i+6$	----------	----------	0.00	0.0

fonction booléenne **XOR** ou **MAJ**				
i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$P_i(i)$	$C(i)$
	...	...		
$i-1$	----------	----------	0.00	0.5
i	----------	----------n----------	-1.00	1.0
$i+1$	----------n----------	----------u----------	0.00	1.0
$i+2$	----------	----------u----------	-1.00	1.0
$i+3$	----------	----------u--------	-1.00	0.5
$i+4$	----------	----------u--------	-1.00	0.0
$i+5$	----------	----------u--------	0.00	0.0
$i+6$	----------	----------	0.00	0.0
	...	...		

FIG. 5.4 – Une collision locale signée à la position de bit $j = 10$, suivant la fonction booléenne utilisée. L'avant-dernière colonne représente les probabilités de réussite à chaque étape (en base logarithmique 2) et la dernière donne le nombre de conditions.

être certain que les paires de messages testées vérifieront ces conditions. Cela consommera des degrés de liberté sur les messages, mais la recherche sera simplifiée et accélérée, car le problème sera alors définitivement réglé. On peut sinon exprimer les contraintes sur les mots de message non étendu pour ensuite ne tester que ceux qui appartiennent au sous-espace vérifiant les conditions durant la recherche d'une paire valide.

Une première méthode de recherche très naïve serait de tirer aléatoirement une paire de messages qui satisfait le masque de différence pour le vecteur W et d'espérer que le chemin différentiel soit aussi vérifié en ce qui concerne les registres. Toutes les conditions entre la première et la dernière étape devraient alors être pris en compte (le i à considérer serait égal à 0) et le coût total serait élevé. Cette méthode n'utilise aucun des degrés de liberté qui nous sont offerts par les mots de message.

Une méthode un peu plus sophistiquée, déjà explicitée précédemment, serait de voir la recherche comme un algorithme de type profondeur en premier en fixant tous les mots de message un par un et en commençant par M_0. Pour cette seconde méthode, le nombre total de noeuds visités N_T est égal à la somme pour chaque étape k des nombres moyens de noeuds visités $N_e(k)$. Nous avons déjà expliqué qu'en général $N_T \simeq N_e(16)$, car l'attaquant peut contrôler complètement les 16 premiers mots de message durant les 16 premières étapes. Dans ce cas présent, la valeur de i à considérer est égale à 16. Cette méthode utilise donc légèrement les degrés de liberté offerts à l'attaquant, en fixant simplement les mots de message des 16 premières étapes de façon indépendante. Nous considérerons par la suite que cette méthode représente la base de comparaison pour une recherche de paires de messages valides sans aucune accélération, et nous la nommerons *méthode de référence*. Notons que pour obtenir la complexité exacte

en termes de nombre d'appels à la fonction de compression, il nous faudrait évaluer le coût du parcours des noeuds de l'arbre.

Dans l'article de Chabaud et Joux [CJ98], une petite amélioration dans la recherche de messages est introduite. Il est remarqué que les conditions sur les registres A_{16} et A_{17} peuvent être satisfaites directement en choisissant avec précaution les instances du message. Cette amélioration utilise encore un peu plus les degrés de liberté disponibles, et permet finalement d'obtenir une complexité de 2^{61} conditions pour trouver une collision sur SHA-0, au lieu de 2^{69} pour la méthode de référence. Le vecteur de perturbation correspondant est donné en figure 5.5. On peut voir que les contraintes 1, 2 et 3 sont vérifiées pour ce vecteur. Nous donnons aussi un chemin différentiel correspondant dans la figure 5.6. Les mots du registre interne A_i pour $i = -4, \ldots, 0$ sont fixées aux valeurs d'initialisation spécifiées pour SHA. Dans cet exemple, les perturbations sont bien toutes placées sur la position de bit 1 et sont signées. Toute autre instance de ces signes est valide et équiprobable tant que toutes les contraintes sont vérifiées. Enfin, on remarque que les corrections du message étendu sur la position de bit 31 ne sont pas signées, car les signes des différences sont complètement sans effet quant à la validité du chemin différentiel (aucune retenue ne se propage).

```
00000  00100010000000101111 01100011100000010100
       01000100100100111011 00110000111110000000
```

FIG. 5.5 – Vecteur de perturbation utilisé pour l'attaque en collision contre SHA-0 par Chabaud et Joux [CJ98]. Le vecteur se lit de la gauche vers la droite, les 5 premiers bits représentant les perturbations virtuelles sur la variable de chaînage en entrée.

5.2.5 Les bits neutres

Il faudra attendre six ans avant de voir une première amélioration de l'attaque de Chabaud et Joux sur SHA-0. Cette amélioration porte sur une meilleure utilisation des degrés de liberté et non sur la construction du vecteur de perturbation. Cette technique, appelée *bits neutres* et introduite pas Biham et Chen [BC04], est la première méthode avancée d'accélération de recherche d'une paire de messages valide pour la famille SHA.

L'idée essentielle est qu'étant donné une paire de messages i-valide trouvée (l'ordre de grandeur de i en pratique étant entre 19 et 22), il existe des bits du message non étendu que l'on peut modifier simultanément sans changer la validité de la paire jusqu'à cette étape i. Chaque modification de ce type est appelée *bit neutre* et peut donc concerner plusieurs bits de messages à la fois (un bit neutre peut en fait être composé de la modification de plusieurs bits de message). À partir de la paire de messages i-valide considérée, on cherche alors le plus grand ensemble possible E_{bn} de bits neutres, tel que toute modification composée d'une paire de E_{bn} soit aussi un bit neutre. Cette recherche d'ensemble peut être conduite de manière assez rapide. Si l'on note N_{bn} le nombre d'éléments de E_{bn}, il a été mesuré expérimentalement que $1/8$ des $2^{N_{bn}}$ combinaisons possibles de cet ensemble (soit $2^{N_{bn}-3}$) sont des bits neutres pour l'étape i. À partir d'un message i-valide original on peut donc générer approximativement $2^{N_{bn}-3}$ nouveaux messages i-valides en appliquant les modifications correspondantes, et ce, sans recommencer la recherche entre les étapes 16 et i.

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010110111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	----------1----1-	30	-1.00	-398.00
01 :	----------1---	----------1----0-	30	-1.00	-369.00
02 :	----------0---	----------0----n-	30	-1.00	-340.00
03 :	----------n-	----------u----1-	30	-1.00	-311.00
04 :	1----------	----------1----n-	30	-2.00	-282.00
05 :	0----------1---	x---------1----0-	30	-1.00	-254.00
06 :	----------0---	x---------0----n-	30	-1.00	-225.00
07 :	----------n-	x---------u----0-	30	-1.00	-196.00
08 :	1----------	----------0----n-	30	-1.00	-167.00
09 :	0----------	x---------1----0-	30	0.00	-138.00
10 :	----------	x---------0----1-	30	0.00	-108.00
11 :	----------	x---------1----1-	30	0.00	-78.00
12 :	----------	----------1----0-	30	-1.00	-48.00
13 :	----------0---	----------0----1-	30	-1.00	-19.00
14 :	----------1---	----------1----u-	30	-2.00	10.00
15 :	----------0-u-	----------n----1-	30	-2.00	38.00
16 :	1----------1---	----------1----0-	0	-3.00	66.00
17 :	0----------0-u-	x---------n----n-	0	-3.00	63.00
18 :	1----------0-n-	x---------u----0-	0	-2.00	60.00
19 :	----------1-u-	----------n----0-	0	-1.00	58.00
20 :	----------u-	----------n----n-	0	0.00	57.00
21 :	----------	x---------1----1-	0	-1.00	57.00
22 :	----------n-	x---------u----u-	0	-1.00	56.00
23 :	----------u-	----------n----u-	0	-1.00	55.00
24 :	----------	----------0----n-	0	-1.00	54.00
25 :	----------	----------1----0-	0	0.00	53.00
26 :	----------	----------0----n-	0	-1.00	53.00
27 :	----------n-	x---------u----n-	0	-1.00	52.00
28 :	----------n-	----------u----1-	0	-2.00	51.00
29 :	----------n-	x---------u----u-	0	-1.00	49.00
30 :	----------	----------0----n-	0	-1.00	48.00
31 :	----------	x---------1----1-	0	0.00	47.00
32 :	----------	----------1----1-	0	0.00	47.00
33 :	----------	x---------1----0-	0	0.00	47.00
34 :	----------	----------0----1-	0	0.00	47.00
35 :	----------	----------1----n-	0	-1.00	47.00
36 :	----------n-	----------u----0-	0	0.00	46.00
37 :	----------	----------0----0-	0	-2.00	46.00
38 :	----------u-	x---------n----1-	0	0.00	44.00
39 :	----------	x---------1----n-	0	-1.00	44.00
40 :	----------	----------1----1-	0	-1.00	43.00
41 :	----------	x---------1----u-	0	-2.00	42.00
42 :	----------u-	x---------n----0-	0	0.00	40.00
43 :	----------	----------0----n-	0	-1.00	40.00
44 :	----------	x---------1----0-	0	-1.00	39.00
45 :	----------	x---------0----n-	0	-2.00	38.00
46 :	----------n-	x---------u----0-	0	0.00	36.00
47 :	----------	----------0----u-	0	-1.00	36.00
48 :	----------	x---------0----u-	0	-2.00	35.00
49 :	----------u-	x---------n----1-	0	-1.00	33.00
50 :	----------	x---------1----n-	0	-1.00	32.00
51 :	----------	x---------1----u-	0	-2.00	31.00
52 :	----------u-	x---------n----0-	0	-1.00	29.00
53 :	----------	x---------0----n-	0	-1.00	28.00
54 :	----------	x---------1----u-	0	-2.00	27.00
55 :	----------u-	x---------n----n-	0	-2.00	25.00
56 :	----------n-	x---------u----0-	0	-2.00	23.00
57 :	----------u-	x---------n----u-	0	-2.00	21.00
58 :	----------	----------0----0-	0	-2.00	19.00
59 :	----------u-	x---------n----u-	0	-1.00	17.00
60 :	----------u-	----------n----u-	0	-1.00	16.00
61 :	----------	----------0----u-	0	-1.00	15.00
62 :	----------	----------0----u-	0	-1.00	14.00
63 :	----------u-	----------n----u-	0	-1.00	13.00
64 :	----------u-	x---------n----n-	0	-1.00	12.00
65 :	----------	x---------1----n-	0	-1.00	11.00
66 :	----------	----------1----1-	0	0.00	10.00
67 :	----------	----------1----0-	0	0.00	10.00
68 :	----------	x---------0----n-	0	-1.00	10.00
69 :	----------n-	----------u----n-	0	-1.00	9.00
70 :	----------n-	----------u----0-	0	-2.00	8.00
71 :	----------u-	x---------n----0-	0	-2.00	6.00
72 :	----------u-	----------n----0-	0	-2.00	4.00
73 :	----------u-	x---------n----n-	0	-1.00	2.00
74 :	----------	x---------1----n-	0	-1.00	1.00
75 :	----------	x---------1----0-	0	0.00	0.00
76 :	----------	----------0----0-	0	0.00	0.00
77 :	----------	x---------0----0-	0	0.00	0.00
78 :	----------	----------0----1-	0	0.00	0.00
79 :	----------	----------1----1-	0	0.00	0.00
80 :	----------				

FIG. 5.6 – Chemin différentiel utilisé pour l'attaque en collision contre SHA-0 par Chabaud et Joux [CJ98].

Si l'on arrive à détecter un ensemble assez grand (et puisque la recherche de E_{bn} nécessite un temps de calcul négligeable), le coût pour trouver la paire de messages i-valide originale sera amorti et l'on pourra dans ce cas considérer que la recherche d'une collision commence à l'étape i. Plus exactement, il faut que la taille de l'ensemble des bits neutres valides soit au moins aussi grande que le nombre de conditions à satisfaire entre l'étape i et 80, soit :

$$2^{N_{bn}-3} \geqslant \prod_{k=i}^{79} P_i^{-1}(k).$$

Cette inégalité doit être vérifiée si l'on souhaite que l'attaque fonctionne à partir d'une seule paire de messages i-valide. On peut, si nécessaire, utiliser plusieurs paires de message i-valides pour relâcher cette inégalité. Néanmoins, après application de la technique des bits neutres, on doit obtenir un nombre suffisant de paires i-valides pour espérer trouver une collision. On voit que le choix de l'étape i à considérer est crucial, car si i est trop petit il sera très facile de trouver un ensemble E_{bn} très grand, mais le gain final ne sera pas suffisamment important. Inversement, choisir un i trop grand ne permettra pas de détecter un ensemble E_{bn} contenant assez d'éléments pour trouver une collision parmi les messages obtenus grâce aux bits neutres. Même si ceci dépend du message i-valide original, on peut vérifier expérimentalement que le choix $i = 20$ semble proche de l'optimal pour SHA-0.

Une limitation qui ne fut pas mentionnée dans l'article original, car peu importante dans le cas de SHA-0, existe cependant. Comme précisé précédemment, durant la recherche d'un message i-valide, il faut tenir compte de contraintes sur le message étendu. L'application des modifications d'un bit neutre peut alors invalider ces contraintes pourtant vérifiées pour le message i-valide original. Ainsi, l'attaquant se retrouve réduit dans son espace de recherche de bits neutres : seuls peuvent être considérés les bits du message non étendu qui ne sont pas prédécesseur d'un bit du message étendu contraint (il est certes possible, mais très rare, qu'un bit neutre modifie plusieurs bits contraints du message non étendu, de telle sorte que les conditions sur le message étendu soient toujours vérifiées). Ce problème n'est pas très contrariant dans le cas de SHA-0 : puisque les perturbations ne seront présentes qu'exclusivement sur la position de bit 1, toutes les contraintes sur le message étendu seront situées sur les positions de bit 1, 6 ou 31. Aucune rotation n'étant utilisée dans l'expansion de message, toutes les contraintes sur le message non étendu seront aussi situées sur les positions de bit 1, 6 ou 31, et l'espace de recherche de bits neutres n'en sera que très peu diminué. Par contre, comme nous le verrons plus tard dans le cas de SHA-1, les contraintes seront situées sur des positions de bits plus variées, mais surtout la rotation dans l'expansion de message influera sur beaucoup plus de bits du message non étendu. Ceci explique pourquoi la technique des bits neutres est assez inefficace pour SHA-1.

Finalement, grâce à cette nouvelle méthode, la complexité pour trouver une collision sur SHA-0 fut ramenée à 2^{56} appels à la fonction de compression.

5.2.6 L'attaque multiblocs

L'amélioration suivante [BCJ05] des cryptanalyses de SHA-0 concerna l'établissement de vecteurs de perturbation et non la recherche de paire de messages valide pour un vecteur donné. L'idée est d'essayer de relâcher certaines des contraintes sur le vecteur de perturbation, et notamment les contraintes 1 et 2 stipulant qu'aucune différence ne doit être présente dans les variables de chaînage d'entrée et de sortie. Ces contraintes ont un sens si l'on recherche

une collision en n'utilisant qu'une seule itération de la fonction de compression, mais on peut imaginer une collision pour laquelle des différences sur les blocs de message sont introduites dans plusieurs itérations consécutives. L'idée de cette méthode, appelée *technique multiblocs*, est née de l'observation qu'il est plus facile de trouver une presque collision sur SHA-0 (quelques bits peuvent être différents sur la variable de chaînage de sortie) qu'une collision réelle.

Comment alors obtenir une collision à l'aide de presque collisions successives ? Le rôle du rebouclage étant ici important, nous considérerons les différences comme étant des différences modulaires : ∂_e^i (respectivement ∂_s^i) représente la différence modulaire en entrée (respectivement sortie) du chiffrement par blocs interne durant l'itération i, et ∂_M^i la différence modulaire sur le bloc de message correspondant. Supposons que pour la première itération où l'on insère des différences dans le message, on puisse trouver un vecteur de perturbation sans différence sur la variable de chaînage d'entrée (vérifiant de ce fait la contrainte 1 : $\partial_e^1 = 0$) et aboutissant à une presque collision sur la variable de chaînage de sortie ($\partial_s^1 \neq 0$). La nouvelle variable d'entrée pour la seconde itération comportera donc quelques différences ($\partial_e^2 = \partial_e^1 + \partial_s^1 = \partial_s^1$) et l'on pourra essayer de chercher un autre vecteur de perturbation qui commence par ces différences et qui débouche sur une autre presque collision ($\partial_s^2 \neq 0$). Il ne faut cependant pas oublier le rebouclage qui réinjecte en sortie les différences d'entrée ($\partial_e^{i+1} = \partial_e^i + \partial_s^i$). On peut continuer, de la même manière, sur plusieurs blocs de message jusqu'à aboutir à une ultime itération k pour laquelle les différences d'entrée annuleront parfaitement celles de sortie lors du rebouclage ($\partial_s^k = -\partial_e^k$). Il faut noter que puisque les différences en entrée et en sortie d'un vecteur de perturbation peuvent être signées comme le souhaite l'attaquant, la dernière itération k fera en fait correspondre la même différence binaire non signée en entrée que celle en sortie (il suffira de forcer des signes inverses pour obtenir une collision après le rebouclage). À la fin du processus, nous obtenons une collision en ayant inséré des différences sur les blocs de message durant plusieurs itérations consécutives (voir figure 5.7). La complexité totale sera alors égale à la somme pour chaque itération des complexités pour trouver une paire de messages valide.

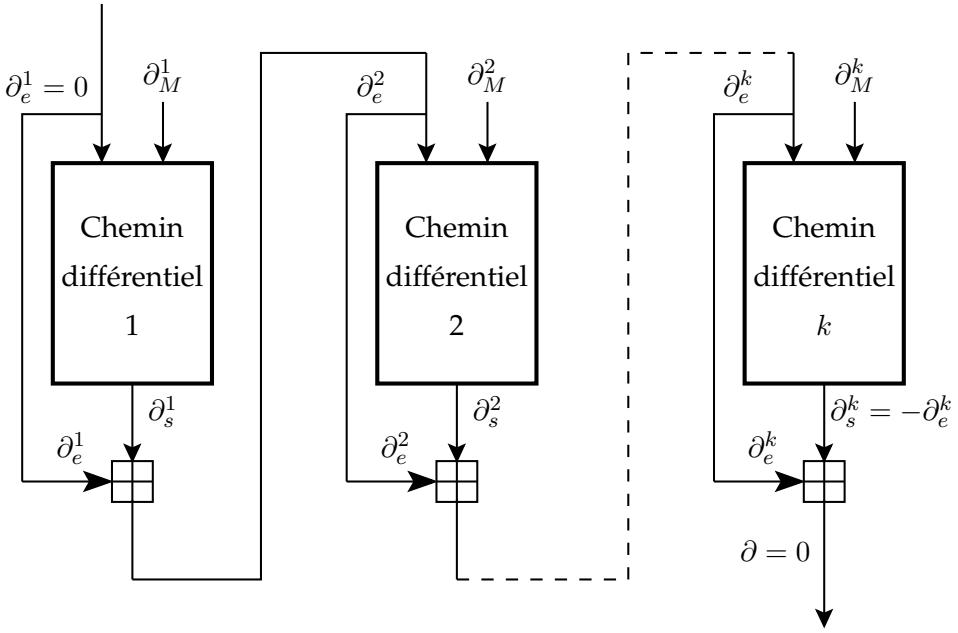

FIG. 5.7 – Principe de l'attaque multiblocs.

Pour planifier l'attaque et prévoir quelles presque collisions nous allons tenter de parcourir, on peut construire au préalable un graphe orienté dont les noeuds représentent les différences dans la variable de chaînage et les arêtes sont des vecteurs de perturbation qui permettent de joindre un masque de différence sur la variable de chaînage d'entrée à un masque sur celle de sortie (toujours sans oublier le rebouclage). Il faut noter que puisque les perturbations sont toujours situées sur la position de bit 1, le nombre de masques possibles (le nombre de sommets) est égal à $2^5 = 32$. On associe ainsi à chaque arête un poids qui correspond au coût estimé de la recherche d'une paire de messages valides pour le vecteur de perturbation correspondant. Une fois le graphe établi grâce à des vecteurs de perturbation dont le coût n'est pas trop grand, on recherche le chemin de poids le plus faible partant du sommet « sans différence » et arrivant à ce même sommet. Le chemin trouvé définira exactement les vecteurs de perturbation à utiliser successivement.

Cependant, dans le cas de SHA-0, le chemin de poids le plus faible dans ce graphe est le vecteur de perturbation déjà trouvé par Chabaud et Joux et ne nécessitant qu'une seule itération. Une autre amélioration est alors nécessaire. On peut augmenter la connectivité du graphe en analysant plus finement le comportement de la fonction booléenne IF utilisée dans les premières étapes de SHA-0. Nous avons supposé précédemment que cette fonction se comportait comme la fonction XOR avec une certaine probabilité et ceci nous a permis de construire des vecteurs de perturbation. Néanmoins, on peut aussi prendre en compte les situations ou cette approximation n'est pas vérifiée, spécialement pour les 5 premières étapes (pour lesquelles s'expriment les différences sur la variable de chaînage d'entrée). Concrètement, une différence d'entrée a pourra se comporter d'une façon autre que celle prévue initialement. Ce comportement sera certaines fois identique à celui d'une autre différence d'entrée b se comportant elle de façon linéaire pour aboutir à la différence de sortie c. Ainsi, en plus de l'arête orientée $b \longmapsto c$, nous avons à présent l'arête $a \longmapsto c$ pour un coût égal.

Grâce à cette augmentation de la connectivité du graphe, il est à présent possible de trouver une attaque plus rapide que celle de Chabaud et Joux. Les vecteurs de perturbation de cette attaque, nécessitant 4 itérations où l'on introduit des différences sur les blocs de message, sont explicités dans la figure 5.8. Pour tous les blocs, on peut observer que la contrainte 2 n'est plus vérifiée ($\partial_s^i \neq 0$). De même pour la contrainte 1 ($\partial_e^i \neq 0$), sauf pour le premier bloc, car aucune différence n'apparaît dans la valeur d'initialisation ($\partial_e^1 \neq 0$). Enfin, la contrainte 3 est toujours présente. La complexité finale est égale à la somme des complexités pour trouver des paires de blocs de message valides pour chaque itération, soit 2^{49} appels à la fonction de compression. Cette évaluation tient compte de la technique des bits neutres qui peut être utilisée dans la recherche de messages pour chaque itération. Ce travail a conduit au calcul de la première collision pour SHA-0.

L'amélioration multiblocs seule ne permet toujours pas d'attaquer SHA-1, car la rotation dans l'expansion de message force les perturbations à se situer sur des positions de bit différentes entre elles. Dans ce cas, le nombre de sommets dans le graphe augmente fortement et il est difficile de construire un graphe assez connexe pour trouver un chemin de poids faible.

5.3 Attaques de Wang *et al.*

À partir de 2004, une équipe de recherche chinoise publia de nombreuses améliorations aux attaques déjà connues sur les fonctions de hachage de la famille MD-SHA [WYY05d, WY05, WYY05c, WYY05b, WYY05a, WLF05, WFL04, YWY06]. Deux grandes nouveautés furent intro-

	Vecteur de perturbation		
Bloc 1	00000 (00000)	00010000101001000111	10010110000011100000
		00000011000000110110	00000110001011011000
Bloc 2	01000 (11000)	10000000010000101001	00011110010110000011
		10000000000011000000	11011000000110001011
Bloc 3	10001 (10011)	00100101000100101111	11000010000100001100
		00101100100000000001	11010011101000010001
Bloc 4	11010 (00010)	00100000000100001010	01000111100101100000
		11100000000000110000	00110110000001100010

FIG. 5.8 – Vecteurs de perturbation pour l'attaque multiblocs sur SHA-0 [BCJ05]. Chaque vecteur se lit de la gauche vers la droite, les 5 premiers bits représentant les perturbations virtuelles sur la variable de chaînage en entrée. Les bits entre parenthèses représentent les vraies perturbations attendues en entrée.

duites, l'une concernant l'établissement de vecteurs de perturbation et l'autre permettant une meilleure recherche d'une paire de messages valide.

5.3.1 La modification de message

Cette méthode permet d'accélérer la recherche d'une paire de messages valide pour un vecteur de perturbation donné. Cette avancée étant le fruit d'un travail indépendant des précédents résultats sur la famille SHA, la technique elle-même de recherche d'une paire de messages valide sera différente.

On peut distinguer deux types : la modification de message simple et la modification de message avancée. L'idée générale d'une modification de message consiste à modifier minutieusement certains bits du message de telle sorte qu'une condition sur les registres internes auparavant non vérifiée devienne valide, tout en évitant de perturber celles déjà remplies pour les étapes précédentes. Ces modifications sont établies juste après avoir trouvé le chemin différentiel et utilisées plus tard si nécessaire durant la recherche d'une paire de messages valide.

Plus exactement, une fois le vecteur de perturbation choisi et le chemin différentiel établi, les conditions sur le message non étendu pourront être déduites. On se place ainsi dans le sous-espace des messages qui vérifient ces contraintes et l'on choisit un membre au hasard dans cet ensemble. On calcule alors étape par étape les valeurs des registres internes pour ce candidat. À chaque rencontre de contrainte non vérifiée sur ces registres, on applique la modification de message correspondante pour forcer la validité et l'on continue à la condition suivante. Avec une probabilité relativement forte, aucune condition précédemment forcée ne sera invalidée par l'application de la modification. Il faut noter que comme pour les bits neutres, on doit faire attention à toujours vérifier les conditions sur le message lorsque l'on effectue une modification.

La modification de message simple concerne seulement les conditions qui appartiennent aux étapes 0 à 15, pour lesquelles l'attaquant possède un contrôle total, car l'expansion de mes-

sage n'est pas encore utilisée. En général, ces modifications ne se composent que d'un seul bit à changer, directement relié au bit du registre interne à éventuellement corriger. Par exemple, si une condition n'est pas vérifiée sur le bit A_i^j, on pourra modifier le bit W_i^j (sous réserve qu'il ne soit pas impliqué dans une contrainte sur le message étendu). Cette technique remplace en fait notre précédente recherche de messages sous forme de parcours d'arbre. Pour cette partie de l'attaque, les conditions peuvent de toute façon être fixées de manière relativement indépendante. Ceci implique que le coût total de l'attaque n'est pas corrélé à cette première phase et donc qu'aucune amélioration ne découlera des modifications de message simples (par analogie, le nombre de noeuds à parcourir dans l'arbre restera constant).

La modification de message avancée permet de satisfaire des conditions postérieures à l'étape 15 et induit ainsi une amélioration de la complexité totale. Approximativement, il est possible de corriger les conditions non valides jusqu'à l'étape 25 dans le cas de SHA. Le reste des conditions seront satisfaites de manière probabiliste, en essayant de nombreuses instances de message. Les modifications de message avancées sont beaucoup plus complexes et nécessitent en général de changer plusieurs bits à la fois. La méthode est aussi moins directe que celle des modifications de message simples. Par exemple, on pourra modifier un bit W_i^j du message, ce qui induira une modification de A_{i+1}^j (on suppose que la retenue ne se propage pas). Cette modification dans le registre va se diffuser dans les mots suivants : dans A_{i+2}^{j+5}, puis dans A_{i+3}^{j-2}, puis dans A_{i+4}^j et A_{i+4}^{j+10} etc. et d'autres modifications seront aussi créées à partir de W_i^j à cause de l'expansion de message. Ces nouvelles modifications vont à leur tour elles-mêmes créer d'autres modifications et ainsi de suite. On espère donc corriger précisément une position de bit du registre interne à des étapes relativement avancées, sans pour autant invalider les précédentes conditions remplies à d'autres positions. En pratique, les retenues se propageant certaines fois, il y aura une probabilité d'échec associée à toute modification. Pour cette raison, l'ordre d'application des modifications de message sera en fait très important. Il faut aussi rappeler que l'attaquant n'est pas complètement libre de modifier les bits de message à cause des contraintes dues au chemin différentiel.

Une fois le chemin différentiel établi, l'attaquant peut engendrer l'ensemble des modifications de message possibles en linéarisant le schéma (comme lors de la construction du vecteur de perturbation). En identifiant les positions de bit où des conditions sont présentes, il peut choisir les meilleures modifications de message en tenant compte de leur probabilité de succès pour le schéma réel (non linéarisé) et en sélectionnant l'ordre d'application optimal. Trouver des modifications de message dépend aussi fortement du vecteur de perturbation considéré. Dans les articles originaux sur SHA, très peu de détails furent donnés sur ce sujet et il est supposé que toutes les modifications de message furent établies *à la main*. Il était de plus impossible de réellement évaluer la complexité du processus dans le cas de SHA-0 ou de SHA-1 étant donné le flou concernant l'implantation des modifications de message. Le cas de MD concernant la modification de message est finalement assez différent de celui de SHA, car l'expansion de message est ici un élément très important à considérer.

On peut enfin remarquer la dualité entre les bits neutres et les modifications de messages : les premiers tentent de multiplier les instances de message valides tandis que les seconds corrigent les mauvaises instances. Dans les deux cas, ces techniques représentent une meilleure utilisation des degrés de liberté durant l'attaque.

5.3.2 La partie non linéaire

La deuxième avancée majeure concerne l'établissement du vecteur de perturbation. La technique multiblocs a déjà permis de s'affranchir des contraintes 1 et 2 sous réserve que l'on réussisse à trouver un chemin de poids faible dans le graphe orienté des connexions possibles entre les variables de chaînage. Nous allons ici essayer de nous débarrasser de la contrainte 3 qui nous impose de ne pas observer deux collisions locales consécutives durant le premier tour (à cause de la fonction booléenne IF qui ne se comporte jamais de façon linéaire dans ce cas).

Voir SHA-1 de façon linéarisée permet de grandement simplifier l'analyse, le problème étant que le retour à la fonction réelle implique de fortes contraintes sur le vecteur de perturbation (contrainte 3). On peut alors essayer de laisser l'évolution des différences se comporter de façon non linéaire. En général, plus le schéma se comportera de façon non linéaire et plus les probabilités de succès seront faibles. Cependant, nous avons déjà vu que les probabilités pour les 16 premières étapes n'influent que très peu sur la complexité totale de l'attaque, car nous avons un contrôle absolu sur le message et les conditions peuvent être vérifiées de façon relativement indépendante. Il est donc naturel d'essayer de permettre un comportement non linéaire à faible probabilité de succès durant ces premières étapes en laissant par exemple se propager les différences sur les retenues lors des additions ou en considérant la fonction booléenne IF réelle.

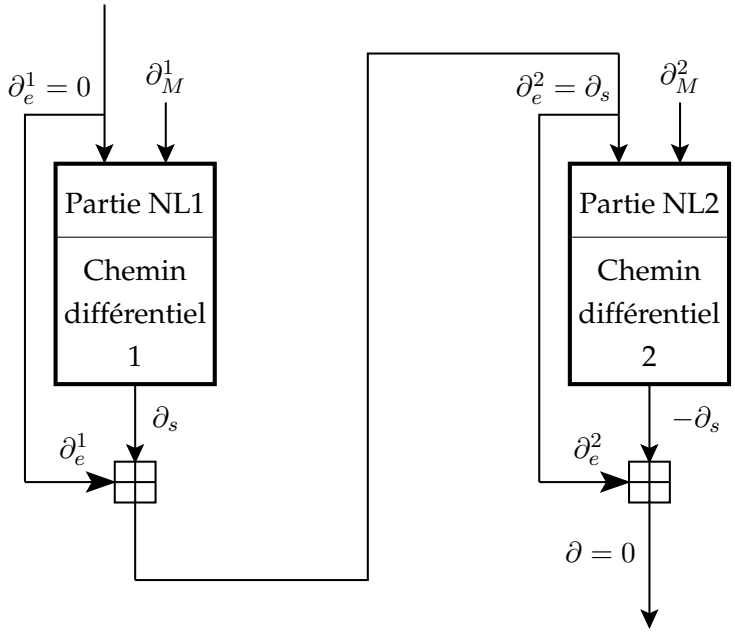

FIG. 5.9 – Attaque multiblocs avec partie non linéaire pour les premières étapes.

Cette technique est en fait très similaire (bien que plus générale et beaucoup plus complexe à mettre en oeuvre) à celle utilisée par Biham *et al.* [BCJ05] pour construire le graphe orienté des connexions possibles entre les variables de chaînage, où l'on considérait que la fonction booléenne IF dans les 5 premières étapes peut se comporter de façon non linéaire. L'effet sera identique : la connectivité du graphe sera augmentée. De plus, la complexité totale étant relativement indépendante de cette partie non linéaire, le poids de l'arête ne dépend que du vecteur

de perturbation entre les étapes 16 et 80. Théoriquement, si l'on arrive toujours à trouver une partie non linéaire quelles que soient les différences présentes sur la variable de chaînage d'entrée, le graphe devient même complet (tous les sommets sont reliés deux à deux par une arête). Autrement dit, on peut relier n'importe quel masque de différences sur la variable de chaînage d'entrée à n'importe quel masque de différences sur celle de sortie (sous réserve qu'un vecteur de perturbation existe), avec des poids relativement faibles. Évidemment, le deuxième effet sera aussi que la contrainte 3 n'est plus à vérifier puisque l'on ne se place plus dans un comportement linéaire et que par conséquent, de meilleurs vecteurs de perturbation peuvent être utilisés.

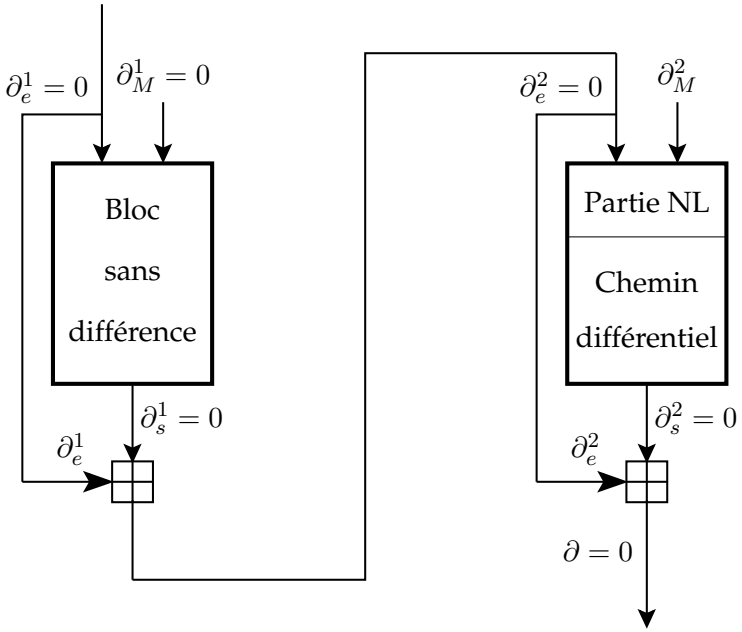

FIG. 5.10 – Attaque en un bloc de différence avec partie non linéaire pour SHA-0 [WYY05d].

Dans le cadre d'une attaque multiblocs, nous n'avons à présent besoin que de deux itérations. Nous n'allons en fait utiliser qu'un seul vecteur de perturbation pour les deux blocs de message qui contiennent des différences. Ceci est explicité dans la figure 5.9. On recherche tout d'abord le meilleur vecteur de perturbation (celui avec la plus forte probabilité de réussite entre les étapes i et 80, où i représente l'étape à partir de laquelle les modifications de message ne peuvent plus être appliquées) sans tenir compte des contraintes 1, 2 et 3 et quelles que soient les différences présentes sur la variable de chaînage d'entrée et celle de sortie. Soit V ce vecteur, qui fait correspondre une différence binaire $\Delta_e^{\oplus}$ en entrée à une différence binaire $\Delta_s^{\oplus}$ en sortie. Ce vecteur étant recherché en linéarisant le schéma, il est par conséquent non signé au départ. Les conditions sur les mots de messages étendus, au choix de l'attaquant, permettront de le signer lors de la recherche d'une instance de paire de messages valide. Son masque de différences binaires en entrée et en sortie peut de ce fait correspondre à des différences binaires signées montantes ou descendantes, fournissant directement les différences modulaires correspondantes. Ainsi, V peut être utilisé comme un vecteur possédant une différence modulaire ∂_s ou $-\partial_s$ sur la variable de chaînage de sortie, il suffit d'inverser les signes. Le coût de ce vecteur restera inchangé, quels que soient les signes utilisés. On choisit donc au hasard une

instance ∂_s de différence modulaire. Le premier bloc commence par une différence nulle et l'on cherchera une partie non linéaire NL_1 permettant de se replacer dans les 16 premières étapes sur le vecteur de perturbation V à partir d'une différence nulle. On obtient alors une différence modulaire ∂_s avant et après application du rebouclage à la fin du premier bloc. Cette même différence sera présente en entrée de la deuxième itération et il faudra rechercher une autre partie non linéaire NL_2 partant cette fois de ∂_s et se replaçant sur V dans les 16 premières étapes. À la fin de V, nous avons une différence $-\partial_s$ qui après application du rebouclage aboutit à une différence nulle, une collision. On peut remarquer que le masque de différences en entrée de V n'a aucune importance et seul celui de sortie est à considérer.

Nous avons considéré jusqu'ici qu'il était toujours possible de trouver une partie non linéaire. Toutefois, dans les articles originaux, aucun détail ne fut donné à ce sujet et il semble que les parties non linéaires de SHA-0 et de SHA-1 furent, comme les modifications de message, trouvées *à la main*.

Il faut noter que même si une partie non linéaire fut utilisée par l'équipe de recherche chinoise pour SHA-0, aucune technique multiblocs ne fut considérée, contrairement au cas de SHA-1. Cela peut s'expliquer par l'incapacité de trouver *à la main* un chemin non linéaire pour les deux itérations pour SHA-0. L'attaque nécessite cependant deux blocs, dont le premier ne contient aucune différence. En effet, certaines conditions sur la valeur des bits dans la variable de chaînage d'entrée sont requises par la partie non linéaire du deuxième et dernier bloc (conditions non vérifiées dans les valeurs d'initialisation spécifiées par l'algorithme). Ceci est explicité dans la figure 5.10

```
00111  01111001010010101000 01100010001011100000
       01000001000000000100 11001101011101000000
```

FIG. 5.11 – Vecteur de perturbation utilisé pour l'attaque en un bloc sur SHA-0 [WYY05d]. Le vecteur se lit de la gauche vers la droite, les 5 premiers bits représentant les perturbations virtuelles sur la variable de chaînage en entrée.

Finalement, le vecteur de perturbation utilisé pour SHA-0 est donné dans la figure 5.11. On peut remarquer que les contraintes 1 et 3 n'ont plus besoin d'être vérifiées grâce à la partie non linéaire. Par contre, la contrainte 2 est toujours vérifiée, car nous ne sommes pas en présence d'une attaque multiblocs. Le chemin différentiel final est donné dans la figure 5.12, les auteurs ont annoncé la recherche d'une collision en approximativement 2^{39} appels à la fonction de compression à l'aide de la technique de modification de message.

5.3.3 La recherche de vecteurs de perturbation pour SHA-1

Nous nous sommes jusqu'à présent surtout concentrés sur le cas de SHA-0, car les précédentes améliorations ne suffisaient pas à attaquer SHA-1. La modification de message et la partie non linéaire aboutirent en fait à la première attaque théorique contre SHA-1 [WYY05b, WYY05c, WYY05a]. Les méthodes sont identiques à celles utilisées pour SHA-0, excepté la recherche de vecteurs de perturbation. En effet, la rotation dans l'expansion de message change complètement la situation et il a fallu créer une nouvelle heuristique pour trouver de bons candidats.

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	----------				
-3 :	----------				
-2 :	----------1-----1-1-				
-1 :	----------1-----1-0-				
00 :	--------11110011111--11110------	n----------n----u-	29	-20.00	-212.97
01 :	u-------01001110111-101nuu--1-u-	----------1----1-	30	-25.00	-203.97
02 :	0-0--11111nuuuuuuuuu1-unnun1---	n----------n----n-	29	-29.03	-198.97
03 :	1-1--111110-0nuu0010u0u--01010n-	----------u----0-	30	-22.00	-199.00
04 :	1----1-nuuuuu000--1111-n110011--	----------1100-0------u----1-	25	-14.00	-191.00
05 :	0------n-000100100100-0-01-nu---	----------101----------n----n-	27	-16.00	-180.00
06 :	-------011011110nu0-unn--0--10n-	n---------0----------0----u-	28	-18.00	-169.00
07 :	0------11101111u0100--000----1n-	n----------1----n-	29	-16.00	-159.00
08 :	--------unnnnnnnnnnn-111---1---	----------0----------u----0-	29	-13.00	-146.00
09 :	1----------000000011001-----0---	n----------111010--------0----1-	23	-7.00	-130.00
10 :	----------111111111111-------n-	n----------n----0-	29	-3.00	-114.00
11 :	0----------n-----1---	n----------0----n-	29	-3.00	-88.00
12 :	0----------0---0---	----------0----n-	30	-3.00	-62.00
13 :	----------1---0-n-	n----------u----0-	29	-2.00	-35.00
14 :	1----------1---	n----------0----0-	29	-3.00	-8.00
15 :	0----------1-u-	n----------u----0-	29	-2.00	18.00
16 :	1----------0---	x----------0----1-	0	-2.00	45.00
17 :	0----------u-	----------n----0-	0	-1.00	43.00
18 :	1----------	x----------0----u-	0	0.00	42.00
19 :	----------	----------1----1-	0	0.00	42.00
20 :	----------	x----------1----0-	0	0.00	42.00
21 :	----------	x----------0----u-	0	-1.00	42.00
22 :	----------u-	----------n----n-	0	-1.00	41.00
23 :	----------n-	----------u----u-	0	-1.00	40.00
24 :	----------	x----------1----n-	0	-1.00	39.00
25 :	----------	----------0----1-	0	0.00	38.00
26 :	----------	----------1----n-	0	-1.00	38.00
27 :	----------n-	x----------u----1-	0	0.00	37.00
28 :	----------	----------1----u-	0	-1.00	37.00
29 :	----------	x----------1----1-	0	0.00	36.00
30 :	----------	x----------1----n-	0	-1.00	36.00
31 :	----------n-	x----------u----0-	0	0.00	35.00
32 :	----------	----------0----1-	0	-2.00	35.00
33 :	----------u-	x----------n----n-	0	-1.00	33.00
34 :	----------n-	x----------u----1-	0	-2.00	32.00
35 :	----------u-	----------n----n-	0	-1.00	30.00
36 :	----------	----------0----u-	0	-1.00	29.00
37 :	----------	x----------1----0-	0	0.00	28.00
38 :	----------	----------0----0-	0	0.00	28.00
39 :	----------	x----------0----1-	0	0.00	28.00
40 :	----------	----------1----1-	0	0.00	28.00
41 :	----------	----------1----n-	0	-1.00	28.00
42 :	----------n-	----------u----1-	0	0.00	27.00
43 :	----------	----------1----u-	0	-1.00	27.00
44 :	----------	x----------1----0-	0	-1.00	26.00
45 :	----------	x----------0----0-	0	-1.00	25.00
46 :	----------	x----------0----0-	0	0.00	24.00
47 :	----------	----------0----u-	0	-1.00	24.00
48 :	----------u-	----------n----1-	0	0.00	23.00
49 :	----------	----------1----n-	0	-1.00	23.00
50 :	----------	x----------0----0-	0	-1.00	22.00
51 :	----------	x----------0----0-	0	-1.00	21.00
52 :	----------	x----------0----1-	0	0.00	20.00
53 :	----------	----------1----1-	0	0.00	20.00
54 :	----------	----------1----1-	0	0.00	20.00
55 :	----------	----------1----1-	0	0.00	20.00
56 :	----------	----------1----0-	0	0.00	20.00
57 :	----------	----------0----n-	0	-1.00	20.00
58 :	----------n-	----------u----0-	0	0.00	19.00
59 :	----------	----------0----u-	0	-1.00	19.00
60 :	----------	x----------1----u-	0	-1.00	18.00
61 :	----------u-	x----------n----n-	0	-1.00	17.00
62 :	----------n-	x----------u----u-	0	-1.00	16.00
63 :	----------	x----------1----u-	0	-1.00	15.00
64 :	----------	----------1----u-	0	-1.00	14.00
65 :	----------u-	----------n----u-	0	-1.00	13.00
66 :	----------u-	x----------n----n-	0	-1.00	12.00
67 :	----------	x----------0----1-	0	-2.00	11.00
68 :	----------u-	----------n----0-	0	0.00	9.00
69 :	----------	----------0----0-	0	-2.00	9.00
70 :	----------n-	----------u----u-	0	-1.00	7.00
71 :	----------u-	x----------n----0-	0	-2.00	6.00
72 :	----------n-	----------u----u-	0	-1.00	4.00
73 :	----------	----------1----1-	0	-2.00	3.00
74 :	----------u-	x----------n----1-	0	0.00	1.00
75 :	----------	----------1----u-	0	-1.00	1.00
76 :	----------	----------1----1-	0	0.00	0.00
77 :	----------	x----------1----0-	0	0.00	0.00
78 :	----------	x----------0----0-	0	0.00	0.00
79 :	----------	----------0----1-	0	0.00	0.00
80 :	----------				

FIG. 5.12 – Chemin différentiel utilisé pour l'attaque en un bloc sur SHA-0 avec partie non linéaire [WYY05d].

Tout d'abord, les remarques précédentes sur la position des perturbations pour SHA-0 restent valides : la position de bit 1 est à privilégier pour introduire une perturbation, car c'est celle qui nécessite le moins de conditions en général. Hélas, dans le cas de SHA-1, il est impossible de confiner à une seule position l'ensemble des perturbations à cause de la rotation dans l'expansion de message. Cette rotation engendre deux problèmes : on ne peut plus se contenter facilement de choisir la position de bit 1, et l'espace de recherche parmi les vecteurs de perturbations devient immense. Dans le cas de SHA-0, nous avions pu réduire l'espace de recherche à un vecteur de 16 bits. Pour SHA-1, l'espace est à présent de 16 mots de 32 bits chacun : un bit pour chaque position d'introduction. Les vecteurs de perturbation entiers (pour les 80 étapes) peuvent être déduits en utilisant comme précédemment la formule d'expansion de message ou son inverse si nécessaire.

i	vecteur de perturbation
16 :	1----------------------------1-
17 :	--------------------------------
18 :	----------------------------1-
19 :	--------------------------------
20 :	----------------------------11
21 :	--------------------------------
22 :	----------------------------1-
23 :	----------------------------1-
24 :	----------------------------1
25 :	--------------------------------
26 :	----------------------------1-
27 :	----------------------------1-
28 :	----------------------------1
29 :	--------------------------------
30 :	--------------------------------
31 :	----------------------------1-
32 :	----------------------------11
33 :	--------------------------------
34 :	----------------------------1-
35 :	----------------------------1-
36 :	--------------------------------
37 :	--------------------------------
38 :	----------------------------1-
39 :	--------------------------------
40 :	--------------------------------
41 :	--------------------------------
42 :	----------------------------1-
43 :	--------------------------------
44 :	----------------------------1-
45 :	--------------------------------
46 :	----------------------------1-
47 :	--------------------------------

i	vecteur de perturbation
48 :	----------------------------1-
49 :	--------------------------------
50 :	--------------------------------
51 :	--------------------------------
52 :	--------------------------------
53 :	--------------------------------
54 :	--------------------------------
55 :	--------------------------------
56 :	--------------------------------
57 :	--------------------------------
58 :	--------------------------------
59 :	--------------------------------
60 :	--------------------------------
61 :	--------------------------------
62 :	--------------------------------
63 :	--------------------------------
64 :	----------------------------1--
65 :	--------------------------------
66 :	--------------------------------
67 :	----------------------------1---
68 :	--------------------------------
69 :	--------------------------------
70 :	----------------------------1----
71 :	--------------------------------
72 :	----------------------------1---
73 :	----------------------------1-----
74 :	--------------------------------
75 :	--------------------------------
76 :	----------------------------1------
77 :	--------------------------------
78 :	----------------------------1-1---
79 :	----------------------------1-------

FIG. 5.13 – Vecteur de perturbation pour l'attaque multiblocs sur SHA-1 en 2^{69} appels à la fonction de compression [WYY05b]. Les 16 premières étapes ont été ôtées, car remplacées par la partie non linéaire en pratique. Les caractères « 0 » ont été remplacés par « - » pour une meilleure lisibilité.

Wang *et al.* [WYY05c] choisirent une méthode assez simple pour évaluer rapidement la qualité d'un vecteur de perturbation. Chaque perturbation commençant dans une étape utilisant la fonction booléenne XOR induit 2 conditions si elle est basée sur la position de bit 1, et 4 conditions sinon. Pour une perturbation commençant dans une étape utilisant la fonction booléenne IF (respectivement MAJ), on compte 5 conditions (respectivement 4 conditions) quelle que soit la position de bit. Les situations où des perturbations affectent deux fonctions booléennes différentes (perturbations introduites à l'étape 37 par exemple) sont aussi prises en compte et traitées au cas par cas. Une fois la méthode d'évaluation d'un vecteur définie, Wang *et al.* testent une sous-partie de l'espace des candidats possibles : tous les vecteurs sur 16 étapes dont les

perturbations sont confinées aux positions de bit 0 ou 1. On considère aussi les vecteurs décalés d'un certain nombre d'étapes positif ou négatif. Cela demande donc de traiter à peu près 64×2^{32} vecteurs différents (2^{32} vecteurs et 64 décalages possibles). Parmi ces candidats, ils gardèrent alors celui ayant le moins de conditions entre les étapes 16 et 80.

Cette méthode aboutit au vecteur présenté dans la figure 5.13. On peut voir qu'un très petit nombre de perturbations sont introduites dans le troisième tour. Cela semble logique puisque la fonction booléenne MAJ impose de nombreuses conditions et est donc coûteuse pour l'attaquant. De plus, cette forme de vecteur, de très faible poids de Hamming dans les tours intermédiaires et de poids plus fort dans les premiers et les derniers tours (ce qui est inévitable du fait de l'expansion de message et de sa rotation), permet de maximiser le ratio de perturbations introduites à la position de bit 1, position qui est très avantageuse dans le deuxième et dernier tour à cause de la fonction booléenne XOR. Nous donnons dans la figure 5.15 le chemin différentiel final pour le premier bloc fourni dans l'article original. Il faut noter que ce chemin comportait une erreur que nous avons corrigé dans notre figure : la valeur d'initialisation spécifiée contredit certaines conditions pourtant nécessaires quant à la validité de la partie non linéaire. C'est la raison pour laquelle la valeur initiale de la variable de chaînage d'entrée n'a pas été ajoutée dans la figure.

i	vecteur de perturbation
16 :	1----------------------------1-
17 :	----------------------------1-
18 :	1----------------------------1-
19 :	----------------------------
20 :	----------------------------1-
21 :	----------------------------
22 :	----------------------------11
23 :	----------------------------
24 :	----------------------------1-
25 :	----------------------------1-
26 :	----------------------------1
27 :	----------------------------
28 :	----------------------------1-
29 :	----------------------------1-
30 :	----------------------------1
31 :	----------------------------
32 :	----------------------------
33 :	----------------------------1-
34 :	----------------------------11
35 :	----------------------------
36 :	----------------------------1-
37 :	----------------------------1-
38 :	----------------------------
39 :	----------------------------
40 :	----------------------------1-
41 :	----------------------------
42 :	----------------------------
43 :	----------------------------
44 :	----------------------------1-
45 :	----------------------------
46 :	----------------------------1-
47 :	----------------------------

i	vecteur de perturbation
48 :	----------------------------1-
49 :	----------------------------
50 :	----------------------------1-
51 :	----------------------------
52 :	----------------------------
53 :	----------------------------
54 :	----------------------------
55 :	----------------------------
56 :	----------------------------
57 :	----------------------------
58 :	----------------------------
59 :	----------------------------
60 :	----------------------------
61 :	----------------------------
62 :	----------------------------
63 :	----------------------------
64 :	----------------------------
65 :	----------------------------
66 :	---------------------------1--
67 :	----------------------------
68 :	----------------------------
69 :	---------------------------1---
70 :	----------------------------
71 :	----------------------------
72 :	---------------------------1----
73 :	----------------------------
74 :	---------------------------1---
75 :	---------------------------1-----
76 :	----------------------------
77 :	----------------------------
78 :	---------------------------1------
79 :	----------------------------

FIG. 5.14 – Vecteur de perturbation pour l'attaque multiblocs sur SHA-1 en 2^{63} appels à la fonction de compression [WYY05a]. Les 16 premières étapes ont été ôtées, car elles sont en pratique remplacées par la partie non linéaire. Les caractères « 0 » ont été remplacés par « $-$ » pour une meilleure lisibilité.

Wang *et al.* ont estimé que grâce à la technique de modification de message une complexité d'approximativement 2^{69} appels à la fonction de compression pouvait être atteinte pour trouver une collision sur SHA-1 en deux blocs [WYY05b]. Quelques mois plus tard, un

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	----------				
-3 :	----------				
-2 :	----------				
-1 :	-11-------				
00 :	-00-------	-nn---------	30	-7.00	-88.19
01 :	-nn--------	nuu---------	26	-13.00	-65.19
02 :	nuu10------	--n---------	28	-17.42	-52.19
03 :	0-100------	u-un--------	28	-27.19	-41.60
04 :	u0010------	nn-n--------	25	-25.00	-40.79
05 :	0011n-----	nn-n--------	27	-27.09	-40.79
06 :	1-0-0nu11-	--n---------	31	-26.42	-40.89
07 :	n--110111-	-nn---------	27	-24.87	-36.30
08 :	-01---unnn	-uu---------	27	-21.83	-34.17
09 :	-00-----10	--u---------	30	-12.00	-29.00
10 :	0----------	unu---------	27	-16.00	-11.00
11 :	----------	-uu---------	29	-10.00	0.00
12 :	0----------	n----------	30	-3.00	19.00
13 :	----------	----------	31	-4.00	46.00
14 :	-1--------	--1-------	29	-6.00	73.00
15 :	u0--------	-u--------	28	-3.00	96.00
16 :	--1-------	-x----------	0	-4.00	121.00
17 :	n-0-------	xnn---------	0	-1.00	117.00
18 :	--1-------	x-n---------	0	-1.00	116.00
19 :	----------	x----------	0	0.00	115.00
20 :	----------	--x---------	0	-3.00	115.00
21 :	----------	--u---------	0	-2.00	112.00
22 :	----------	x----------	0	-2.00	110.00
23 :	----------	-x---------	0	-3.00	108.00
24 :	----------	xx---------	0	-4.00	105.00
25 :	----------	-x---------	0	-3.00	101.00
26 :	----------	----------	0	-2.00	98.00
27 :	----------	-x---------	0	-3.00	96.00
28 :	----------	xx---------	0	-4.00	93.00
29 :	----------	xx---------	0	-3.00	89.00
30 :	----------	----------	0	-1.00	86.00
31 :	----------	-x---------	0	-2.00	85.00
32 :	----------	xx---------	0	-4.00	83.00
33 :	----------	-x---------	0	-4.00	79.00
34 :	----------	x----------	0	-2.00	75.00
35 :	----------	-x---------	0	-3.00	73.00
36 :	----------	-x---------	0	-3.00	70.00
37 :	----------	-x---------	0	-2.00	67.00
38 :	----------	----------	0	-1.00	65.00
39 :	----------	----------	0	-1.00	64.00
40 :	----------	x----------	0	-2.00	63.00
41 :	----------	x----------	0	-1.00	61.00
42 :	----------	x----------	0	-2.00	60.00
43 :	----------	x----------	0	-1.00	58.00
44 :	----------	----------	0	-2.00	57.00
45 :	----------	x----------	0	-2.00	55.00
46 :	----------	x----------	0	-3.00	53.00
47 :	----------	----------	0	-2.00	50.00
48 :	----------	x----------	0	-3.00	48.00
49 :	----------	----------	0	-2.00	45.00
50 :	----------	x----------	0	-3.00	43.00
51 :	----------	----------	0	-1.00	40.00
52 :	----------	x----------	0	-1.00	39.00
53 :	----------	x----------	0	0.00	38.00
54 :	----------	----------	0	0.00	38.00
55 :	----------	----------	0	0.00	38.00
56 :	----------	----------	0	0.00	38.00
57 :	----------	----------	0	0.00	38.00
58 :	----------	----------	0	0.00	38.00
59 :	----------	----------	0	0.00	38.00
60 :	----------	----------	0	0.00	38.00
61 :	----------	----------	0	0.00	38.00
62 :	----------	----------	0	0.00	38.00
63 :	----------	----------	0	0.00	38.00
64 :	----------	----------	0	-1.00	38.00
65 :	----------	----------	0	-1.00	37.00
66 :	----------	----------	0	-1.00	36.00
67 :	----------	----------	0	-2.00	35.00
68 :	----------	----------	0	-2.00	33.00
69 :	----------	----------	0	-2.00	31.00
70 :	----------	----------	0	-2.00	29.00
71 :	----------	----------	0	-2.00	27.00
72 :	----------	----------	0	-3.00	25.00
73 :	----------	----------	0	-3.00	22.00
74 :	----------	----------	0	-3.00	19.00
75 :	----------	----------	0	-3.00	16.00
76 :	----------	----------	0	-3.00	13.00
77 :	----------	----------	0	-3.00	10.00
78 :	----------	----------	0	-3.00	7.00
79 :	----------	----------	0	-4.00	4.00
80 :	----------				

FIG. 5.15 – Chemin différentiel pour le premier bloc de l'attaque multiblocs sur SHA-1 en 2^{69} appels à la fonction de compression [WYY05b].

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010110111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	u0u----------un	27	-4.01	-106.93
01 :	nun----------0-----0----00-0u	--n----------nu----	29	-13.68	-83.95
02 :	-1n00-1----------00----111-00-u01	-nn----------1-0-1--0----000	23	-16.00	-68.63
03 :	nu01-----unnnnnnnnnnnnnn1---n01	unu----------11--------u-u-u-	24	-19.75	-61.63
04 :	n-101------0000000un0010nun---u-	--n----------n---1nu	27	-38.05	-57.38
05 :	0-01n-0--u-1111111111-0un0nu0100	n-nu----------u------	28	-17.42	-68.43
06 :	u-011-0--0-1111-----nu---101-0--	nn-n----------n-u--nu	25	-22.00	-57.84
07 :	--0-nuuuuuuuuuuu---0-10-00-01-n	nn-n----------n---u-	27	-19.42	-54.84
08 :	110---01111011111-1-11----n0--	--u----------	31	-24.00	-47.26
09 :	n0----1111111111111---un----1-0u	-un----------nn--u-	27	-4.74	-40.26
10 :	-11----------11--1-n-	-nn----------u----un	27	-10.00	-18.00
11 :	100----------un11--0---	--u----------n------	30	-4.00	-1.00
12 :	0----------01----n-	nnu----------u----u-	27	-5.00	25.00
13 :	0----------11------	-uu----------n-	29	-1.00	47.00
14 :	1----------	n-0----------n	29	-1.00	75.00
15 :	----------n	----------n-----	31	-1.00	103.00
16 :	-1----------	-1----------xx	0	-4.00	133.00
17 :	x0----------x-	-u----------x-x--x-	0	-4.00	129.00
18 :	--1----------x-	-x----------x------	0	-4.00	125.00
19 :	x----------x-	xxx----------x-x--x-	0	-6.00	121.00
20 :	----------	x-x----------	0	-2.00	115.00
21 :	----------x-	x----------x------	0	-2.00	113.00
22 :	----------	--x----------x	0	-3.00	111.00
23 :	----------xx	--x----------xx-----	0	-3.00	108.00
24 :	----------	x----------x	0	-2.00	105.00
25 :	----------x-	-x----------x----x-	0	-3.00	103.00
26 :	----------x-	xx----------x----xx	0	-4.00	100.00
27 :	----------x	-x----------x---x-	0	-3.00	96.00
28 :	----------	----------xx	0	-2.00	93.00
29 :	----------x-	-x----------x----x-	0	-3.00	91.00
30 :	----------x-	xx----------x----xx	0	-4.00	88.00
31 :	----------x	xx----------x---x-	0	-3.00	84.00
32 :	----------	----------x	0	-1.00	81.00
33 :	----------	-x----------x-	0	-2.00	80.00
34 :	----------x-	xx----------x----xx	0	-4.00	78.00
35 :	----------xx	-x----------xx---x-	0	-4.00	74.00
36 :	----------	x----------x	0	-2.00	70.00
37 :	----------x-	-x----------x----x-	0	-3.00	68.00
38 :	----------x-	-x----------x----x-	0	-3.00	65.00
39 :	----------	-x----------x-	0	-2.00	62.00
40 :	----------	----------x-	0	-2.00	60.00
41 :	----------x-	----------x------	0	-2.00	58.00
42 :	----------	x----------x-	0	-2.00	56.00
43 :	----------	x----------	0	-1.00	54.00
44 :	----------	x----------x-	0	-2.00	53.00
45 :	----------x-	x----------x------	0	-1.00	51.00
46 :	----------	----------	0	-2.00	50.00
47 :	----------x-	x----------x------	0	-2.00	48.00
48 :	----------	x----------	0	-3.00	46.00
49 :	----------x-	----------x------	0	-2.00	43.00
50 :	----------	x----------	0	-3.00	41.00
51 :	----------x-	----------x------	0	-2.00	38.00
52 :	----------	x----------x-	0	-3.00	36.00
53 :	----------	----------	0	-1.00	33.00
54 :	----------	x----------	0	-1.00	32.00
55 :	----------	x----------	0	0.00	31.00
56 :	----------	----------	0	0.00	31.00
57 :	----------	----------	0	0.00	31.00
58 :	----------	----------	0	0.00	31.00
59 :	----------	----------	0	0.00	31.00
60 :	----------	----------	0	0.00	31.00
61 :	----------	----------	0	0.00	31.00
62 :	----------	----------	0	0.00	31.00
63 :	----------	----------	0	0.00	31.00
64 :	----------	----------	0	0.00	31.00
65 :	----------	----------	0	0.00	31.00
66 :	----------	----------x--	0	-1.00	31.00
67 :	----------x--	----------x--------	0	-1.00	30.00
68 :	----------	----------x--	0	-1.00	29.00
69 :	----------	----------x--x	0	-2.00	28.00
70 :	----------x---	----------x-------x	0	-2.00	26.00
71 :	----------	----------x--x	0	-2.00	24.00
72 :	----------	----------x--x-	0	-2.00	22.00
73 :	----------x----	----------x-------x-	0	-2.00	20.00
74 :	----------	----------xx-x-	0	-3.00	18.00
75 :	----------x---	----------x--x--x--	0	-3.00	15.00
76 :	----------x-----	----------x-----xx--	0	-3.00	12.00
77 :	----------	----------x--xx-	0	-3.00	9.00
78 :	----------	----------x--x-x-	0	-3.00	6.00
79 :	----------x------	----------x-------x-x-	0	-3.00	3.00
80 :	----------				

FIG. 5.16 – Chemin différentiel pour le premier bloc de l'attaque multiblocs sur SHA-1 en 2^{63} appels à la fonction de compression [WYY05a].

i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010110111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	u0u00000100010111100100100 1100un	0	0.00	40.38
01 :	nun0000001000000011000011110010u	01n10100100010110010010111nu1000	0	0.00	40.38
02 :	11n00011010010000010111110001u01	1nn00010000010010110110101011000	0	0.00	40.38
03 :	nu0110100unnnnnnnnnnnnnnn1101n01	unu10010111110001110100001u0u0u1	0	0.00	40.38
04 :	n0101010000000000un0010nun110u0	00n01101010101111010-110n0001nu	1	0.00	40.38
05 :	0101n0011u11111111111-0un0nu0100	n1nu1100010110 10-111-----u111011	6	-4.00	41.38
06 :	u00110011011111101011nu-1-1010010	nn0n1110010101001 1-1010-n0u10nu	2	-1.00	43.38
07 :	0001nuuuuuuuuuuu-110010-000011n	nn0n01101010-000011-0----1n001u1	6	-5.00	44.38
08 :	11001001111011111101011-000n000	11u10110100100000-0-0------00000	8	-3.51	45.38
09 :	n0011111111111111---un---0110u	1un0110001111 11-111------0nn00u1	7	-1.00	49.87
10 :	11100011011-------------11-110n0	1nn10100111111111-----01-u1110un	6	-4.00	55.87
11 :	10010110----------------un11-00011	01u0100010011111--0-0----0n011101	7	-1.42	57.87
12 :	011--0-----------------01---0n1	nnu001001-1011-001-------u0110u0	9	-4.46	63.46
13 :	0----------------------11---0--	0uu100000010-000----1-----0000n1	10	-3.00	68.00
14 :	1---1-------------------------1--	n00010110010-1--0-0-------11001n	11	-1.00	75.00
15 :	------------------------0--n	01011001-1111-101-------00n10010	9	-2.00	85.00
16 :	-1--------------------------1---	01001000011-010----1------0101uu	0	-4.00	92.00
17 :	u0--------------------------0-u-	0u000010010-0--0-1-------n0n01n0	0	-4.00	88.00
18 :	0-1-------------------------0-n-	1n00010-1011-111--------0u101010	0	-4.00	84.00
19 :	n-0---------------------------u-	unn1110010-100----0------n1u01n1	0	-1.00	80.00
20 :	--0----------------------------	u1n01011-1-1--1-0-------10000100	0	-1.00	79.00
21 :	----------------------------u-	n10111-0110-101----------n101111	0	0.00	78.00
22 :	----------------------------	11n111101-1-0----1--------11110n	0	-4.00	78.00
23 :	----------------------------un	10u1000-1-1--1-0--------1nu00100	0	0.00	74.00
24 :	----------------------------	u1101-0110-000------------00010n	0	-3.00	74.00
25 :	----------------------------n-	0u110000-0-0----1--------u0011n0	0	-2.00	71.00
26 :	----------------------------n-	nn0010-0-0--0-0----------1u0111nu	0	-3.00	69.00
27 :	----------------------------u	1u01-0110-0-1------------1n110u0	0	-1.00	66.00
28 :	----------------------------	0010100---1----1---------10000nu	0	-2.00	65.00
29 :	----------------------------n-	1u111-1-0----0-----------u0000u0	0	-2.00	63.00
30 :	----------------------------u-	un0-1101-1----------------n0001un	0	-3.00	61.00
31 :	----------------------------n	nn0001---1----1---------11u001u1	0	-1.00	58.00
32 :	----------------------------	1011-1-0----0------------111011u	0	-1.00	57.00
33 :	----------------------------	0u-0011-0----------------111101u0	0	-2.00	56.00
34 :	----------------------------u-	un000---0----1---------11n1010nn	0	-3.00	54.00
35 :	----------------------------nn	0u0-1-1----0------------0uu011u1	0	-1.00	51.00
36 :	----------------------------	u-1101----------------------001111n	0	-3.00	50.00
37 :	----------------------------n-	0n10---0----0-----------1u0000u1	0	-2.00	47.00
38 :	----------------------------u-	1u-0-0----------------------n1000u1	0	-2.00	45.00
39 :	----------------------------	-u100----------------------001000u0	0	-1.00	43.00
40 :	----------------------------	111---1----0------------10110u0	0	-1.00	42.00
41 :	----------------------------u-	0-0-1----------------------1n000110	0	-1.00	41.00
42 :	----------------------------	u000------------------------001011n-	0	-1.00	40.00
43 :	----------------------------	n0---0----------------------01011011	0	-1.00	39.00
44 :	----------------------------	x0-1------------------------101111u0	0	-2.00	38.00
45 :	----------------------------u-	u01------------------------1n0101-0	0	0.00	36.00
46 :	----------------------------	1---0----------------------00010011	0	-2.00	36.00
47 :	----------------------------u-	n-1------------------------10n01101-	0	-1.00	34.00
48 :	----------------------------	u1-------------------------00100-10	0	-3.00	33.00
49 :	----------------------------u-	---0-----------------------00n011001	0	-1.00	30.00
50 :	----------------------------	x0-------------------------000000-1	0	-3.00	29.00
51 :	----------------------------u-	0--------------------------11n10-011	0	-1.00	26.00
52 :	----------------------------	x-0------------------------001101n-	0	-2.00	25.00
53 :	----------------------------	1--------------------------011101-0-	0	-1.00	23.00
54 :	----------------------------	x--------------------------100-0011	0	-1.00	22.00
55 :	----------------------------	x0-------------------------1000101--	0	0.00	21.00
56 :	----------------------------	------------------------0111-0-0	0	0.00	21.00
57 :	----------------------------	------------------------010-0111-	0	0.00	21.00
58 :	----------------------------	0------------------------100001---	0	0.00	21.00
59 :	----------------------------	------------------------0001-1-0-	0	0.00	21.00
60 :	----------------------------	------------------------00-1011--	0	0.00	21.00
61 :	----------------------------	------------------------01101---0	0	0.00	21.00
62 :	----------------------------	------------------------100-1-1--	0	0.00	21.00
63 :	----------------------------	------------------------00-1110---	0	0.00	21.00
64 :	----------------------------	------------------------1000---1-	0	0.00	21.00
65 :	----------------------------	------------------------000-0-0---	0	0.00	21.00
66 :	----------------------------	------------------------1001-n--	0	-1.00	21.00
67 :	----------------------------n--	------------------------00u1---0--	0	0.00	20.00
68 :	----------------------------	------------------------1-0-1-x--	0	-1.00	20.00
69 :	----------------------------	------------------------1011-u--x	0	-2.00	19.00
70 :	----------------------------u---	------------------------n0---1--x	0	-1.00	17.00
71 :	----------------------------	------------------------0-1-1-x--u	0	-1.00	16.00
72 :	----------------------------	------------------------000-u--x-	0	-2.00	15.00
73 :	----------------------------u----	------------------------n1---1--x-	0	-1.00	13.00
74 :	----------------------------	------------------------0-0-xu-n-	0	-2.00	12.00
75 :	----------------------------u---	------------------------0n0-n--x--	0	-2.00	10.00
76 :	----------------------------n-----	------------------------u1---1-xx--	0	-2.00	8.00
77 :	----------------------------	------------------------1-1-x--nx-	0	-2.00	6.00
78 :	----------------------------	------------------------00-n--x-x-	0	-3.00	4.00
79 :	----------------------------n------	------------------------u1---1--x-n-	0	-1.00	1.00
80 :	----------------------------				

FIG. 5.17 – Une instance signée du chemin différentiel de la figure 5.16.

autre vecteur fut trouvé, lequel aboutit à une complexité annoncée de 2^{63} appels à la fonction de compression [WYY05a]. Ce nouveau vecteur, en fait identique au premier, mais décalé de 2 étapes, est présenté dans la figure 5.14. Il comporte plus de conditions que le précédent, mais les modifications de message y sont plus efficaces et permettent en fin de compte un gain pour l'attaquant. Le chemin différentiel non signé correspondant pour le premier bloc est donné dans la figure 5.16. Une instance possible des signes des perturbations est enfin donnée dans la figure 5.17. Il faut noter que nous avons encore corrigé une nouvelle erreur présente dans [WYY05a] concernant la partie non linéaire du chemin différentiel originellement proposé.

Les complexités annoncées par Wang *et al.* furent assez controversées sur plusieurs points. D'une part, très peu de détails étaient donnés sur l'attaque et sur toute possible implantation. Plusieurs erreurs furent découvertes quelque temps après la publication des attaques. Au cours de la rédaction de cette thèse, nous avons nous-même trouvé plusieurs erreurs auparavant inconnues. De plus, étant donné que seul le chemin différentiel pour le premier bloc était explicité, on peut supposer que celui pour le deuxième est beaucoup plus difficile à trouver *à la main*. Un travail non automatisé pour trouver des parties non linéaires conduit souvent à des conditions sur la variable de chaînage d'entrée, ce qui n'est pas très contrariant pour le premier bloc (puisqu'il est facile de rajouter préalablement un bloc sans différence pour vérifier ces conditions sur la variable de chaînage d'entrée), mais qui peut l'être beaucoup plus pour le deuxième : les conditions sur la variable de chaînage d'entrée du deuxième bloc devront être comptées durant le calcul du nombre de conditions total pour le premier bloc, ce qui aura une influence significative sur la complexité finale de l'attaque. D'autre part, le coût réel d'une modification de message pouvait sembler assez imprécis. L'attaque finale étant très complexe, aucune implantation n'a à ce jour été publiée pour vérifier ce coût supposé. De nombreux travaux ont cependant comblé ce manque de détails et il est maintenant admis qu'une telle attaque est réalisable avec une complexité d'approximativement 2^{63} appels à la fonction de compression [Coc07].

5.3.4 Une analyse plus fine des conditions

La complexité annoncée par Wang *et al.* pour SHA-1 tient compte d'une analyse assez fine des conditions d'un chemin différentiel. Tout d'abord, on peut se concentrer sur la fin du vecteur, à savoir les 5 dernières étapes. On remarque que ces 5 dernières étapes produisent les variables de chaînage qui seront effectivement utilisées lors du rebouclage. Dans ce cas précis, le rebouclage n'utilisant que l'addition modulaire, il n'est plus nécessaire de considérer plusieurs types de différences et l'on peut se contenter de la différence modulaire. Pour le premier bloc, on pourra donc laisser les différences introduites se propager arbitrairement, sans imposer de conditions (à part leur signe modulaire durant l'introduction qui est choisi par l'attaquant à travers le message étendu). Pour le deuxième bloc, il suffira d'insérer des perturbations inverses d'un point de vue de l'addition modulaire pour corriger parfaitement et ainsi obtenir une collision après le rebouclage (la propagation des différences binaires n'est plus importante). Hélas, s'il y a propagation de différences, cela impliquera aussi la propagation de beaucoup de différences binaires dont le signe est inconnu après application de la fonction booléenne XOR. On souhaite éviter cette situation et l'on peut laisser les différences se propager tant qu'elles ne sont pas utilisées comme entrée de la fonction booléenne XOR. Cette situation se présente pour les deux dernières étapes. Toutes les perturbations insérées sur les deux dernières étapes peuvent donc être considérées sans conditions pour les deux blocs (puisque même si les différences se

propagent à travers les retenues, elles ne s'exprimeront pas dans la fonction booléenne).

Une analyse encore plus poussée met en évidence que pour le premier bloc, toutes les conditions relatives aux corrections des deux dernières étapes peuvent être omises. Autrement dit, nous aurons $P_i(78) = P_i(79) = 1$. Ceci est dû au fait que pour les deux dernières étapes, même si la correction d'une différence à l'entrée de la fonction booléenne XOR ne se passe pas bien, nous pourrons corriger ce comportement dans le deuxième bloc avec la même probabilité que si tout avait été parfaitement corrigé au premier. Seules les deux dernières étapes sont concernées pour les mêmes raisons que précédemment (tout comportement invalide ne doit pas pouvoir s'exprimer à travers la fonction booléenne). Ainsi, nous n'avons pas à contrôler la moindre condition pour les deux dernières étapes du premier bloc, sans pour autant changer la probabilité de succès pour le deuxième. Ceci explique aussi pourquoi le premier bloc coûte moins cher que le deuxième lors de la recherche d'une collision pour SHA, le coût de ce dernier étant le terme dominant pour la complexité totale de l'attaque.

non compressé				
i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$P_i(i)$	$C(i)$
$i-1$	----------	----------	0.00	1.5
i	----------	----------unn----------	-3.00	3.0
$i+1$	----------unn----------	----------nuu----------	0.00	3.0
$i+2$	----------	----------nuu----------	-3.00	3.0
$i+3$	----------	----------nuu--------	-3.00	1.5
$i+4$	----------	----------nuu--------	-3.00	0.0
$i+5$	----------	----------nuu--------	0.00	0.0
$i+6$	----------	----------	0.00	0.0

compressé				
i	$\bigtriangledown A_i$	$\bigtriangledown W_i$	$P_i(i)$	$C(i)$
$i-1$	----------	----------	0.00	0.5
i	----------	----------unn----------	-1.00	1.0
$i+1$	----------u----------	----------nuu----------	0.00	1.0
$i+2$	----------	----------nuu----------	-1.00	1.0
$i+3$	----------	----------nuu--------	-1.00	0.5
$i+4$	----------	----------nuu--------	-1.00	0.0
$i+5$	----------	----------nuu--------	0.00	0.0
$i+6$	----------	----------	0.00	0.0

FIG. 5.18 – Compression de bits de 3 collisions locales adjacentes aux positions de bit 10, 11 et 12. Les probabilités pour cet exemple correspondent aux fonctions booléennes XOR ou MAJ.

Moins qu'une réelle avancée de cryptanalyse, la *compression de bits* (terme introduit par Yajima *et al.* dans [YIN08a, YIN08b]) permet aussi de gagner quelques conditions étant donné un vecteur de perturbation. Elle intervient lorsque l'on introduit plusieurs différences consécutives sur des bits adjacents d'un même mot. Dans ce cas précis, l'introduction de ces perturbations peut être vue comme une seule et unique perturbation, positionnée sur le bit de poids le plus faible des bits concernés. Nous ne compterons alors que les conditions pour cette unique perturbation. Ceci s'explique par le fait qu'au lieu de contrôler que la première différence ne se propage pas, nous allons au contraire la forcer à se diffuser (le coût restant le même). On peut de ce fait gratuitement continuer de propager cette différence puis arrêter la propagation par la dernière perturbation (celle de poids fort). Le coût est nul, puisque tout est contrôlé par le signe de l'introduction des perturbations adjacentes, choisies préalablement par l'attaquant dans le message étendu. Dans le registre interne, une seule différence apparaîtra et nous n'aurons donc que celle-ci à corriger lorsqu'elle influencera la sortie de la fonction booléenne. Il faut cependant noter que certaines exceptions existent : il ne doit pas y avoir de différences consécutives sur les positions 1 et 2, 26 et 27 ou 31 et 0, car la propagation de la différence par l'addition modulaire ne serait plus assurée à cause des rotations de 2 ou de 5 positions dans la formule de

mise à jour des registres internes (la propagation des différences binaires signées serait arrêtée par le modulo de l'addition modulaire). De plus, si au moins l'une des perturbations se trouve dans une situation où deux différences sont présentes dans la fonction booléenne (situations 3, 5 ou 6), alors la technique ne s'applique plus pour les mêmes raisons que précédemment. Bien entendu, cette méthode ne sera pas utile dans le cas de SHA-0 puisque toutes les perturbations sont positionnées sur le bit 1 uniquement. Par contre, elle pourra être utilisée dans le cas de SHA-1, par exemple pour les perturbations insérées à l'étape 34 du vecteur de la figure 5.17. Nous donnons un exemple de compression de bit dans la figure 5.18.

Toutes ces petites améliorations permettent, lorsqu'elles sont considérées, un gain assez significatif en complexité. On peut généraliser ces techniques : à certaines étapes, plusieurs tronçons de chemin différentiels différents pourront convenir (ils aboutissent tous finalement au même masque de différences sur les registres internes après quelques étapes). On peut donc observer la réalisation de n'importe lequel de ces tronçons de chemin, et l'on peut ainsi additionner leurs probabilités de succès respectives pour obtenir la probabilité réelle lorsque l'on cherchera une paire de messages valide.

CHAPITRE 6
Amélioration des méthodes de cryptanalyse

Sommaire

Nous étudions ici de nouvelles méthodes ou des améliorations de cryptanalyses existantes pour la famille de fonctions de hachage SHA. Ces avancées ont toutes pour point de départ commun d'essayer de mieux comprendre les concepts sous-jacents des attaques de Wang *et al.*, très peu décrites et difficiles à vérifier. Cela permet soit la découverte de nouvelles techniques, soit une automatisation des anciennes, apportant ainsi une amélioration des résultats.

6.1 Un problème à plusieurs dimensions

Après la publication des attaques de l'équipe chinoise contre les nombreuses fonctions de hachage de la famille MD-SHA, il fallut au reste de la communauté de recherche en cryptographie une assez longue période pour comprendre le fonctionnement réel de ces techniques et pour vérifier leur exactitude. Cela ne fut pas aisé, du fait que peu de détails étaient donnés et que l'attaque complète est assez complexe. La preuve en est que, certaines erreurs furent découvertes après la publication des travaux et que l'attaque contre SHA-1 en 2^{63} appels à la fonction de compression n'a à ce jour toujours pas été publiée, ni décrite en détail. Une première étape de théorisation de ces avancées fut nécessaire.

Les chercheurs ont rencontré une autre difficulté dans l'analyse des avancées de Wang *et al.*. En effet, tout ayant été établi *à la main* et était décrit sans amples explications, il est en pratique impossible de réutiliser ces attaques en essayant d'y apporter une amélioration puisque tout le travail initial devrait alors être refait. Par exemple, la recherche d'une partie non linéaire ou les modifications de message devront être reconstituées à la main par le cryptanalyste. Un premier sujet d'étude pourrait donc être de trouver des algorithmes permettant d'effectuer ces tâches de manière automatisée dans l'espoir d'obtenir de meilleures attaques sans pour autant apporter des concepts innovants de cryptanalyse. Une seconde voie pourrait ensuite être, grâce à ces nouveaux outils, d'essayer de mettre au point de nouvelles méthodes pour une ou plusieurs étapes de l'attaque de Wang *et al.*. Par exemple, on pourra tenter d'améliorer la recherche de modifications de message ou de trouver de nouvelles heuristiques en ce qui concerne les vecteurs de perturbation.

Une question subsiste même si l'on dispose d'outils automatisés et de nouvelles méthodes : comment obtenir la meilleure attaque finale possible à partir de toutes les techniques connues ? On se rend rapidement compte que le problème est complexe et n'est pas unidimensionnel. Utiliser les outils dans un ordre précis n'apporte en général pas la meilleure attaque. Un attaquant pourrait être tenté de choisir tout d'abord un vecteur de perturbation qui possède un poids de Hamming faible pour minimiser la complexité, puis de trouver les parties non linéaires (si nécessaire) pour aboutir à un chemin différentiel complet ; et enfin, de rechercher une paire de messages valide à l'aide d'une méthode avancée d'accélération (modifications de message, bits neutres, etc.). Cependant, il ne sait pas à l'avance si une partie non linéaire peut être trouvée de manière efficace pour le vecteur de perturbation choisi, ou quel sera l'effet de l'accélération de recherche de paires valides suivant ce vecteur. Une telle stratégie est sous-optimale et il faut en pratique prendre en compte le plus de paramètres possible en même temps. C'est ce qui explique, pour SHA-1, le passage contre-intuitif par Wang *et al.* de l'attaque en 2^{69} appels à la fonction de compression [WYY05b] à l'attaque améliorée [WYY05a] de complexité 2^{63} : la première possède a priori un meilleur vecteur de perturbation que la seconde, mais l'accélération de recherche y est bien moins efficace. De plus, il est possible que le nombre de degrés de liberté dans le chemin différentiel final soit insuffisant pour trouver une collision, ce qui obligera l'attaquant à tout recommencer depuis le début. Le sens inverse d'utilisation des outils n'est pas plus efficace puisqu'il n'est pas évident de deviner à l'avance l'efficacité de l'accélération de recherche sans connaître le vecteur de perturbation ou la partie non linéaire. Une méthode de proche en proche semble pour l'instant être la meilleure approche possible. Par exemple, si après avoir choisi un vecteur de perturbation et engendré une partie non linéaire, on observe que l'accélération de recherche est difficile à mettre en oeuvre, on peut essayer d'engendrer d'autres parties non linéaires pour corriger ce problème. Si cela ne suffit toujours pas, il faudra sans doute changer le vecteur de perturbation ou modifier la technique d'accélération de recherche.

L'attaque comporte donc plusieurs parties à considérer et ne peut juste se réduire au chemin différentiel final. Nous avons alors quatre paramètres interdépendants qui influent sur la complexité totale de l'attaque : le vecteur de perturbation, les parties non linéaires, l'accélération de la recherche d'une paire de messages valide et la quantité de degrés de liberté disponibles dans le chemin différentiel final. Ceci est explicité dans la figure 6.1.

Le vecteur de perturbation est bien entendu la colonne vertébrale de l'attaque et détermine approximativement la complexité de celle-ci. Il nous oblige à fixer les positions de différences sur les mots de message et consomme donc au moins la moitié des degrés de liberté initialement disponibles (puisque décider si l'on souhaite ou non la présence d'une différence sur une

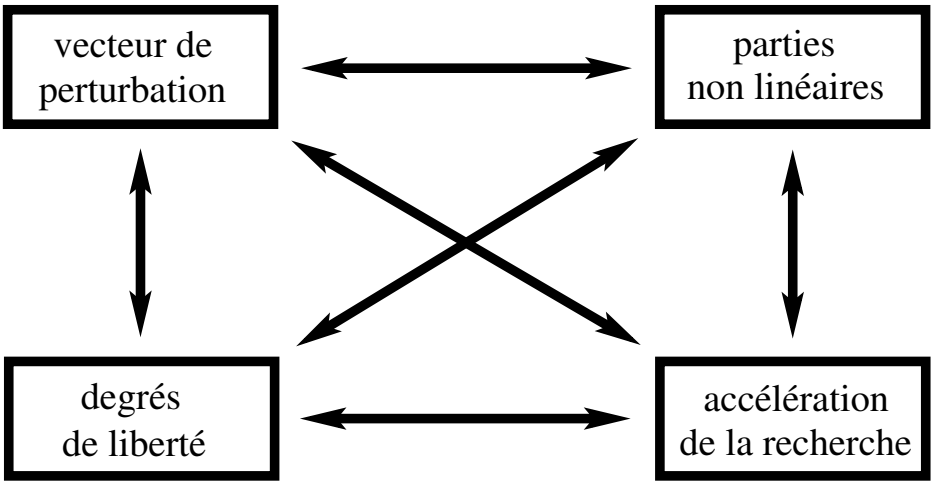

FIG. 6.1 – La cryptanalyse de SHA, un problème à plusieurs dimensions.

position de bit consomme un degrés de liberté). Il est assez difficile de prédire la possibilité de trouver une partie non linéaire étant donné un vecteur, la seule méthode étant concrètement de chercher cette partie. Concernant l'accélération de la recherche d'une paire de messages valide, on peut utiliser une heuristique naturelle en raisonnant sur le nombre de conditions présentes juste après la partie non linéaire. Ces conditions sont en effet celles susceptibles d'être contrôlées durant cette phase d'accélération, et l'on tentera d'estimer grossièrement leur nombre.

Comme expliqué précédemment, la partie non linéaire n'influe pas directement sur la complexité finale de l'attaque, mais elle est une condition sine qua none pour obtenir un chemin différentiel valide. En outre, on peut juger de la qualité de ce morceau de chemin différentiel : si la partie non linéaire impose beaucoup de conditions, on consommera beaucoup de degrés de liberté, essentiels pour obtenir une accélération de recherche puissante. Les degrés de liberté finaux doivent aussi être en nombre suffisant pour espérer trouver une collision avec le chemin différentiel final.

Comme nous le verrons dans les sections suivantes, l'accélération de la recherche existe sous de nombreuses formes, chacune de ces formes ayant des qualités et des défauts. On peut généralement observer qu'une accélération puissante consommera de nombreux degrés de liberté. Il faudra alors utiliser les formes adéquates et établir un compromis entre la rapidité de vérification du chemin différentiel final (le nombre de conditions) et la probabilité de trouver une collision avec ce chemin (probabilité égale à $N_e^{-1}(0)$ et donc directement dépendante des degrés de liberté disponibles) pour obtenir la meilleure complexité finale. En effet, on peut très bien imaginer une attaque où l'on engendre pour le même bloc plusieurs chemins différentiels différents à partir du même vecteur de perturbation : à partir du vecteur de perturbation considéré, on peut varier la manière de signer les perturbation ou encore engendrer plusieurs parties non linéaires différentes. Chacun de ces chemins aura une probabilité relativement faible d'aboutir à une collision à cause du manque de degrés de liberté, mais la vérification sera rapide grâce à l'accélération de recherche puissante. En engendrant plusieurs chemins, la somme de ces probabilités sera suffisante pour obtenir une collision. L'avantage ici serait d'obtenir une accélération très puissante (et demandant donc beaucoup de degrés de liberté) pour chacun des chemins différentiels. Dans ce cas, il faut bien entendu compter l'ensemble des étapes dans

la complexité finale, à savoir, pour chaque chemin différentiel : la génération des parties non linéaires, l'établissement de l'accélération de recherche et bien entendu la recherche finale d'une paire de messages valide. L'attaquant doit être capable de trouver en un temps relativement faible les parties non linéaires et les accélérations de recherche pour chaque chemin différentiel.

On s'aperçoit que les degrés de liberté sont les éléments essentiels à considérer pour optimiser l'attaque. Nous présentons dans les sections suivantes des méthodes avancées pour chaque étape de l'attaque, tout en gardant à l'esprit la problématique des degrés de liberté. Néanmoins, la meilleure technique à ce jour pour SHA-1 semble être une utilisation de proche en proche, tout en utilisant fortement l'intuition de l'attaquant. Dans le cas de SHA-0, les choses sont beaucoup plus simples puisque nous disposons d'une très grande quantité de degrés de liberté (grâce à la physionomie du chemin différentiel, borné à une seule position de bit).

Il reste un dernier point à clarifier : le calcul du coût final. Les attaques mettant en défaut la résistance à la recherche de collisions pour SHA-0 ou SHA-1 sont devenues très complexes, et leur coût total est assez difficile à évaluer de manière théorique. En général, la plupart des estimations théoriques diffèrent d'un facteur non négligeable des implantations. Ceci peut aussi être expliqué par une implantation non optimale. Une manière naturelle pour rigoureusement comparer deux attaques différentes, ou deux outils différents, serait de mesurer le nombre moyen d'appels à la fonction de compression en une unité de temps donnée et sur une plate-forme donnée, avec une bonne implantation de l'algorithme (pour cela, on utilisera par exemple *OpenSSL* [OpenSLL]). On pourra ainsi obtenir une comparaison assez précise du coût pratique en exécutant les attaques sur des plates-formes similaires. Dans la suite, nous parlerons de *complexité mesurée* lorsque nous utilisons cette méthode de comparaison.

6.2 Recherche de chemin : le vecteur de perturbation

Nous nous occupons dans cette section de trouver un vecteur de perturbation adéquat. Nous avons déjà expliqué des techniques avancées pour chercher un bon vecteur, mais nous essayons ici de montrer les différences entre ces techniques et la méthode de Wang *et al.*.

6.2.1 La technique de Wang *et al.*

L'algorithme de recherche de vecteurs de perturbation de Wang *et al.* est assez simple. On construit tous les vecteurs sur 16 étapes consécutives ayant des perturbations sur les positions de bit 0 ou 1 (2^{32} vecteurs possibles), puisque la position 1 est à privilégier pour des raisons de probabilités déjà expliquées. Chacun de ces petits vecteurs définit entièrement un vecteur de perturbation entier, grâce à la formule d'expansion de message. On considère toutes les fenêtres de 16 étapes consécutives ($80 - 16 = 64$ possibilités en tout), ce qui nous donne finalement 64×2^{32} candidats.

Pour chaque vecteur testé, on calcule de façon très simple les conditions du chemin différentiel induit : dans le cas de la fonction booléenne XOR, chaque perturbation sur la position de bit 1 induit 2 conditions et 4 conditions sinon. Pour la fonction booléenne IF (respectivement MAJ), on compte 5 conditions (respectivement 4) quelle que soit la position de bit considérée. On tient aussi compte des perturbations qui affectent deux fonctions booléennes différentes. Une compression de bits simple est considérée (seulement dans le cas de deux perturbations

adjacentes sur les positions de bit 0 et 1). On tient aussi compte des conditions qui peuvent être relaxées à la fin du chemin différentiel.

Enfin, parmi tous les candidats, Wang *et al.* choisissent le vecteur imposant le moins de conditions entre les étapes 16 et 79. Un autre vecteur fut proposé ultérieurement (il s'agit en fait du même vecteur décalé de 2 étapes), contenant plus de conditions, mais permettant une meilleure accélération de recherche d'une paire de messages valide.

6.2.2 Les techniques avancées

La technique décrite par Wang *et al.* est assez simple et donne des vecteurs de perturbation relativement bons. Il est néanmoins possible d'affiner cette recherche.

Une première amélioration simple consisterait à considérer une compression de bit généralisée, et non uniquement bornée aux positions de bit 0 et 1. L'attaquant peut aussi remarquer que les conditions comptées pour une perturbation introduite à la position de bit 31 sont trop nombreuses. Ceci est dû au fait que l'une des corrections de la perturbation s'effectue sur cette même position de bit, et l'effet de propagation de la retenue peut être relaxé. Ensuite, les cas où deux perturbations sont utilisées en même temps en entrée d'une fonction booléenne ne sont pas pris en compte. Ces cas permettent généralement une diminution du nombre de conditions. Enfin, Wang *et al.* ayant sous-estimé l'avantage pour un attaquant d'introduire des perturbations à la position de bit 31, on pourra étendre la recherche à cette position en plus de 0 et 1 dans les petits vecteurs sur 16 étapes. Ceci nous donne en tout 64×2^{48} vecteurs, mais on peut réduire ce nombre en ne testant que ceux dont le poids de Hamming n'est pas trop grand. Cette heuristique semble naturelle et n'ôte que des vecteurs comportant vraisemblablement beaucoup de conditions. Plus généralement, toutes ces améliorations peuvent être déduites des tableaux B.1 et B.2 en Appendice.

En programmant cette recherche, nous obtenons le tableau 6.1 pour SHA-0 et le tableau 6.2 pour SHA-1 qui résument les résultats en fonction de l'étape de départ considérée. On peut remarquer que ce programme nous fournit de meilleurs chemins différentiels que ceux de Wang *et al.*. Une première version de ces travaux a été publiée par Yajima *et al.* [YIN08a] puis nous avons participé à l'amélioration de ces résultats dans [YIN08b].

i	nombre de conditions
16	40 (39)
17	38 (38)
18	37,5 (36,5)
19	36 (36)
20	35,5 (34,5)
21	35 (34)
22	33,5 (32,5)
23	33 (32)

i	nombre de conditions
24	32 (31)
25	32 (31)
26	32 (31)
27	30 (30)
28	30,5 (29,5)
29	30 (29)
30	29 (28)
31	29,5 (27,5)

TAB. 6.1 – Nombre de conditions suivant l'étape i de début de comptage dans le cas de SHA-0. Les nombres entre parenthèses tiennent compte des conditions qui peuvent être relaxées à la fin du chemin différentiel (c'est le nombre réel de conditions pour le premier bloc).

i	nombre de conditions
16	87 (85)
17	83,5 (81,5)
18	79,5 (77,5)
19	76,5 (73,5)
20	74 (71)
21	72 (69)
22	70,5 (67,5)
23	67,5 (64,5)

i	nombre de conditions
24	64,5 (61,5)
25	62 (59)
26	59 (57)
27	56,5 (54,5)
28	53,5 (51,5)
29	50,5 (48,5)
30	48 (46)
31	45,5 (43,5)

TAB. 6.2 – Nombre de conditions suivant l'étape i de début de comptage dans le cas de SHA-1. Les nombres entre parenthèses tiennent compte des conditions qui peuvent être relaxées à la fin du chemin différentiel (c'est le nombre réel de conditions pour le premier bloc).

6.3 Recherche de chemin : la partie non linéaire

L'une des composantes les plus importantes des attaques de Wang *et al.* [WYY05d, WYY05b, WYY05a, WYY05c] concerne la génération de la partie non linéaire. Hélas, ces recherches ayant été réalisées de façon non automatisée, grâce à un long et patient travail des chercheurs, toute réutilisation des chemins différentiels de Wang *et al.* est impossible à moins d'accomplir une nouvelle fois cette tâche fastidieuse. Il est donc important de pouvoir engendrer des parties non linéaires de manière automatisée.

Dans cette section, nous décrivons les travaux de De Cannière et Rechberger [CR06] pour SHA-0 ou SHA-1, permettant de construire des parties non linéaires de façon automatisée et assez rapide, étant donné un vecteur de perturbation et une variable de chaînage d'entrée initiale. Grâce à ces travaux ont pu être engendrées une collision pour SHA-1 réduit à 64 étapes [CR06], puis une collision sur une version réduite à 70 étapes [CMR07]. L'article original ne fournit cependant que très peu de détails, insuffisants pour permettre une implantation par le lecteur. Nous donnons donc ici tous les détails nécessaires à la programmation de cet outil très utile et efficace à la fois pour SHA-0 et SHA-1. La rapidité d'exécution de l'algorithme étant un critère très important, nous proposons également des méthodes pour améliorer ce paramètre. Il faut noter que nous explicitons ici notre propre implantation, ce qui peut se traduire par quelques différences en comparaison des travaux originaux de De Cannière *et al.*

Même si cet algorithme constitue la meilleure approche actuelle, on peut citer d'autres avancées dans le domaine des outils automatisés pour la recherche de parties non linéaires, par exemple le travail de Yajima *et al.* [YSN07].

Pour construire des parties non linéaires d'un chemin différentiel, nous aurons besoin de deux procédures. Ces procédures devront être très rapides à exécuter. Nous consacrons les deux sections suivantes à cette problématique. Nous examinerons ensuite comment l'algorithme général utilise ces procédures pour trouver des parties non linéaires de manière heuristique.

6.3.1 Calcul efficace de probabilité pour un chemin différentiel

Tout d'abord, nous devons être en mesure de calculer efficacement et exactement la probabilité incontrôlée $P_i(i)$ pour chaque étape i, et ce, de manière très rapide et pour n'importe quel type de chemin différentiel. Une manière très naïve et très lente de calculer $P_i(i)$ serait de parcourir toutes les possibilités suivant les conditions imposées sur les mots d'état interne $A_{i-4}, \ldots, A_i$ et sur le mot de message W_i, puis de compter les candidats qui vérifient les conditions sur le mot de sortie A_{i+1}. Par exemple, considérons le tronçon de chemin différentiel très simple de la figure 6.2. On testera tous les mots possibles pour le registre A_{i-4}, au nombre de 2^{32} puisque chaque bit contient la condition « aucune différence » dénotée par le caractère « - ». Pour chaque candidat de A_{i-4}, on testera toutes les valeurs possibles pour A_{i-3} (2^{31} candidats), puis A_{i-2} (2^{31} candidats), etc. Pour chaque combinaison, on regarde si les contraintes sur A_{i+1} sont vérifiées. On s'aperçoit que la complexité devient très vite rédhibitoire.

i	∇A_i	∇W_i
	...	...
$i-4$	`--------------------------------`	
$i-3$	`-----------------------------1---`	
$i-2$	`-----------------------------0---`	
$i-1$	`--------------------------------`	
i	`--------------------------------`	`------------------------------x-`
$i+1$	`------------------------------x-`	
	...	...

termes	contraintes
A_{i-3}	`------------------------------1-`
A_{i-2}	`------------------------------0-`
A_{i-1}	`--------------------------------`
$\Phi_i(A_{i-1}, A_{i-2}, A_{i-3})$	`--------------------------------`
A_{i-4}	`--------------------------------`
A_i	`--------------------------------`
W_i	`------------------------------x-`
K_i	`01101110110110011110101110100001`
A_{i+1}	`------------------------------x-`

FIG. 6.2 – Un exemple de calcul de $P_i(i)$, en supposant que l'étape i appartienne au deuxième tour. Le tableau du bas prend en compte les rotations de la formule de mise à jour du registre interne A_{i+1}.

Cette méthode étant bien trop coûteuse, nous l'améliorons en faisant un calcul position de bit par position de bit. Pour cela, nous allons devoir considérer la retenue qui se propage entre chaque position durant l'addition. Cette retenue ne pourra atteindre qu'un nombre limité de valeurs qu'il nous faut déterminer au préalable. Nous avons à faire à une addition à 5 termes : deux mots d'état interne A_i et A_{i-4}, la sortie de la fonction booléenne (prenant en entrée A_{i-1}, A_{i-2} et A_{i-3}), le mot de message W_i et la constante K_i. Si pour une position de bit donnée tous ces éléments sont à 1, le total sera 5 et nous aurons donc une retenue sortante égale à 2 (la retenue initiale étant nulle). Si pour la position de bit suivante nous avons toujours tous

les éléments à 1, le total sera 7 à cause de la nouvelle retenue arrivant et nous aurons par conséquent une retenue sortante égale à 3. Nous continuons, ainsi de suite, la prochaine étape aboutissant à un total de 8, la retenue sortante étant égale à 4. Enfin, le total suivant est de 9 et la retenue reste égale à 4. Finalement, nous avons montré que la retenue ne pouvait prendre que des valeurs situées entre 0 et 4 compris.

Nous allons calculer la probabilité $P_i(i)$ comme le produit des probabilités conditionnelles de succès pour chaque position de bit. Nous allons donc aussi maintenir un tableau de probabilités concernant la retenue entre chaque position de bit. Ce tableau sera composé pour chaque position de bit d'une matrice 5×5 : il existe 5 valeurs de retenue possibles et il faut prendre en compte des transitions puisque l'on a à faire à un chemin différentiel (même si certaines transitions de retenue sont impossibles, par exemple de 4 vers 0). On commence par la position 0, en considérant bien entendu que la retenue arrivant en cette position est nulle. On parcourt alors toutes les possibilités pour chaque bit concerné, suivant les contraintes du chemin différentiel, et l'on compte le nombre de candidats qui vérifient la condition sur le bit de sortie visé. On peut de cette façon rapidement calculer la probabilité de succès pour cette position de bit, ainsi que la matrice de probabilité de la retenue sortante, en ne prenant en compte que les candidats valides (ceux qui sont invalides ne doivent pas influer sur les probabilités de la matrice de retenue sortante). On continue à la position suivante en prenant maintenant en compte les probabilités de la matrice de retenue entrante et ainsi de suite jusqu'au bit 31.

Pour illustrer cette méthode, prenons l'exemple de la figure 6.2, où le tableau du bas est le plus lisible du fait que nous y avons incorporé les rotations des registres internes durant la mise à jour de A_{i+1}. En commençant à la position de bit 0, le calcul est très simple puisque nous avons uniquement des contraintes « aucune différence » sur tous les termes de l'addition. Ainsi, nous avons une probabilité égale à 1 de n'avoir aucune différence en sortie sur le bit 0 de A_{i+1}, ce qui est bien attendu dans notre chemin différentiel (si nous avions la contrainte « 1 » sur le bit 0 de A_{i+1}, la probabilité de succès aurait été de $1/2$). Pour ce qui concerne la matrice de retenue sortante, sans expliciter le calcul exact, nous avons une probabilité de $1/16$ d'avoir la transition $0 \rightarrow 0$, $10/16$ d'avoir $1 \rightarrow 1$, $5/16$ d'avoir $2 \rightarrow 2$, et une probabilité nulle pour les 22 transitions restantes (puisqu'aucune différence n'est présente en entrée, la retenue ne change jamais). Pour la position de bit 1, nous avons une différence en entrée de A_i et une différence attendue en sortie sur A_{i+1}. On peut facilement s'apercevoir que cela se produit avec probabilité 1, mais il faudra tout de même parcourir tous les cas pour calculer la nouvelle matrice de retenue sortante. En effet, la retenue en sortie aura cette fois une probabilité de $1/2$ de contenir une différence (une transition $i \rightarrow j$ avec $i \neq j$), selon que nous ayons une propagation de retenue ou non. Du fait que pour la position de bit 2 nous ne souhaitons aucune différence sur A_{i+1} et puisqu'aucune différence n'apparaît sur les entrées, nous avons une probabilité de succès égale à $1/2$ pour cette position de bit. Seules les instances valides (celles ne propageant pas la différence sur la retenue) seront considérées pour le calcul de la nouvelle matrice de probabilités de retenue. Cette matrice comporte de ce fait des probabilités nulles pour toutes les transitions $i \rightarrow j$ avec $i \neq j$. Finalement, tous les cas des autres positions de bit sont très simples parce que plus aucune différence n'apparaît ni dans les entrées, ni dans les retenues et nous attendons constamment la condition « aucune différence » sur A_{i+1}, ce qui se produit toujours avec probabilité 1. Nous pouvons conclure que $P_i(i) = 1/2$.

Cette fonction de calcul de $P_i(i)$ étant cruciale et fréquemment utilisée, pour obtenir une bonne vitesse d'exécution il est très important durant l'implantation d'éviter de recalculer une valeur déjà calculée auparavant, en n'hésitant pas pour ce faire à stocker des résultats. Pour accélérer le calcul, il est recommandé de précalculer et de stocker dans des tableaux, suivant les

entrées possibles pour une position de bit, les sorties possibles correspondantes pour le registre A_{i+1} et la retenue sortante. Enfin, on peut utiliser certaines symétries existantes, par exemple entre les mots de registre interne dans les fonctions booléennes MAJ et XOR, ou encore entre A_i et A_{i-4} durant l'addition.

6.3.2 Calcul efficace de raffinage de conditions

La seconde procédure dont nous avons besoin est un algorithme de raffinage des conditions. Par raffinage, nous entendons la détermination de conditions non encore établies et nécessaires au bon déroulement du chemin différentiel, sur les registres internes ou sur les mots de message. Prenons par exemple le cas de la figure 6.3, qui est très similaire à celui de la figure 6.2. Nous y avons relaxé un certain nombre de conditions sur le mot de registre interne A_{i+1} en ajoutant des « ? ». L'analyse précédente nous a montré que nous avons deux possibilités équiprobables : soit la différence présente sur la position de bit 1 dans W_i provoque une propagation dans la retenue lors du calcul de l'addition, soit elle reste bornée à la position de bit 1 de A_{i+1}. Dans les deux cas, nous nous apercevons qu'une différence existera nécessairement sur le bit 1 de A_{i+1} et qu'aucune différence ne sera présente sur le bit 0. De ce fait, nous pouvons raffiner ces conditions en remplaçant les « ? » par un « - » pour le bit 0 et par un « x » pour le bit 1 dans le chemin différentiel de la figure 6.3. Pour le reste des positions de bit, rien ne peut être déduit.

i	∇A_i	∇W_i
	...	...
$i-4$	----------	
$i-3$	----------1---	
$i-2$	----------0---	
$i-1$	----------	
i	----------	----------x-
$i+1$	??????????	
	...	...

i	∇A_i	∇W_i
	...	...
$i-4$	----------	
$i-3$	----------1---	
$i-2$	----------0---	
$i-1$	----------	
i	----------	----------x-
$i+1$	??????????x-	
	...	...

FIG. 6.3 – Un exemple de raffinage. Le premier chemin n'est pas raffiné et nous pouvons trouver deux nouvelles conditions sur les bits 0 et 1 de A_{i+1}. Nous obtenons ainsi une caractéristique raffinée pour le deuxième chemin différentiel.

Comment maintenant implanter une telle fonction dans des cas beaucoup plus complexes que celui de la figure 6.3 ? En premier lieu, nous allons maintenir un grand tableau, que nous appellerons *tableau de raffinage*, indiquant pour chaque bit de chaque étape si un raffinage à cette position de bit est nécessaire ou non (cela peut être implanté grâce à un tableau de 80×32 variables booléennes, une entrée est égale à 1 pour *à raffiner* et à 0 pour *déjà raffiné*). Nous étendons

le chemin différentiel en considérant aussi les matrices de retenue entre chaque position de bit durant l'addition. En effet, nous pouvons aussi raffiner les conditions sur les retenues, même si cela n'est pas visible directement sur le chemin différentiel que nous affichons. L'algorithme est très simple : nous parcourons tous les bits de toutes les étapes et nous vérifions qu'aucun bit du tableau de raffinage n'est à 1. Si nous rencontrons un tel bit, nous devons analyser cette position et agir en conséquence. Soit aucune nouvelle condition ne peut être déduite et nous modifions le bit correspondant du tableau de raffinage à 0 pour indiquer qu'aucun raffinage n'est possible, soit nous trouvons une nouvelle condition et nous l'appliquons au chemin différentiel. Cependant, tout ajout d'une nouvelle condition impose à l'algorithme de revérifier certains bits puisque l'état a changé (de nouveaux bits du tableau de raffinage sont donc à 1). Tout le problème est d'analyser en détail quels nouveaux bits peuvent potentiellement être influencés par cette modification pour obtenir une bonne rapidité d'exécution de l'algorithme. Par exemple, revérifier toutes les positions de bit possibles pour toutes les étapes possibles lorsque nous n'avons ajouté qu'une seule condition représente un gaspillage d'opérations.

Nous pouvons déduire assez facilement les règles de mise à jour suivantes, en fonction du type de condition modifiée :

- **si nous modifions une condition de l'état interne à la position j de l'étape i, soit A_i^j :** nous devons raffiner six positions, à savoir le bit j des étapes i et $i+2$, le bit $j+5$ de l'étape $i+1$, et enfin les bits $j-2$ des étapes $i+3$, $i+4$ et $i+5$.
- **si nous modifions une condition du mot de message à la position j de l'étape i, soit W_i^j :** nous devons raffiner à nouveau la même position, à savoir le bit j de l'étape i.
- **si nous modifions une condition de la matrice de retenue entre les positions j et $j+1$ de l'étape i (matrice entrante pour la position de bit $j+1$, matrice sortante pour la position de bit j) :** nous devons raffiner deux positions, à savoir les bits j et $j+1$ de l'étape i.

Il faut noter que les matrices de retenue sortantes des positions de bit 31 peuvent être oubliées puisque l'addition s'opère modulo 2^{32}. De même, les matrices de retenue entrantes à la position de bit 0 ne peuvent être raffinées du fait que la retenue est toujours nulle dans ce cas.

Une fois ces règles de mise à jour définies, il nous reste à décrire la fonction qui raffine une position de bit si besoin est. Pour cela, nous pouvons utiliser une version légèrement modifiée de l'algorithme de calcul de probabilité incontrôlée $P_i(i)$ de la section précédente. Pour une position de bit donnée j à une étape i, nous avons en entrée les conditions déduites du chemin différentiel sur le bit j du mot de message W_i et les bits correspondants des mots de l'état interne A_{i-4} à A_i (nous avons aussi les contraintes sur la matrice de retenue entrante à la position j). Nous allons simplement tester toutes les possibilités qui vérifient ces conditions d'entrée et retenir celles compatibles avec les contraintes imposées en sortie sur le bit j de A_{i+1} et sur la matrice de retenue sortante. Nous pouvons alors déduire plusieurs informations. Tout d'abord, en ce qui concerne le raffinage des sorties, l'espace de sortie atteint peut être différent de celui défini par la condition sur le bit j de A_{i+1}. Si tel est le cas, soit les espaces sont disjoints (ce qui implique une impossibilité et donc un chemin différentiel invalide), soit nous pouvons préciser la condition sur le bit j de A_{i+1} (et donc ensuite mettre à jour les nouveaux bits à vérifier suivant les règles établies précédemment). De plus, nous pouvons aussi raffiner exactement de la même manière la matrice de retenue sortante à la position j. Pour ce qui concerne les entrées, nous pouvons examiner quel espace a permis d'atteindre de façon valide la condition sur le bit j de A_{i+1} et celles sur la matrice de retenue sortante à la position j. Cela nous permettra de préciser les contraintes sur les entrées si nécessaire (puis de mettre à jour les nouveaux bits à

vérifier le cas échéant), ou d'invalider le chemin différentiel si nous avons affaire à des espaces disjoints. Cette méthode est présentée graphiquement dans la figure 6.4. Il faut être attentif durant l'implantation à moduler le raffinage suivant l'étape dans laquelle on se trouve, puisque la formule de mise à jour est différente suivant le tour considéré.

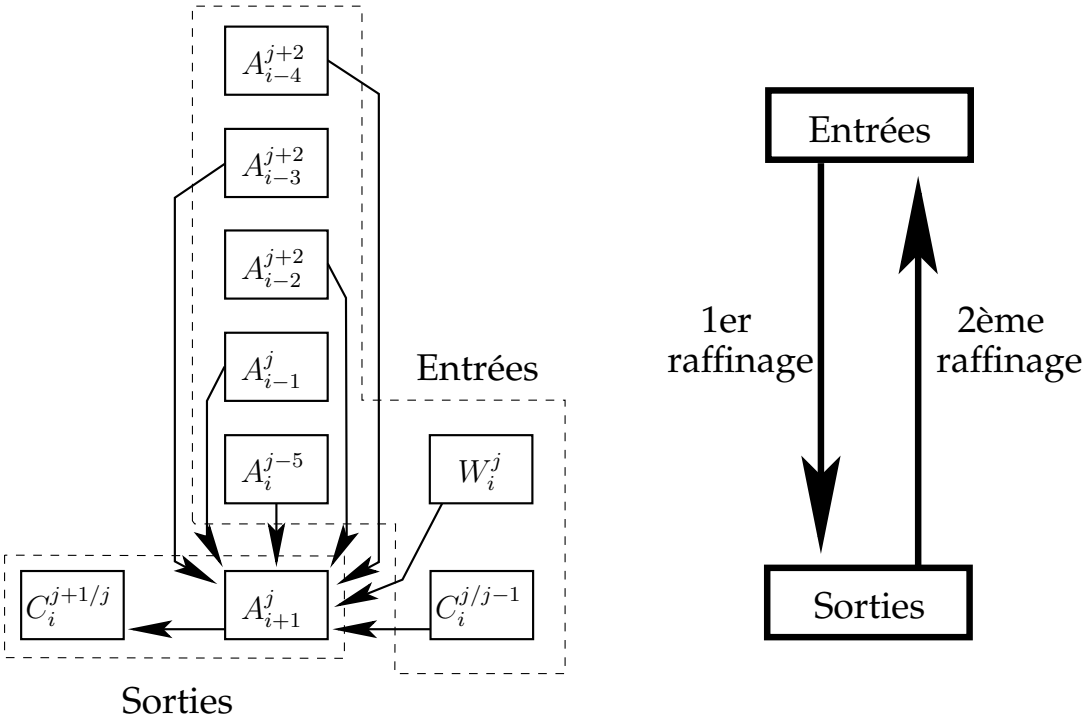

FIG. 6.4 – Technique de raffinage pour une position j de l'étape i. Une fois l'ensemble des espaces possibles parcourus et ceux valides déduits, on raffine les sorties (le bit j de A_{i+1} et la matrice de retenue sortante notée $C_i^{j+1/j}$) et les entrées (les bits du message W_i^j, les bits correspondants des mots de l'état interne A_{i-4} à A_i et la matrice de retenue entrante notée $C_i^{j/j-1}$). Chaque modification déduite (si cela se produit) doit être suivie de la mise à jour des nouveaux bits à raffiner dans le tableau de raffinage.

Nous avons ainsi entièrement décrit une technique qui permet de trouver des conditions non précédemment déduites sur un chemin différentiel : chaque bit égal à 1 du tableau de raffinage sera traité (plus d'autres si des modifications ont eu lieu), et le tableau de raffinage en sortie de l'algorithme sera entièrement nul. La procédure proposée ne permet de raffiner qu'à un degré 1 puisqu'on ne cherche pas à établir d'autres conditions plus complexes qui pourraient être déduites si l'on augmentait le champ de vision de l'algorithme. Même si un raffinage de degrés supérieur était possible à implanter (quoiqu'assez complexe), son coût serait excessif compte tenu de l'utilisation que nous voulons en faire. La méthode de raffinage proposée présente l'avantage d'être assez simple d'implantation, mais également très rapide.

6.3.3 Structure de l'algorithme

Nous disposons à présent d'un algorithme qui calcule la probabilité incontrôlée $P_i(i)$, mais aussi et surtout d'une fonction permettant le raffinage de conditions non précédemment dé-

duites. Cette dernière nous renvoie de plus une impossibilité si un chemin différentiel possède des conditions contradictoires.

Avant de décrire le coeur de l'algorithme de recherche de parties non linéaires, nous devons préparer le chemin différentiel de départ. Nous avons en entrée de l'algorithme un vecteur de perturbation et une valeur de variable de chaînage (pouvant potentiellement comporter des différences). Nous créons donc en premier lieu le squelette du chemin différentiel grâce à ces deux entrées, mais nous allons relaxer toutes les conditions sur les registres internes entre A_0 et A_k. Ce paramètre k dépend grandement du vecteur de perturbation et devra être déterminé empiriquement par l'attaquant pour faciliter la tâche de l'algorithme. Bien entendu, nous ne modifions pas les conditions sur les mots de message entre ces étapes puisque la partie non linéaire ne concernera que les registres internes. En outre, nous avons observé qu'en relaxant légèrement les conditions sur les positions de bit de poids très faible ou très fort des registres internes après l'étape k, l'algorithme présente un meilleur comportement. Une fois ce premier squelette créé, nous appelons la fonction de raffinage pour aboutir à un chemin différentiel CD_{depart} qui sera notre point de départ. Nous donnons un exemple d'un tel squelette dans la figure 6.5.

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010110111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	xxx-x----------	32	0.00	-244.40
01 :	????x----------	----x----------xx--	32	0.00	-212.40
02 :	???????????????????????????????x	---xx----------	32	0.00	-180.40
03 :	????????????????????????????????	x-xxx----------x-x-	32	0.00	-148.40
04 :	????????????????????????????????	xx--x----------x----	32	0.00	-116.40
05 :	????????????????????????????????	--x-xx----------x----	32	0.00	-84.40
06 :	????????????????????????????????	xxxx-x----------x-x--	32	0.00	-52.40
07 :	????????????????????????????????	x-xx-x----------x----	32	0.00	-20.40
08 :	????????????????????????????????	----x----------	32	0.00	11.60
09 :	????????????????????????????????	x--xx----------xx--	32	0.00	43.60
10 :	????????????????????????????????	xx-xx----------x----	32	0.00	75.60
11 :	????????????????????????????????	----x----------x----	32	-16.00	107.60
12 :	???----------??????????????	x-xxx----------x----	32	-18.00	123.60
13 :	???----------??????????	x--xx----------	32	-20.00	137.60
14 :	???----------?????????	-xx----------	32	-22.00	149.60
15 :	???----------???????	----------x---	32	-25.00	159.60
16 :	??----------?????	xx----------	0	-12.28	166.60
17 :	??x----------??	x--x----------x-x--	0	-11.16	154.31
18 :	??----------?	---x----------x----	0	-9.33	143.15
	...	...			

FIG. 6.5 – Un exemple de squelette de départ CD_{depart}, avec $k = 11$. Nous ne montrons ici qu'un tronçon du chemin différentiel total étant donné que la partie non linéaire se situe dans les premières étapes. On peut observer que les conditions sur le registre A_1 ont été raffinées puisqu'au départ ce registre était entièrement non contraint. Nous avons remarqué empiriquement que la forme en triangle des conditions relaxées après l'étape k donne de bons résultats.

Le coeur de l'algorithme est alors très simple : nous allons tirer aléatoirement un bit des registres internes entre les étapes 0 et k, et réagir suivant sa valeur. Si ce bit ne possède aucune condition (« ? »), nous imposons qu'aucune différence n'y soit présente (« - »), nous mettons à 1 le bit du tableau de raffinage correspondant à la modification et nous raffinons le chemin différentiel. Si ce bit possède déjà une différence (« x »), nous forçons son signe en tirant aléatoirement une différence montante (« n ») ou descendante (« u »), ensuite nous mettons à jour le tableau de raffinage et nous raffinons le chemin. Pour tout autre type de condition, nous ne faisons rien. On poursuit de la même manière, et si l'on rencontre une impossibilité à un moment donné, on peut revenir une étape en arrière en annulant la précédente modification (ou

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010111001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	xnn0u1111010111110----------0	14	-2.00	-60.68
01 :	-0u0u111011001000-------0-0----1	-1--n1001100----------uu10	19	-16.07	-48.68
02 :	-01-1nnnnnnnnnnnnnnnnnnnn-nnnn	-1-nu-11111111----------0	20	-4.00	-45.75
03 :	-n-uu-010000111-------u---u---00	x-nuu001101----------u-x-	22	-12.42	-29.75
04 :	x---10111101n010u-----????-----E	xx--n----------x----	31	-20.42	-20.16
05 :	-x-0-00------un-nn-nnD-?--?-?---	--x-xn----------x----	31	-17.43	-9.58
06 :	--x0----------00n-n---???-?---D-	xxxx-n----------x-x--	31	-16.65	4.00
07 :	---01------n---01B5x---B?-D-B??x	x-xx-n----------x---	31	-22.51	18.35
08 :	n-n-0--------0----0-u0--u-?-?---	----x----------	32	-21.36	26.84
09 :	-n--0--------0-------1--------x--	x--xx----------xx--	32	-16.82	37.48
10 :	--010------------Bx---u---??---?	xx-xx----------x----	32	-21.69	52.66
11 :	--u--------------Bx-0---Ex-u-n-0	----n----------x----	31	-19.85	62.98
12 :	----1--------------------1-n--uu1	x-xxx----------u----	31	-16.02	74.13
13 :	?-n-1----------n-----n	x--un----------	30	-8.74	89.11
14 :	x1--1--------------------DB--Dx?0	-xx----------	32	-15.49	110.37
15 :	?-n-1----------n-----0	----------x--0	31	-8.53	126.88
16 :	-x--1----------?1	xx----------	0	-9.50	149.36
17 :	-?u11----------x	x--x----------x-x--	0	-8.68	139.86
18 :	----0----------	---x----------x----	0	-5.63	131.18
	...	...			

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010111011001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	xnn0u1111010111110001111--01---0	6	-3.00	-57.31
01 :	-0u0u111011001000010100000-0010-1	-1--n10011001010001--001----uu10	9	-6.00	-54.31
02 :	-01-1nnnnnnnnnnnnnnnnnnnnn-nnnn	-1-nu-1111111110100----1001-1--0	10	-7.00	-51.31
03 :	-n-uu-0100001111001100u010u11000	x-nuu00110100101011011101--u-n0	5	-3.00	-48.31
04 :	n1--10111101n010u01110uu1n1---1n	xx--n--------00-1--------10n----	25	-14.09	-46.31
05 :	-u-0-00------un-nn0nnn0u11u0u---	--u-xn--------10000--------x----	25	-13.00	-35.40
06 :	--u0----------00n1n1-1n1u--01-0-	xuux-n----------1-----1----u-u--	25	-18.00	-23.40
07 :	---01------n---01-0u101u00n00u1u	x-xx-n----------0---n---	29	-16.42	-16.40
08 :	n0n-0--------0----0-u0--u1n1n1-1	----u----------0----------	30	-15.00	-3.82
09 :	-n1-0--------0------100110-0-u01	x--xn----------1---xx--	30	-16.42	11.18
10 :	--010------------un---u001u0011n	xn-nu----------x----	29	-14.00	24.77
11 :	01u--------------un00---1n-u1n-0	----n----------1u1---	28	-9.19	39.77
12 :	01--1--------------------1-n10uu1	x-nxx----------u---0	29	-14.42	58.58
13 :	1-n-1--------------11----n00110n	x--un----------1	29	-9.00	73.16
14 :	u1--1----------nu-1nun0	-un-1----------1	28	-9.00	93.16
15 :	n1n-1----------n--1110	----1----------n0-0	28	-9.42	112.16
16 :	1n1-1----------10-0u1	xn-1----------0	0	-7.42	130.75
17 :	-uu11----------1---n	x--n----------u-u--	0	-1.00	123.33
18 :	----0----------	---u----------u----	0	0.00	122.33
	...	...			

FIG. 6.6 – Un exemple de sortie de l'algorithme de recherche de parties non linéaires, en partant du squelette CD_{depart} de la figure 6.5. Le premier chemin représente la sortie avant la recherche exhaustive avec $l = 25$, d'où la présence de bits non contraints et de différences non signées dans les registres internes entre les étapes 0 et $k = 17$. Le deuxième chemin représente la sortie finale CD_{final} après application de la recherche exhaustive.

plusieurs si un retour avait déjà été effectué précédemment) puis en continuant normalement (les états successifs devront ainsi être stockés). On limitera cependant le nombre de retours à une certaine quantité r, déterminée empiriquement par l'attaquant. On arrête lorsque le chemin raffiné comporte au plus l bits des registres internes sans aucune condition (« ? »), le nombre l étant aussi déterminé empiriquement par l'attaquant. On aboutit à un chemin différentiel temporaire CD^i_{temp}, pour lequel nous allons tester presque exhaustivement les combinaisons possibles des bits qui restent sans condition (ou avec une différence non signée), et chercher un candidat débouchant sur un chemin différentiel valide (ne renvoyant pas d'impossibilité après le raffinage final). Si l'on aboutit à un tel candidat, nous avons trouvé une partie non linéaire CD_{final} ; sinon on réitère l'algorithme en partant de CD_{depart}. Pour plus de clarté, cette méthode est présentée sous forme algorithmique dans la figure 6.7. Nous donnons aussi un

exemple d'exécution de l'algorithme dans la figure 6.6

```
Entrées : k, r, l et CD_depart
Sorties : CD_final
tant que (vrai) faire
    prof_temp = 0;
    r_temp = 0;
    ChargeChemindepart(CD_depart);
    tant que ((r_temp<r) et (NombreBitsNonContraints()>l)) faire
        bit = ChoisitBitAleatoirement(k);
        prof_temp = prof_temp + 1;
        SauveChemin(prof_temp);
        impossibilité = faux;
        suivant (bit.valeur) faire
            cas où « ? »
                bit.valeur = « - » ;
                TableauRaffinage[bit] = 1;
                impossibilité = RaffineChemin();
            fin
            cas où « x »
                bit.valeur = ChoisitAleatoire(« u »,« n »);
                TableauRaffinage[bit] = 1;
                impossibilité = RaffineChemin();
            fin
        fin
        si (impossibilité == vrai) alors
            prof_temp = prof_temp - 1;
            ChargeChemin(prof_temp);
            r_temp = r_temp + 1;
        fin
    fin
    si (impossibilité == faux) alors
        si (RechercheExhaustive() == vrai) alors
            retourner solution;
        fin
    fin
fin
```

FIG. 6.7 – Algorithme de recherche de parties non linéaires. La fonction `ChoisitBitAleatoirement`(k) choisit aléatoirement un bit entre les étapes 0 et k. La fonction `NombreBitsNonContraints`() renvoie le nombre de bits sans contrainte (« ? ») dans le chemin différentiel. La fonction `ChoisitAleatoire`(a,b) renvoie a avec probabilité $1/2$ et b avec probabilité $1/2$. La fonction `RaffineChemin`() raffine le chemin différentiel suivant le tableau de raffinage et réinitialise à zéro ce tableau. Enfin, la fonction `RechercheExhaustive`() parcourt exhaustivement les candidats possibles (nous imposons que « ? » puisse devenir « - », « n » ou « u » ; et que « x » puisse devenir « n » ou « u ») et renvoie vrai si une solution valide après raffinage est trouvée.

Bien que cet outil soit entièrement heuristique, nous pouvons tenter d'analyser qualitativement quelques étapes. Premièrement, le fait de forcer des bits non contraints (« ? ») à des conditions de non-différence augmente les chances d'aboutir à une partie non linéaire possédant un faible nombre de différences. Nous avons déjà observé qu'un faible nombre de différences augmente la probabilité de succès d'un chemin différentiel, et donc, dans notre cas, accroît les chances de trouver une partie non linéaire. Ensuite, le fait d'imposer un signe si l'on rencontre une différence permet de rapidement entrer dans un espace de solutions plus petit, car de nombreuses conditions seront imposées par le raffinage qui suit cette modification. Cela aura pour effet de plus rapidement se rendre compte si l'on évolue dans un espace sans solutions. Il est assez facile de trouver une partie non linéaire non signée, mais dès que l'on tente de forcer les signes des différences (ce qui doit être fait de toute manière avant la recherche d'une paire de messages valide), on aboutit fréquemment à une impossibilité. Ceci s'explique par le fait que notre raffinage ne s'applique qu'à un degré 1, et certaines conditions contradictoires très complexes ne peuvent être repérées facilement. Imposer un signe aux différences permet ainsi d'éviter de continuer à chercher dans un chemin où des conditions contradictoires très complexes existent, sans être découvertes. Limiter à r le nombre de retours empêche l'algorithme de boucler indéfiniment lorsqu'il se trouve dans un sous-espace sans solutions. Choisir un r très grand laissera aboutira à demeurer trop longtemps dans des sous-espaces sans solutions, utiliser un r très petit à ne pas tolérer assez de tentatives pour qu'une solution soit trouvée. Si l'on arrive à un chemin temporaire CD^i_{temp}, nous avons une probabilité non négligeable qu'une solution puisse en être extraite. Pour cette raison, nous nous arrêtons lorsqu'il reste moins de l bits non contraints et nous recherchons presque exhaustivement l'hypothétique solution pour éviter tout risque de la manquer. Ne pas utiliser cette amélioration revient à demander à l'algorithme de chercher la solution complète, tout en sachant qu'il s'arrêtera après r erreurs. On peut enfin remarquer que l ne doit pas être choisi trop grand sous peine de ne pas pouvoir exécuter la recherche exhaustive.

En pratique, nous avons observé que le jeu de paramètres $k = 11$, $r = 150$ et $l = 70$ donne un bon résultat. En pratique, nous ne testons pas 2^l cas puisqu'en parcourant l'ensemble des candidats sous forme d'un arbre il est possible de couper un grand nombre de branches, ce qui accélère grandement la recherche. Réitérer l'algorithme avec des légères variations pour ces valeurs peut permettre d'augmenter la probabilité de succès. Bien entendu, de nombreuses améliorations sont sans doute possibles, par exemple traiter les cas où l'on choisit un bit qui possède des conditions de type « 3 », « 5 », « 7 », « A », etc., au lieu de ne rien faire. Aussi, il semble indispensable de mémoriser les conditions raffinées sur les matrices de retenue, pour éviter de devoir les recalculer si l'on opère un retour à cause d'une impossibilité.

Le lecteur avisé remarquera que nous n'avons pas utilisé la fonction de calcul de probabilité incontrôlée $P_i(i)$. Nous utilisons en fait cette fonction pour parfaire notre partie non linéaire finale. En essayant aléatoirement de fixer des conditions sur CD_{final}, nous pouvons garder uniquement celles qui améliorent grandement la probabilité de succès de la partie non linéaire. On pourra continuer à améliorer le chemin différentiel de cette manière tant qu'assez de degrés de liberté sont disponibles.

Nous pouvons enfin observer qu'il est facile d'inclure des conditions dans le chemin différentiel de départ CD_{depart}, puis de chercher une partie non linéaire qui prenne en compte ces conditions. Cela sera un outil indispensable pour la section suivante, qui traite des attaques boomerang.

6.4 Recherche de candidats valides : les attaques boomerang

L'accélération de la recherche de candidats valides est primordiale pour la rapidité de l'attaque finale. Les bits neutres et les modifications de message sont déjà des techniques avancées, mais de nouvelles méthodes plus sophistiquées sont apparues, comme les modifications sous-marines introduites par Naito *et al.* [NSS06] ou les tunnels de Klima [Kli06]. Nous montrons dans cette section que l'attaque boomerang [Wag99], qui était originalement un outil de cryptanalyse pour les algorithmes de chiffrement par blocs, peut être transposée au contexte des fonctions de hachage pour donner une généralisation des techniques d'accélération de la recherche de candidats valides. Cette généralisation permet tout d'abord d'améliorer la compréhension des concepts sous-jacents à ces techniques. De plus, elle aboutit à de nouvelles variantes très intéressantes d'accélération de la recherche de collisions. Ces résultats sont le fruit de notre coopération avec Antoine Joux dans le but d'améliorer les cryptanalyses de SHA-1, et ont fait l'objet de publications [JP07a, JP07b]. En collaboration avec Stéphane Manuel [MP08], nous avons ensuite appliqué ces techniques avec succès à SHA-0.

Il existe d'autres travaux ayant trait à la recherche de candidats valides. On peut par exemple noter le travail de Sugita *et al.* [SKP07], assez à l'écart du chemin tracé par Wang *et al.*, qui permet de trouver des collisions pour SHA-1 à l'aide de modifications de message utilisant des bases de Gröbner [Buc65]. Cette voie, bien qu'intéressante, semble contrainte à ne cryptanalyser que des versions réduites de SHA-1 à cause de sa très grande consommation de degrés de liberté (le meilleur résultat parvient jusqu'à l'étape 58 de SHA-1).

6.4.1 L'attaque boomerang pour les algorithmes de chiffrement par blocs

L'attaque boomerang fut découverte par Wagner [Wag99] comme un outil de cryptanalyse des algorithmes de chiffrement par blocs. Elle permet d'enchevêtrer deux chemins différentiels indépendants et partiels (ne couvrant pas le nombre total d'étapes) en une attaque globale contre l'algorithme de chiffrement. L'idée conductrice est simple. Supposons que l'on dispose d'un chemin différentiel $\bigtriangledown_1$ sur la première moitié du chiffrement par blocs, commençant par une différence $\Delta^{\oplus}_{E1}$ en entrée et aboutissant à une différence $\Delta^{\oplus}_{S1}$ en sortie avec une probabilité p_1. Supposons d'autre part que l'on dispose d'un autre chemin différentiel $\bigtriangledown_2$ sur le déchiffrement de la deuxième moitié du chiffrement par blocs, commençant par une différence $\Delta^{\oplus}_{S2}$ sur le bloc de chiffré et aboutissant à une différence $\Delta^{\oplus}_{E2}$ sur le bloc intermédiaire avec une probabilité p_2. En utilisant ces deux chemins différentiels partiels, on peut dessiner un diagramme (voir figure 6.8) qui implique quatre paires texte clair/chiffré (P_1, C_1), (P_2, C_2), (P'_1, C'_1), (P'_2, C'_2) .

Ce diagramme peut aboutir à une attaque de la manière suivante. Tout d'abord, l'attaquant choisit aléatoirement un texte clair P_1 et demande le chiffrement de P_1 et de $P_2 = P_1 \oplus \Delta^{\oplus}_{E1}$. On note respectivement C_1 et C_2 les deux chiffrés obtenus. Ensuite, l'attaquant calcule les chiffrés C'_1 et C'_2 en appliquant la différence $\Delta^{\oplus}_{S2}$ sur C_1 et C_2 respectivement : $C'_1 = C_1 \oplus \Delta^{\oplus}_{S2}$ et $C'_2 = C_2 \oplus \Delta^{\oplus}_{S2}$. Il demande le déchiffrement de C'_1 et C'_2 et obtient les textes clairs P'_1 et P'_2. L'idée de l'attaque est de remarquer que lorsque la paire de textes clairs (P_1, P_2) est valide pour le chemin différentiel $\bigtriangledown_1$ et lorsque les deux déchiffrements (C_1, C'_1) et (C_2, C'_2) suivent le chemin différentiel $\bigtriangledown_2$, alors les valeurs intermédiaires correspondant à P'_1 et P'_2 présentent une différence $\Delta^{\oplus}_{S1}$. Enfin, si la paire de textes clairs (P'_1, P'_2) est aussi une paire valide pour le chemin différentiel $\bigtriangledown_1$, l'attaquant obtient par conséquent $P'_1 \oplus P'_2 = \Delta^{\oplus}_{E1}$.

En supposant une indépendance entre les quatre instances de chemins différentiels, on obtient une probabilité de succès finale égale à $p_1^2 \times p_2^2$. Si ces probabilités ne sont pas trop faibles,

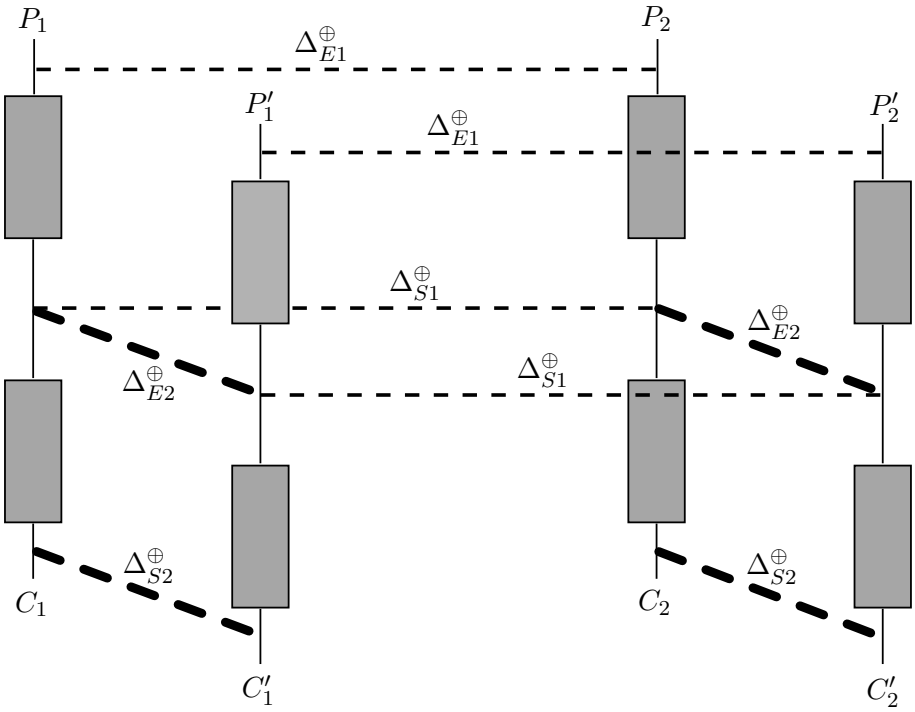

FIG. 6.8 – Vue schématique de l'attaque boomerang contre les algorithmes de chiffrement par blocs.

cela permettra à l'attaquant de distinguer avec une probabilité de succès non négligeable l'algorithme de chiffrement par blocs d'une permutation idéale. Il faut enfin noter que de nombreuses variantes existent, mais leur description dépasse le cadre de cette thèse.

6.4.2 Adapter l'attaque boomerang aux fonctions de hachage itérées

À première vue, du fait que beaucoup de fonctions de compression sont fondées sur un algorithme de chiffrement par blocs, il semble tentant d'appliquer directement les attaques boomerang aux fonctions de hachage itérées correspondantes. Toutefois, plusieurs problèmes empêchent cette approche directe. En particulier, la capacité de déchiffrer, élément essentiel pour les attaques boomerang, ne peut être obtenue dans le contexte des fonctions de hachage à cause de la résistance à la recherche de préimages. De plus, le contexte des fonctions de hachage est différent de celui des algorithmes de chiffrement par blocs puisqu'aucun secret, aucune clé ne sont à considérer ici.

Cependant, les attaques boomerang peuvent être utilisées autrement et devenir utiles pour la cryptanalyse des fonctions de hachage itérées. Plus exactement, il est possible d'adapter la variante à texte clair choisi des attaques boomerang (aussi appelée attaque boomerang amplifiée [KKS00]) pour accélérer des attaques différentielles déjà établies. L'idée est de manier, en plus du chemin différentiel principal utilisé de manière classique dans les attaques différentielles, plusieurs chemins différentiels partiels qui se comportent très bien sur un nombre limité d'étapes, mais ne couvrent pas le nombre total d'étapes de la fonction de compression. Autrement dit, ces chemins auront une bonne probabilité de succès sur un petit nombre d'étapes, mais ne peuvent être utilisés pour essayer de trouver une collision sur la version complète

de la fonction de hachage. Pour combiner ces petits chemins, appelés *chemins différentiels auxiliaires*, et le chemin différentiel principal, on utilise le même type de diagramme que pour les attaques boomerang sur les algorithmes de chiffrement par blocs (voir figure 6.9). Néanmoins, même si l'idée générale est la même, quelques différences apparaissent. La première différence, déjà citée, concerne l'incapacité de l'attaquant d'inverser la fonction de compression. Deuxièmement, nous n'aurons plus de symétrie avec les deux caractéristiques différentielles $\bigtriangledown_1$ et $\bigtriangledown_2$ qui jouent presque le même rôle. À la place, il y aura un chemin différentiel principal $\bigtriangledown_P$, qui sera notre cible, et plusieurs chemins différentiels auxiliaires $\bigtriangledown_A^i$ qui vont nous aider à trouver une paire de messages valide pour $\bigtriangledown_P$.

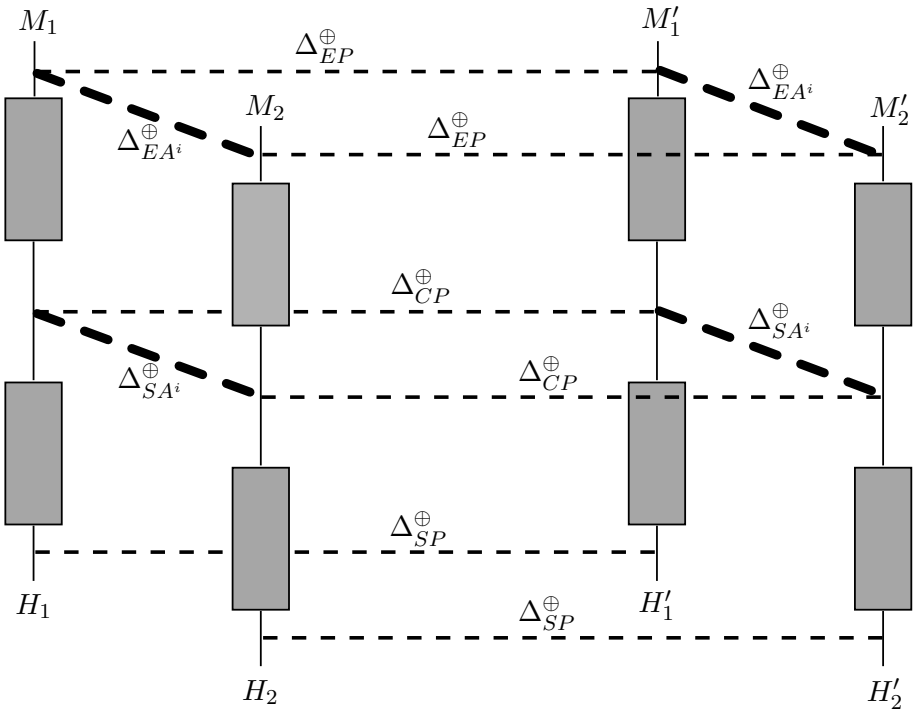

FIG. 6.9 – Vue schématique de l'attaque boomerang contre les fonctions de hachage.

Notre adaptation des attaques boomerang aux fonctions de hachage itérées nécessite une caractéristique différentielle principale. On commence donc en premier lieu par trouver un chemin différentiel principal pour la fonction de hachage considérée. Par souci de simplicité et sans perte de généralité, on suppose que ce chemin n'utilise qu'un unique bloc de message et ne nécessite donc qu'une seule itération pour aboutir à une collision. Il est trivial de généraliser la description de l'attaque au cas de chemins différentiels utilisant des presque collisions et de plusieurs itérations. Le chemin principal $\bigtriangledown_P$ imposera une certaine différence $\Delta^{\oplus}_{EP}$ et certaines conditions sur les mots de message. Nous chercherons alors des paires de messages (M, M') (avec $M' = M \oplus \Delta^{\oplus}_{EP}$) en espérant que l'une de ces paires aboutisse à une collision en sortie de la fonction de compression ($\Delta^{\oplus}_{SP} = 0$), ce qui se produira avec probabilité $p_{\Delta P}$ pour chaque candidat. Comme précédemment décrit pour la famille SHA, cette probabilité tient en compte des premières étapes où les mots du message M peuvent être choisis indépendamment les uns des autres et nous les appelons *étapes précoces*. Pour l'instant, nous divisons le reste des étapes

en deux sous-parties : *étapes centrales* et les *étapes finales* [‡]. Ceci est explicité dans la figure 6.10.

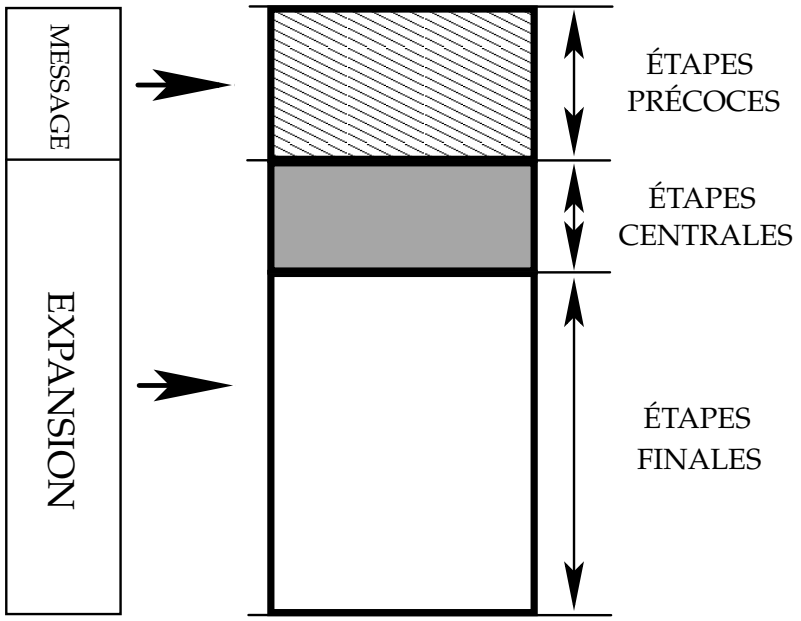

FIG. 6.10 – Division d'un chemin différentiel en trois sous-parties.

À chacune de ces parties on associe une probabilité de succès p_C et p_F respectivement et l'on note $\Delta^{\oplus}_{CP}$ la différence attendue sur l'état interne à la fin des étapes centrales. En supposant les étapes indépendantes, nous avons $p_{\Delta P} = p_C \cdot p_F$. Le but de notre attaque sera alors d'améliorer la probabilité de succès p_C et ainsi diminuer la complexité finale. Pour cela, nous utilisons comme outils des chemins différentiels auxiliaires qui couvriront avec une bonne probabilité de succès à la fois les étapes précoces, mais aussi celles centrales. Supposons un chemin différentiel $\bigtriangledown^i_A$ prédisant qu'avec probabilité $p_{\Delta A^i}$, deux messages M et $M \oplus \Delta^{\oplus}_{EA^i}$ aboutissent, à la fin des étapes centrales, à deux états internes avec une différence prévue (et non nécessairement nulle) $\Delta^{\oplus}_{SA^i}$. Choisissons à présent une paire de messages M_1 et $M'_1 = M_1 \oplus \Delta^{\oplus}_{EP}$ valide pour le chemin différentiel principal $\bigtriangledown_P$ durant les étapes précoces et centrales. Puis, supposons que les deux paires (M_1, M_2) avec $M_2 = M_1 \oplus \Delta^{\oplus}_{EA^i}$ et (M'_1, M'_2) avec $M'_2 = M'_1 \oplus \Delta^{\oplus}_{EA^i}$ sont toutes les deux valides pour le chemin différentiel auxiliaire $\bigtriangledown^i_A$. Dans ce cas, on peut s'apercevoir que les différences sur les états internes vont s'annuler à la fin des étapes centrales et que la paire de messages (M_2, M'_2) est donc aussi valide pour le chemin différentiel principal $\bigtriangledown_P$ jusqu'à la fin de ces étapes centrales (voir figure 6.9). Ici, nous n'avons considéré que des différences binaires, ce qui signifie que notre nouvelle paire de messages (M_2, M'_2) peut vérifier le chemin différentiel pour les différences binaires, mais pas pour celles modulaires ou modulaires signées. On supposera de ce fait que les chemins auxiliaires sont assez indépendants de celui principal pour éviter ce genre de comportement. En supposant cette indépendance, la paire de messages (M_2, M'_2) fournira une collision avec probabilité $p^2_{\Delta A^i} \times P_F$.

Cette idée d'attaque semble assez prometteuse en théorie, car lorsque $p^2_{\Delta A^i} < p_C$, on peut s'attendre à une amélioration de l'attaque classique par la technique boomerang. En pratique, la

[‡]On peut imaginer une autre sous-partie : nous avons déjà vu que dans le cas des attaques multiblocs, les toutes dernières étapes sont un peu particulières puisque quelques conditions peuvent y être ignorées.

situation est plus complexe. À moins de posséder une première paire de messages (M_1, M_1'), il est impossible de construire la deuxième paire (M_2, M_2'). Par conséquent, au mieux, cette amélioration double le nombre de paires de messages candidates. Heureusement, lorsqu'un grand nombre de chemins différentiels auxiliaires peut être trouvé, ce qui est une hypothèse raisonnable étant donné la faible quantité d'étapes considérée, on pourra utiliser cette amélioration plusieurs fois. En supposant que $p_{\Delta A^i} = 1$ pour tous les i chemins différentiels auxiliaires, on amplifie une paire de messages candidate en 2^i paires. Il est évident qu'il faudra arranger les chemins différentiels auxiliaires de manière appropriée pour éviter tout chevauchement ou toute incompatibilité similaire. Lorsque $p_{\Delta A^i}$ est inférieur à 1 (mais pas trop faible), nous amplifions tout de même une seule paire de messages en plusieurs. On peut voir tout de suite l'utilité d'une telle attaque pour l'accélération de la recherche d'une paire de messages valide pour un chemin différentiel.

On pourrait opposer deux principales objections à cette amélioration. Premièrement, l'hypothèse d'indépendance peut paraître peu naturelle, puisque toutes ces paires de messages sont corrélées. Expérimentalement, cette hypothèse est fausse. Cependant, nous avons remarqué que pour des caractéristiques différentielles bien choisies, le biais induit par les dépendances joue en la faveur de l'attaquant et non contre lui. Le principal argument étant que, puisque M et $M \oplus \Delta_{EP}^{\oplus}$ suivent des états internes successifs très similaires durant le calcul de la fonction de compression, la probabilité de succès finale des deux paires pour le chemin différentiel auxiliaire est plus proche de $p_{\Delta A^i}$ que de $p_{\Delta A^i}^2$. La seconde objection serait que les étapes précoces et centrales ne semblent pas gratuites pour l'attaquant dans les chemins différentiels auxiliaires. Cela peut être un problème majeur puisque nous souhaitons que $p_{\Delta A^i}$ soit bien plus grand que p_C. En fait, il existe plusieurs moyens d'éviter ce problème, dépendants de la fonction de hachage considérée et des propriétés diverses des chemins différentiels utilisés. Nous donnons des exemples dans la section suivante et nous verrons plus tard que l'attaque boomerang pour les fonctions de hachage est en fait une généralisation des bits neutres et des modifications de message utilisés pour la famille SHA.

6.4.3 Les différentes approches possibles

Nous avons décrit dans la section précédente l'attaque boomerang pour les fonctions de hachage de manière théorique. Nous allons maintenant examiner les différentes approches pratiques qui se présentent à nous. En particulier, nous discuterons des buts possibles d'un chemin différentiel auxiliaire et de la manière de l'utiliser.

Les approches « bits neutres » ou « modification de message »

Concentrons-nous préalablement sur l'objectif d'un chemin différentiel auxiliaire. On peut remarquer la similarité entre la description de ces chemins et la technique des bits neutres pour SHA-0, précédemment expliquée. En fait, les bits neutres sont une instance de l'attaque boomerang. Une fois le chemin différentiel principal établi, Biham et Chen [BC04] tentent de trouver des modifications qui ne détruiront pas la validité d'une paire de messages jusqu'à une certaine étape de ce chemin. Chacune de ces modifications forme en fait un chemin différentiel auxiliaire pour lequel on considère que les étapes centrales vont approximativement de 16 à 20. Une fois un grand nombre de bits neutres trouvés, Biham et Chen multiplient les instances valides en parcourant toutes les combinaisons de modifications possibles (comme nous multiplions les instances en appliquant toutes les combinaisons de chemins différentiels auxiliaires) et observent expérimentalement que ces nouveaux candidats seront eux-mêmes valides avec

une certaine probabilité. Ceci s'explique dans notre modèle par l'interdépendance des chemins différentiels auxiliaires, et notre formalisation plus adéquate fournit des outils pour essayer de diminuer cet effet néfaste pour l'attaquant. La différence essentielle entre les deux techniques est que les bits neutres sont cherchés de manière empirique par Biham et Chen tandis que nous donnons ici une explication théorique des mécanismes sous-jacents. Nous pouvons donc effectuer des recherches de bits neutres plus efficaces, ciblées et prévues à l'avance dans le chemin différentiel principal.

Il existe une manière radicalement différente d'utiliser les attaques boomerang. Au lieu de multiplier les instances valides, comme pour les bits neutres, on peut à la place tenter de corriger celles qui sont invalides. Cette méthode, duale de la précédente, généralise en fait les modifications de message introduites par Wang *et al.* [WLF05, WYY05b, WY05, WYY05d]. Les modifications trouvées par Wang *et al.* pour corriger certaines conditions qui ne seraient pas vérifiées sont en fait des chemins différentiels auxiliaires dont la différence en sortie n'est pas nulle. On souhaitera donc que les différences obtenues à la fin du chemin auxiliaire corrigent des conditions invalides du chemin principal durant les étapes centrales. La probabilité de succès d'une modification sera alors dépendante de la qualité du chemin différentiel auxiliaire, mais aussi de l'interdépendance entre tous ceux utilisés. De la même façon que pour les bits neutres, nous pouvons à présent mieux construire des modifications de message, et ce, de manière automatisée, en cherchant des chemins différentiels auxiliaires. L'interdépendance sera aussi plus facile à mesurer et à diminuer. L'attaque sera améliorée et il nous sera plus facile de calculer le coût réel d'un processus complet de modification de message.

Dans les deux cas d'utilisation, l'amélioration des attaques boomerang provient du fait que l'on peut considérer que le calcul du coût réel du chemin différentiel principal commence au début des étapes finales au lieu du début des étapes centrales. Dans le cas des bits neutres, on amortit le coût de recherche d'une paire de messages valide jusqu'à la fin des étapes centrales en multipliant les instances à partir d'une seule, ce qui nous donne un coût unitaire faible. Dans le cas des modifications de message, on utilise les chemins différentiels auxiliaires pour engendrer directement et efficacement une paire de messages valide jusqu'à la fin des étapes centrales.

La mise en place des chemins auxiliaires

Nous avons vu deux objectifs très différents pour les attaques boomerang, mais l'on peut aussi distinguer les mises en place possibles de celles-ci.

La première, la plus simple, consiste à placer l'attaque boomerang tout à la fin de la procédure d'attaque : on établit tout d'abord un chemin différentiel, puis l'on cherche et utilise les caractéristiques auxiliaires lors de la recherche de paires de messages, comme cela est effectué pour la technique des bits neutres originale. Il faut noter que le processus de recherche de chemins auxiliaires devra être exécuté pour chaque paire de messages valide jusqu'aux étapes centrales. Nous retombons donc exactement sur la méthode originale de Biham et Chen dans le cas d'une utilisation en bits neutres. Dans le cas des modifications de message, cette configuration semble inadaptée, car nous n'essayons pas d'amortir le coût, mais d'engendrer directement une paire de messages valide jusqu'à la fin des étapes centrales, ce qui devra être réalisé très rapidement. Cela sera très difficile puisque le processus de recherche de chemins auxiliaires devra être exécuté pour chaque candidat. De façon générale, cette méthode présente l'avantage d'être simple, mais ne fournira pas de très bons résultats en ce qui concerne l'accélération de recherche d'une paire de messages valide. En effet, la probabilité qu'il existe des bits neutres

ou des modifications de message puissants et ayant de bonnes chances de succès est faible. De plus, dans le cas de la famille SHA, lorsque des parties non linéaires sont ajoutées, beaucoup de degrés de liberté sont consommés. De nombreuses conditions sur le chemin différentiel sont nécessaires pour les parties non linéaires, et ces conditions pourraient aider aussi l'accélération de recherche.

Cette dernière remarque nous amène à la deuxième méthode, où l'accélération de recherche est directement prise en compte dans le chemin différentiel (de manière totale ou seulement partielle). L'attaquant peut en effet adapter la physionomie du chemin différentiel principal pour améliorer les chemins auxiliaires. Pour améliorer encore l'efficacité, l'attaquant pourra directement ajouter des conditions dans le chemin différentiel pour améliorer les probabilités de succès des caractéristiques auxiliaires. Cela nous permettra d'engendrer des bits neutres beaucoup plus puissants (valides jusqu'à une étape plus tardive) ou des modifications de message ayant une meilleure probabilité de succès et rendant donc le processus total de modification beaucoup plus rapide. Pour le cas de la famille SHA, nous pouvons distinguer deux variantes de la deuxième méthode : l'intégration dans le chemin différentiel principal des chemins auxiliaires avant ou après celle des parties non linéaires. Si l'on intègre avant, nous pourrons trouver de bien meilleurs chemins auxiliaires et nous économiserons des degrés de liberté du fait que certaines conditions seront communes à ces chemins et aux parties non linéaires. Cependant, l'attaque sera beaucoup plus complexe à mettre en oeuvre et demandera un important travail de préparation. De plus, trop de chemins auxiliaires vont compliquer l'établissement des parties non linéaires. Un compromis sera donc nécessaire. On peut remarquer que la modification de message avec une recherche de chemins auxiliaires postérieure aux parties non linéaires est illustrée par la technique de modification de message originale de Wang *et al.* et par celle des modifications sous-marines introduites par Naito *et al.* [NSS06]. On peut aussi noter que la technique des tunnels de Klima [Kli06], appliqués à MD5, correspond à des bits neutres avec quelques conditions sur le chemin différentiel ajoutées après l'établissement des parties non linéaires. De nouvelles et intéressantes perspectives s'ouvrent à nous, car les configurations restantes n'ont pas encore été utilisées.

Le tableau 6.3 récapitule toutes les différentes variantes de l'attaque boomerang pour les fonctions de hachage.

	bits neutres	**modification de message**
avant parties non linéaires	nouvelle variante	nouvelle variante
après parties non linéaires	tunnels [Kli06]	modification de message [WYY05b] modifications sous-marines [NSS06]
après chemin différentiel total	bits neutres [BC04]	nouvelle variante

TAB. 6.3 – Les différentes variantes de l'attaque boomerang pour les fonctions de hachage.

6.4.4 Application à la famille SHA

Nous allons nous concentrer dans cette section sur le cas de la famille SHA, pour laquelle nous appliquons les attaques boomerang et améliorerions ainsi les techniques existantes d'accélération de recherche de candidats valides. Nous divisons cette description en trois étapes : la construction des chemins auxiliaires, l'incorporation de ces chemins dans la caractéristique principale, puis leur utilisation lors de la recherche de candidats valides. En collaboration avec Antoine Joux [JP07a, JP07b], nous avons publié la partie théorique des attaques boomerang et leur application pour SHA-1. Nous avons ensuite traité le cas de SHA-0 en coopération avec Stéphane Manuel dans [MP08].

Construire des chemins différentiels auxiliaires pour SHA

Notre but ici est de construire un chemin différentiel pour SHA sur un nombre d'étapes limité et dont les différences en sortie sont relativement contrôlées. Bien entendu, plus le nombre d'étapes sera important et plus le chemin différentiel sera puissant, mais difficile à construire. Il faut aussi garder à l'esprit que l'attaquant souhaite utiliser des chemins différentiels auxiliaires légers, en ce sens que peu de degrés de liberté sont nécessaires pour obtenir une bonne probabilité de succès $p_{\Delta A^i} \simeq 1$.

On peut d'ores et déjà remarquer que l'analyse sera légèrement différente suivant que l'on souhaite appliquer l'approche « bits neutres » ou « modification de message ». Nous nous concentrerons en premier sur la première approche. Même si la physionomie générale du chemin différentiel principal est connue, il semble assez risqué de laisser se propager des différences dans les chemins auxiliaires. Naturellement, nous allons donc essayer de construire des chemins qui aboutissent à une collision interne, et ce, à l'étape la plus tardive possible et pour un coût en termes de degrés de liberté suffisamment faible. Les problèmes étant très similaires, nous allons utiliser la technique classique de construction de chemins différentiels : les collisions locales. Il faudra utiliser le moins possible de collisions locales pour avoir une bonne probabilité de succès intrinsèque (cette probabilité peut être augmentée à loisir au prix d'une consommation de degrés de liberté en ajoutant des conditions). Le vecteur de perturbation pour le chemin auxiliaire ne concernant qu'un nombre d'étapes très réduit, la rotation dans l'expansion de message de SHA-1 ne sera pas très contrariante et sa construction pour SHA-0 et SHA-1 sera quasi identique, contrairement au cas du chemin différentiel principal.

Les collisions locales ont déjà été décrites en section 5.2.1, mais nous utilisons ici une version un peu plus générale. Nous considérerons une variante où l'on admettra un comportement non linéaire des fonctions booléennes (et notamment de la fonction IF utilisée pour les 20 premières étapes de SHA). Auparavant, du fait de la complexité pour construire un chemin différentiel sur 80 étapes, nous avions utilisé un modèle linéaire et ainsi constamment corrigé les perturbations insérées. Ce modèle était utile, car valide pour les trois fonctions booléennes utilisées dans SHA, et seul le vecteur de perturbation était à considérer. Comme les chemins auxiliaires concernent presque uniquement la fonction booléenne IF, on augmentera le nombre de collisions locales possibles en se permettant de corriger ou non lorsque la perturbation s'exprime dans la fonction booléenne (contrairement à la fonction booléenne XOR, la fonction IF peut absorber des différences). Les autres corrections restent identiques et le tableau 6.4 résume les différentes possibilités considérées. Au lieu d'une seule collision locale dans le modèle linéaire, nous disposons à présent de $2^3 = 8$ collisions locales non linéaires différentes.

Nous souhaitons utiliser le moins de collisions locales possible, 5 semblant être déjà une

étape	**type**	**contraintes**
i	pas de retenue	$A_{i+1}^j = a$ et $W_i^j = a$
$i+1$	correction	$W_{i+1}^{j+5} = \overline{a}$
$i+2$	pas de correction	$A_{i-1}^{j+2} = A_i^{j+2}$
	correction	$A_{i-1}^{j+2} \neq A_i^{j+2}$ et $A_{i-1}^{j+2} = W_{i+2}^j \oplus \overline{a}$
$i+3$	pas de correction	$A_{i+2}^{j-2} = 0$
	correction	$A_{i+2}^{j-2} = 1$, $W_{i+3}^{j-2} = \overline{a}$
$i+4$	pas de correction	$A_{i+3}^{j-2} = 1$
	correction	$A_{i+3}^{j-2} = 0$, $W_{i+4}^{j-2} = \overline{a}$
$i+5$	correction	$W_{i+5}^{j-2} = \overline{a}$

TAB. 6.4 – Variantes possibles d'une collision locale avec comportement non linéaire de la fonction booléenne pour une perturbation insérée sur W_i^j dans le premier tour de SHA.

quantité trop grande. Par contre, nous ne nous limitons pas concernant l'étape finale, car un chemin auxiliaire très léger et peu puissant pourra nous être utile (la raison en sera détaillée plus loin). Il semble a priori logique d'ajouter des perturbations sur une seule position de bit, pour les mêmes raisons que lors de la construction d'un chemin différentiel principal. Nos chemins auxiliaires seront ainsi invariants par rotation sur la position de bit. Nous forçons donc des collisions locales non linéaires durant les 16 premières étapes simplement en parcourant exhaustivement les 16 positions possibles, et pour toutes les collisions locales non linéaires possibles. Nous souhaitons éviter de poser des conditions sur des bits postérieurs à l'étape 15 puisque l'attaquant n'y a plus de contrôle direct, ce qui impliquerait une diminution significative de la probabilité $p_{\Delta A^i}$. Plus précisément, pour le cas de SHA-1, nous interdisons l'introduction de perturbations à partir de l'étape 11 du fait que certaines corrections correspondantes se dérouleraient sur une position de bit décalée à cause de l'expansion de message (si la perturbation est introduite sur une position de bit j, certaines corrections s'opèreront de manière erronée sur la position de bit $j+1$ et réinjecteraient des différences non contrôlées). Cela nous donne $8^{16} = 2^{48}$ candidats au total pour SHA-0 et $8^{11} = 2^{33}$ candidats pour SHA-1, mais nous ne testons que ceux qui comportent au plus 5 collisions locales non linéaires, ce qui réduit grandement l'espace de recherche. Pour chaque candidat engendré, on calcule l'expansion de message à partir de l'étape 16 et l'on identifie l'étape k à laquelle arrive la première différence non contrôlée. Cela impliquera que nous aurons une collision interne jusqu'à l'étape $k-1$ et ce chemin pourra être utilisé comme bit neutre. En pratique, même si des différences non contrôlées arrivent aux étapes k, $k+1$, etc., ces perturbations n'interféreront pas immédiatement avec le chemin différentiel principal. Ceci explique pourquoi un chemin auxiliaire avec la première différence non contrôlée à l'étape k peut être utilisé comme un bit neutre jusqu'à une étape plus tardive que $k-1$. Plus l'étape visée est tardive et plus la probabilité de succès du bit neutre décroît. Cela dépend bien entendu aussi de la position de bit considérée, relativement au chemin différentiel principal. On peut mesurer cet effet en pratique pour faire le bon choix de l'étape finale du bit neutre.

La méthode de recherche précédemment expliquée nous fournit de nombreux candidats et nous donnons ici deux bons exemples de chemins différentiels auxiliaires. Le premier, noté AP_1 (*Auxiliary Path* 1), aura très peu de contraintes, mais l'étape finale est assez précoce. En d'autres termes, il sera léger, mais peu puissant. Au contraire, le second, noté AP_2, induira beaucoup de contraintes, mais aura une étape finale assez tardive. Nous donnons en figure 6.11 et 6.12 la description de AP_1 et AP_2 respectivement. AP_1 utilise une collision locale à l'étape 6 et la première différence non contrôlée apparaît à l'étape 19, avec W_{19}^{j-2}. Si les conditions sur le chemin sont confirmées, nous aurons $p_{\Delta A^i} = 1$: chaque bit a peut prendre n'importe quelle valeur, tant que les contraintes $\overline{a}$ sont vérifiées. AP_2 utilise trois collisions locales aux étapes 0, 2, 10 et la première différence non contrôlée apparaît à l'étape 24, avec W_{24}^{j}. On peut noter que AP_2 possède des conditions sur la variable de chaînage, ce qui complique légèrement son utilisation. En pratique, AP_1 et AP_2 pourront être utilisés jusqu'aux étapes 22 et 27 respectivement, au prix d'une légère diminution de la probabilité de succès du bit neutre.

	W_0 à W_{15}	W_{16} à W_{31}
perturbations	00000010000000000	
différences sur W^j	00000010000000000	00001011011001111
différences sur W^{j+5}	00000001000000000	00000101101100111
différences sur W^{j-2}	00000000000100000	0001$\underline{1}$001000000010

i	∇A_i	∇W_i
-1 :	----------	
00 :	----------	----------
01 :	----------	----------
02 :	----------	----------
03 :	----------	----------
04 :	----------	----------
05 :	----------b----	----------
06 :	----------b----	----------a--
07 :	----------a--	----------$\overline{a}$-------
08 :	----------0	----------
09 :	----------1	----------
10 :	----------	----------
11 :	----------	----------$\overline{a}$
12 :	----------	----------
13 :	----------	----------
14 :	----------	----------
15 :	----------	----------

FIG. 6.11 – Chemin différentiel auxiliaire AP_1. Le premier tableau donne les 32 premières étapes du vecteur de perturbation par des collisions locales non linéaires. La première différence non contrôlée apparaît à l'étape 19. Le second tableau fournit, pour le cas où l'on se positionne sur la position de bit $j = 2$, les contraintes à ajouter pour avoir une probabilité de réussite égale à 1 pour ce chemin auxiliaire. Les lettres représentent une valeur de bit, et la valeur complémentaire est notée par cette même lettre avec une barre au-dessus.

Pour ce qui concerne l'approche « modification de messages », le même type de méthode peut être utilisé. L'attaquant utilise des collisions locales pour obtenir une chemin auxiliaire où la première différence non contrôlée est assez tardive. Il essaye ensuite de pouvoir influer

	W_0 à W_{15}	W_{16} à W_{31}
perturbations	1010000000100000	
différences sur W^j	1010000000100000	00000000$\underline{1}$0110110
différences sur W^{j+5}	0101000000010000	0000000001011011
différences sur W^{j-2}	0001111100000011	0000000000001110

i	∇A_i	∇W_i
-1 :	----------d----	
00 :	----------d----	----------a--
01 :	----------e-a--	----------$\overline{a}$-------
02 :	----------e---1	----------b--
03 :	----------b-0	----------$\overline{b}$------$\overline{a}$
04 :	----------0	----------$\overline{a}$
05 :	----------0	----------$\overline{a}$
06 :	----------	----------$\overline{b}$
07 :	----------	----------$\overline{b}$
08 :	----------	----------
09 :	----------f----	----------
10 :	----------f----	----------c--
11 :	----------c--	----------$\overline{c}$-------
12 :	----------0	----------
13 :	----------0	----------
14 :	----------	----------$\overline{c}$
15 :	----------	----------$\overline{c}$

FIG. 6.12 – Chemin différentiel auxiliaire AP_2. Le premier tableau donne les 32 premières étapes du vecteur de perturbation par des collisions locales non linéaires. La première différence non contrôlée apparaît à l'étape 24. Le second tableau fournit, pour le cas où l'on se positionne sur la position de bit $j = 2$, les contraintes à ajouter pour avoir une probabilité de réussite égale à 1 pour ce chemin auxiliaire. Les lettres représentent une valeur de bit, et la valeur complémentaire est notée par cette même lettre avec une barre au-dessus.

directement sur une condition précise grâce à ces premières différences non contrôlées. On pourra aussi étudier les cas de propagation de ces différences pour pouvoir atteindre d'autres positions : ce n'est pas la différence non contrôlée qui corrigera elle-même la condition, mais sa propagation (on espère néanmoins que la différence non contrôlée n'invalidera pas une condition déjà vérifiée sur une étape précédente).

Il faut noter que de nombreux bits neutres ou modifications de message existent pour les étapes 16, 17 ou 18, réalisables par exemple en introduisant une perturbation sur les toutes dernières étapes précoces durant lesquelles l'attaquant jouit d'un contrôle sur les messages insérés. Cela s'explique par la très faible diffusion étant donné le peu d'étapes considérées et cela résulte essentiellement de la remarque de Chabaud et Joux [CJ98] qui stipule que les étapes 16 à 18 peuvent être attaquées de manière indépendante en adaptant l'implantation de l'attaque en conséquence.

Enfin, nous pouvons évaluer le coût en degrés de liberté d'un chemin différentiel auxiliaire comme étant le nombre de conditions qu'il force dans la caractéristique principale diminué de l'inverse du logarithme en base deux de l'inverse de sa probabilité de succès. En effet, les

conditions représentent le coût direct tandis que sa probabilité de succès dénote la nouvelle quantité de degrés de liberté que nous apporte ce chemin.

Placer les chemins différentiels auxiliaires pour SHA

Une fois les chemins différentiels auxiliaires définis, il faudra à présent les placer à l'intérieur de la caractéristique principale. La méthode de placement dépendra du moment où l'on souhaite ajouter ces chemins.

Si l'on souhaite ajouter les chemins auxiliaires après avoir sélectionné une paire de messages, on se retrouvera dans le cas des bits neutres. On essayera donc de trouver des positions de bit où les conditions nécessaires au bon déroulement du chemin auxiliaire sont vérifiées. On voit tout le problème de cette méthode, car peu de positions seront satisfaisantes si l'on ne règle pas cette partie en amont.

Si l'on souhaite ajouter les chemins auxiliaires juste après l'établissement des parties non linéaires du chemin principal, on utilisera exactement la même technique. La probabilité de trouver des positions satisfaisantes sera augmentée puisque nous aurons des degrés de liberté qui proviendront de tous les bits non contraints du chemin différentiel principal.

Enfin, si l'on souhaite ajouter les chemins auxiliaires avant l'établissement des parties non linéaires du chemin principal, nous disposerons d'encore plus de degrés de liberté. Il faudra juste tenir compte du squelette du chemin différentiel principal, défini par le vecteur de perturbation. On évitera ainsi de placer un chemin auxiliaire sur des positions de bit qui contiennent des différences dans celui principal (on utilisera des positions de bit situées approximativement entre 6 et 28), car certaines conditions ne pourraient être vérifiées pour les deux messages de la paire testée. En général, si les chemins auxiliaires sont situés sur des positions proches du centre (positions de bit 15 ou 16), l'interaction entre ces chemins et le chemin principal sera minimisée. Par contre, pour économiser des degrés de liberté, on pourra tenter de combiner deux chemins auxiliaires ayant des conditions communes. C'est précisément ici qu'intervient l'outil de génération automatique des parties non linéaires [CR06]. Nous allons placer le plus de caractéristiques différentielles auxiliaires possible dans le squelette du chemin principal, puis engendrer des parties non linéaires qui tiennent compte des conditions de ces chemins auxiliaires. Bien entendu, plus nous placerons de chemins et plus l'accélération de recherche sera ensuite puissante. En contrepartie, la perte de degrés de liberté se traduira par un ralentissement voire un blocage de l'outil automatique de génération de parties non linéaires. Un compromis sera donc ici nécessaire, situé au maximum de chemins auxiliaires qu'il est possible d'insérer en préservant un comportement relativement bon de l'outil automatique.

Plus généralement, il faut aussi considérer les types de chemins auxiliaires à inclure. Il est absolument inutile de placer uniquement des chemins très puissants et très coûteux en termes de degrés de liberté. Un certain nombre de caractéristiques auxiliaires peu puissantes seront suffisantes pour amortir le coût de recherche d'une paire de messages valide à une étape assez précoce. Ensuite, plus on essayera d'amortir le coût pour une étape tardive, plus les chemins à utiliser devront être puissants. Il est donc important d'utiliser une bonne proportion des différents chemins auxiliaires disponibles pour ne pas gaspiller des degrés de liberté. Par exemple, si nous avons une condition à une étape i puis trois conditions à une étape $i+5$, l'idéal serait d'utiliser un chemin auxiliaire de étant un bit neutre pour l'étape i puis trois chemins étant des bits neutres pour l'étape $i+5$. Utiliser quatre chemins étant des bits neutres pour l'étape $i+5$ aboutira au même résultat, mais consommera a priori plus de degrés de liberté.

Enfin, surtout dans le cas des modifications de message où l'interdépendance entre les chemins auxiliaires joue un rôle assez important, on peut organiser l'attaque en ordonnançant rigoureusement ces chemins. Par cette méthode, on essayera de limiter au maximum les dépendances en sachant exactement dans quel ordre seront utilisés les chemins auxiliaires.

Utiliser des chemins différentiels auxiliaires pour SHA

Une fois les chemins différentiels auxiliaires trouvés et placés, la dernière partie concerne leur utilisation. Dans les deux cas d'utilisation, l'algorithme est simple.

Pour le cas de l'approche de type « bits neutres », supposons que l'on possède des chemins auxiliaires dont les puissances vont de l'étape i pour les plus faibles à l'étape j pour les plus forts. On cherche préalablement une paire de messages i-valide, puis on multiplie cette instance en déclenchant les chemins auxiliaires de puissance i. Pour toutes les nouvelles instances, on ne garde que celles $i+1$-valides (certaines seront même éliminées avant si la probabilité de succès du bit neutre n'est pas égale à 1) et on continue à la prochaine étape. On multiplie, de la même manière, les instances jusqu'à l'étape j, puis l'on vérifie enfin parmi toutes les instances restantes si l'une d'elles suit parfaitement le chemin différentiel principal jusqu'à la dernière étape. Nous réitérons cet algorithme tant qu'aucune paire de messages valide n'a été trouvée, ou jusqu'à ce que tous les degrés de liberté soient consommés (auquel cas il faudra chercher une nouvelle instance de chemin différentiel principal).

Pour l'approche de type « modification de message », supposons que l'on possède des chemins auxiliaires qui corrigent des conditions de l'étape i pour les plus faibles, et jusqu'à l'étape j pour les plus forts. On cherche d'abord une paire de messages $i-1$-valide, puis l'on teste si les conditions à l'étape i sont vérifiées. Si tel est le cas, on continue à l'étape suivante sans déclencher les modifications de message. Autrement, on utilise les modifications de messages correspondant aux conditions non vérifiées puis l'on teste si la modification n'a pas changé les contraintes avant l'étape i et si l'étape i est à présent correcte. Il faudra tester une nouvelle instance de message si toutes les conditions ne sont pas vérifiées jusqu'à l'étape i, sinon nous pouvons passer à l'étape suivante. De la même manière, nous corrigeons notre instance jusqu'à l'étape j, puis vérifions enfin si notre instance suit parfaitement le chemin différentiel principal jusqu'à la fin. Nous réitérons cet algorithme tant qu'aucune paire de messages valide n'a été trouvée, ou jusqu'à ce que tous les degrés de liberté soient consommés (auquel cas il faudra chercher une nouvelle instance de chemin différentiel principal).

Il faut noter que nous utilisons des différentielles auxiliaires formées de collisions locales. Si toutes les conditions relatives à ces collisions locales sont intégrées dans le chemin principal (et donc, nous avons une probabilité de succès égale à 1), nous n'avons pas besoin de recalculer tout le cheminement à chaque déclenchement d'un chemin auxiliaire. Nous pouvons directement commencer le calcul à partir de la première différence non contrôlée. Nous obtenons ainsi un gain de performance supplémentaire par l'implantation des attaques boomerang.

CHAPITRE 7
Application à la cryptanalyse de la famille SHA

Sommaire

Nous appliquons ici toutes les améliorations décrites dans le chapitre précédent aux cas de SHA-0 et SHA-1, en utilisant deux blocs de message pour trouver une collision. Les résultats concernant SHA-0 ont été publiés à la conférence FSE 2008 avec Stéphane Manuel [MP08], ceux concernant SHA-1 à la conférence CRYPTO 2007 avec Antoine Joux [JP07a].

7.1 Cas de SHA-0

Notre amélioration par rapport aux attaques précédentes sur SHA-0 [WYY05d, NSS06] est double. Premièrement, en utilisant les attaques boomerang, nous améliorons l'accélération de recherche. De plus, grâce à l'outil automatique de génération de parties non linéaires [CR06] et grâce à notre recherche améliorée de vecteurs de perturbation, nous aboutissons à un bien meilleur chemin différentiel nécessitant deux blocs de message (voir figures 7.3 et 7.4), les précédentes attaques n'en utilisant qu'un seul. Ce chemin comporte 39 conditions pour le premier bloc et 38 conditions pour le second entre les étapes 17 et 80 (cela représente approximativement la complexité de l'attaque sans accélération de recherche). Il faut noter qu'il existe un meilleur chemin pour minimiser le nombre de conditions, mais son comportement avec l'outil automatique de génération de parties non linéaire est beaucoup plus délicat lorsque l'on souhaite ajouter des caractéristiques différentielles auxiliaires.

Nous avons utilisé les deux types de chemins auxiliaires AP_1 et AP_2 selon l'approche « bits neutres ». Pour le premier bloc, nous avons 2 chemins auxiliaires AP_1 aux positions de bit 9 et 11, puis 5 chemins AP_2 aux positions de bit 10, 14, 19, 22 et 27. Pour le second bloc, nous avons 2 chemins auxiliaires AP_1 aux positions de bit 9 et 11, puis 3 chemins AP_2 aux positions de bit 17, 22 et 30. Cette différence entre les deux blocs s'explique par le fait que la valeur d'initialisation de la variable de chaînage pour SHA est très structurée (l'égalité entre deux bits situés à la même position sur les deux derniers mots de la variable de chaînage initialisée n'est pas distribuée de manière uniforme), et facilite donc grandement l'incorporation de chemins auxiliaires AP_2 qui requièrent des conditions sur la variable de chaînage. Cependant, en moyenne, il est aisé de placer 5 différentielles auxiliaires.

Finalement, nous aboutissons à une attaque contre SHA-0 [MP08] ne requérant qu'une heure de calcul sur un ordinateur ordinaire (au lieu de 100 heures précédemment). Nous avons mesuré la complexité en nombre d'appels à la fonction de compression de SHA-0, $2^{32,2}$ pour le premier bloc et 2^{33} pour le deuxième. Cela nous donne une complexité totale de $2^{33,6}$ appels à la fonction de compression de SHA-0, ce qui est la meilleure attaque à ce jour. Théoriquement, nous attendions $2^{39-7} = 2^{32}$ appels pour le premier bloc et $2^{38-5} = 2^{33}$ pour le deuxième, ce qui est très proche de la complexité mesurée. Nous donnons dans la figure 7.1 un exemple de collision pour SHA-0. Dans notre évaluation de complexité, nous n'avons pas compté la génération de parties non linéaires, et ce, pour deux raisons. Tout d'abord, elle est relativement faible (en moyenne, 30 minutes de calcul sur un ordinateur ordinaire). Ensuite, une fois le premier bloc trouvé, le chemin différentiel pour le deuxième bloc nous permet d'engendrer un grand nombre de collisions sans recalculer la partie non linéaire. Ainsi, le coût relatif de la génération des parties non linéaires par collision trouvée devient complètement négligeable.

i	Message 1 - Premier Bloc	Message 2 - Premier Bloc
0-3	**4643450B 41D35081 FE16DD9B 3BA36244**	**46434549 41D350C1 FE16DDDB 3BA36204**
4-7	**E6424055 16CA44A0 20F62444 10F7465A**	**66424017 96CA44A0 A0F62404 10F7465A**
8-11	**5A711887 51479678 726A0718 703F5BFB**	**5A7118C5 D147963A 726A0718 703F5BB9**
12-15	**B7D61841 A5280003 6B08D26E 2E4DF0D8**	**B7D61801 A5280041 6B08D26C AE4DF0D8**

i	Message 1 - Second Bloc	Message 2 - Second Bloc
0-3	**9A74CF70 04F9957D EE26223D 9A06E4B5**	**9A74CF32 04F9953D EE26227D 9A06E4F5**
4-7	**B8408AF6 B8608612 8B7E0FEA E17E363C**	**38408AB4 38608612 0B7E0FAA E17E363C**
8-11	**A2F1B8E5 CA079936 02F2A7CB F724E838**	**A2F1B8A7 4A079974 02F2A7CB F724E87A**
12-15	**37FFC03A 53AA8C43 90811819 312D423E**	**37FFC07A 53AA8C01 9081181B B12D423E**

Valeur de haché finale
6F84B892 1F9F2AAE 0DBAB75C 0AFE56F5 A7974C90

FIG. 7.1 – Un exemple de paire de messages aboutissant à une collision pour SHA-0 et sa valeur de haché associée [MP08], pour les chemins différentiels des figures 7.3 et 7.4. Les octets sont notés sous forme hexadécimale.

7.2 Cas de SHA-1

Nous décrivons ici comme exemple des techniques boomerang pour SHA-1 une attaque sur la version réduite à 70 étapes. Le chemin différentiel du premier bloc est présenté en figure 7.5 et le deuxième en figure 7.6. Ce chemin comporte 42 conditions pour le premier bloc et 45 conditions pour le second entre les étapes 17 et 80 (cela représente approximativement la complexité de l'attaque sans accélération de recherche). Ce nombre de conditions ne peut être lu directement sur les figures 7.5 et 7.6, car il nous faut prendre en compte les contraintes qui peuvent être relaxées durant les dernières étapes, ainsi que la compression de bit. Ce travail étant antérieur à celui sur SHA-0, nous n'avons utilisé qu'un seul type de chemin auxiliaire, AP_2, selon l'approche « bits neutres ». Pour le premier et le deuxième bloc, nous avons 5 chemins auxiliaires AP_2 aux positions de bit 7, 9, 10, 12 et 16.

i	Message 1 - Premier Bloc	Message 2 - Premier Bloc
0-3	BDD77848 4FF53120 678B09E0 6C08A508	2DD77838 FFF53173 578B09E8 6C08A54B
4-7	950A1CB9 3A92154B B78CA6D8 1092006C	450A1CCB 8A92155B 478CA6BA D092002E
8-11	A3C3331B 9CE9568E 1D629EB0 7051A403	A3C3332B 7CE956CC 3D629ED0 9051A442
12-15	F04FC758 3BBE0731 76C54123 8A00A65A	D04FC708 FBBE0770 96C54151 2A00A659

i	Message 1 - Second Bloc	Message 2 - Second Bloc
0-3	A77D4037 5E854D1E 0425118C 8D5788C3	377D4047 EE854D4D 34251184 8D578880
4-7	3117F80B 300B5150 4EF7758D A4F02975	E117F879 800B5140 BEF775EF 64F02937
8-11	B4237099 9A7E7BB8 3EFFF106 DFFE9648	B42370A9 7A7E7BFA 1EFFF166 3FFE9609
12-15	D8EC1118 4A3C66FC A9FD35D5 4E6E26CC	F8EC1148 8A3C66BD 49FD35A7 EE6E26CF

Valeur de haché finale
8F2FB5E0 EA262496 653A9B0E 23D75B12 B936129B

FIG. 7.2 – Un exemple de paire de messages aboutissant à une collision pour SHA-1 réduit à 70 tours et sa valeur de haché associée [JP07a, JP07b], pour les chemins différentiels des figures 7.5 et 7.6. Les octets sont notés sous forme hexadécimale.

Finalement, nous présentons une attaque contre SHA-1 réduit à 70 étapes [JP07a, JP07b, JP07c], ne requérant qu'une dizaine d'heures de calcul sur 8 ordinateurs ordinaires. Nous avons mesuré la complexité en nombre d'appels à la fonction de compression de SHA-1, $2^{36,5}$ pour le premier bloc et 2^{39} pour le deuxième. Cela nous donne une complexité totale de $2^{39,2}$ appels à la fonction de compression de SHA-1. Théoriquement, nous attendions $2^{42-5} = 2^{37}$ appels pour le premier bloc et $2^{45-5} = 2^{40}$ pour le deuxième, ce qui est proche de la complexité mesurée. Nous donnons dans la figure 7.2 un exemple de collision pour SHA-1 réduit à 70 étapes. Dans notre évaluation de complexité, nous n'avons pas compté la génération de parties non linéaires, car elle est largement négligeable. À titre de comparaison, De Cannière *et al.* publièrent une attaque en collision contre SHA-1 réduit à 64 étapes [CR06] nécessitant 2^{35} appels à la fonction de compression, puis une attaque sur une version réduite à 70 étapes [CMR07] de complexité 2^{44} appels. Récemment, Mendel *et al.* [MRR07] ont annoncé une complexité légèrement inférieure à 2^{61} appels à la fonction de compression pour trouver une collision sur la version totale de SHA-1, et un calcul distribué pour réaliser ce but a été démarré en 2007. Ces derniers résultats n'ont pas encore été publiés mais l'on peut supposer qu'ils se fondent sur leur précédents travaux [CMR07].

L'amélioration réelle des attaques boomerang pour la version totale de SHA-1 est encore à déterminer, même si nous conjecturons qu'il peut en résulter une diminution de la complexité d'un facteur de 2^5 sur les algorithmes de recherche de collisions de Wang *et al.*. L'attaque totale étant très difficile à mettre en oeuvre et le nombre d'opérations requis étant hors de portée d'ordinateurs ordinaires, des travaux supplémentaires restent nécessaires pour confirmer cette conjecture.

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010111011100111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	0100111001001011000001010n0010u1	0	0.00	-10.00
01 :	11101101111111111100111011n1111u0	0100000011011011010100001n000000	0	0.00	-10.00
02 :	01100111011111111111101n10101101010	11110110000111110100111011n011011	0	0.00	-10.00
03 :	00110101001010 0n00001100n0uuuuu0	00111000101010010111001 00u000101	0	0.00	-10.00
04 :	111100000nu0000010101100011100000	u1100100010000000101000000u0101n1	0	0.00	-10.00
05 :	00111n0000001001100010000 0u0n1u1	n0010100110010000101010010100000	0	0.00	-10.00
06 :	10110101110110110000101u100u1001	n0100010111101000011110000u000100	0	0.00	-10.00
07 :	100unnnnnnnnn0100nu0100101u11001	0001001011110100010101100101101 0	0	0.00	-10.00
08 :	1000011100001n000n100u0n010nn001	0101101001110001000110001n0001u1	0	0.00	-10.00
09 :	00100000000000 10un00nu1u1un01100	n1010001010001111001011 00u1110n0	0	0.00	-10.00
10 :	11100110100101000nu01u10un00n100	0111101001100010010001110001 1000	0	0.00	-10.00
11 :	01111000111000110 1nuu10101000101	0111000100110111010110011u1110u0	0	0.00	-10.00
12 :	01001101011010000010u0000n110000	10110111110101-----1-----u000001	10	-4.00	-10.00
13 :	01011001110000 0----010-0-01001u0	101001010----------------n0000u1	16	-1.00	-4.00
14 :	10111100--------------1--110u011	01101-0----0--1----0---0-1-011u0	16	-2.00	11.00
15 :	10100------------------0-1-u0100	n0101-0----0--1----0---0-1-11000	16	-2.00	25.00
16 :	--01-------------------------n0011	010001110----------------00101n0	0	0.00	39.00
17 :	----------------------------1n-	n1000-0----1--1----1---0-u-10011	0	-2.00	39.00
18 :	1-----------------------------0--	01000-0----1--1----0---0-0-011u0	0	0.00	37.00
19 :	----------------------------	n00110100----------------0001011	0	0.00	37.00
20 :	----------------------------	n0110-0----1-------0-----0-000u1	0	-1.00	37.00
21 :	------------------------------n-	u1100-1------------------u-10111	0	0.00	36.00
22 :	----------------------------	00001-1------------------0-00110	0	-2.00	36.00
23 :	------------------------------n-	n1011-1----0-------0-----u-11001	0	0.00	34.00
24 :	----------------------------	u0000-0------------------1-11100	0	-2.00	34.00
25 :	------------------------------n-	01101-1------------------u-10111	0	0.00	32.00
26 :	----------------------------	u1010-1----0-------1-----0-011u0	0	-1.00	32.00
27 :	----------------------------	01001-1------------------0-01110	0	0.00	31.00
28 :	----------------------------	u0000-0------------------1-11011	0	0.00	31.00
29 :	----------------------------	u0111-0------------------0-00010	0	0.00	31.00
30 :	----------------------------	01101-1------------------1-10010	0	0.00	31.00
31 :	----------------------------	10110-1------------------0-01001	0	0.00	31.00
32 :	----------------------------	00111-1------------------1-00100	0	0.00	31.00
33 :	----------------------------	01011-1------------------1-11101	0	0.00	31.00
34 :	----------------------------	00010-0------------------0-010u0	0	-1.00	31.00
35 :	------------------------------u-	10001-0------------------n-10110	0	0.00	30.00
36 :	----------------------------	11100-0------------------0-000u1	0	-1.00	30.00
37 :	----------------------------	n0010-0------------------0-001u0	0	-1.00	29.00
38 :	------------------------------u-	n1101-0------------------n-11110	0	0.00	28.00
39 :	----------------------------	n1100-1------------------0-001n0	0	-1.00	28.00
40 :	----------------------------	n1111-0------------------0-10000	0	-1.00	27.00
41 :	----------------------------	n0010-1------------------0-11010	0	-1.00	26.00
42 :	----------------------------	n0100-0------------------1-110u1	0	-1.00	25.00
43 :	------------------------------u-	00000-1------------------n-01010	0	0.00	24.00
44 :	----------------------------	00011-0------------------0-100n0	0	-1.00	24.00
45 :	----------------------------	n0111-1------------------1-10110	0	-1.00	23.00
46 :	----------------------------	n0111-1------------------0-00010	0	-1.00	22.00
47 :	----------------------------	u0010-1------------------1-00000	0	0.00	21.00
48 :	----------------------------	01101-0------------------0-010n0	0	-1.00	21.00
49 :	------------------------------n-	11111-1------------------u-10011	0	0.00	20.00
50 :	----------------------------	01000-1------------------0-100u0	0	-1.00	20.00
51 :	----------------------------	u1110-1------------------0-10010	0	-1.00	19.00
52 :	----------------------------	n1101-1------------------1-11110	0	-1.00	18.00
53 :	----------------------------	n0001-1------------------1-001u0	0	-1.00	17.00
54 :	------------------------------u-	11011-0------------------n-11110	0	0.00	16.00
55 :	----------------------------	10001-0------------------0-000n0	0	-1.00	16.00
56 :	----------------------------	n0111-1------------------0-001n1	0	-2.00	15.00
57 :	------------------------------n-	n0110-1------------------u-11101	0	-1.00	13.00
58 :	----------------------------	u1110-1------------------1-11001	0	-2.00	12.00
59 :	------------------------------n-	u1110-0------------------u-010u1	0	-2.00	10.00
60 :	------------------------------u-	n1111-1------------------n-100n1	0	-1.00	8.00
61 :	----------------------------	01010-0------------------0-010n1	0	-1.00	7.00
62 :	----------------------------	01111-1------------------1-11111	0	0.00	6.00
63 :	----------------------------	10011-1------------------0-00010	0	0.00	6.00
64 :	----------------------------	n1000-0------------------0-10110	0	0.00	6.00
65 :	----------------------------	01000-0------------------1-00011	0	0.00	6.00
66 :	----------------------------	01000-0------------------0-101u1	0	-1.00	6.00
67 :	------------------------------u-	01001-0------------------n-01001	0	0.00	5.00
68 :	----------------------------	10001-0------------------0-100u0	0	-1.00	5.00
69 :	----------------------------	u0010-1------------------1-11000	0	0.00	4.00
70 :	----------------------------	u1010-0------------------1-011n1	0	-1.00	4.00
71 :	------------------------------n-	u0101-0------------------u-01101	0	0.00	3.00
72 :	----------------------------	00011-1------------------0-100u0	0	-1.00	3.00
73 :	----------------------------	n1010-1------------------0-11000	0	0.00	2.00
74 :	----------------------------	n1100-0------------------0-10010	0	0.00	2.00
75 :	----------------------------	u1110-1------------------1-110n1	0	-1.00	2.00
76 :	------------------------------n-	11011-1------------------u-00100	0	0.00	1.00
77 :	----------------------------	00111-0------------------1-000n1	0	-1.00	1.00
78 :	----------------------------	n0011-0------------------1-11101	0	0.00	0.00
79 :	----------------------------	u0101-0------------------1-01000	0	0.00	0.00
80 :	----------------------------				

FIG. 7.3 – Chemin différentiel pour le premier bloc de l'attaque sur SHA-0 en $2^{33,6}$ appels à la fonction de compression [MP08].

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	11101101000010101000111010101 0u1				
-3 :	01000110011100010110101100101000				
-2 :	10010000011011111010001110100111				
-1 :	11100110001011011100010100001001				
00 :	01011110101010111100001100111101	10011010011101101100111 10u1100n0	0	0.00	-21.46
01 :	u0111011010011100010111unn1010n1	00000100101110011001010 10u111101	0	0.00	-21.46
02 :	11010011001110011011u000110u0111	11101110001001000010001 00n111101	0	0.00	-21.46
03 :	11100111100001 0u000unnnnnn000100	10011010010001100110010 01n110101	0	0.00	-21.46
04 :	u0100101unn01010000u100011110110	u0111000010000000000101 01u1101u0	0	0.00	-21.46
05 :	n000un0010000011u00100000000nn0n0	u0111000011000000000001 1000010010	0	0.00	-21.46
06 :	nnn0010001011110011100n1nu1u011u	u0001011011111110100001 011u101010	0	0.00	-21.46
07 :	10nuuuuuuuuuuuuu11100n00un0u1001	11100001011111111111011000111100	0	0.00	-21.46
08 :	00011111000000000unnn11010001n001	10100010111100011011100 01u1001n1	0	0.00	-21.46
09 :	00000111111111111110001n111un111	u1001010000001111001100 10n1101u0	0	0.00	-21.46
10 :	11101101101111111110100nu111uu011	00000010111100001010011 11001011	0	0.00	-21.46
11 :	00111110010001010011011uu0n000u0	11110111011001001110101 01n1110n0	0	0.00	-21.46
12 :	01000110100011100011111 1nuu1u011	00110111111 1----------n111010	13	-1.54	-21.46
13 :	101010000000----0--------01111u0	01010----------u0000u1	20	-3.00	-10.00
14 :	00110001----------1010010	10010------0----1--------00110n1	18	-2.00	7.00
15 :	10011----------10101n0	n0110------0----1--------0111110	18	-1.00	23.00
16 :	0----------011000	10000----------11010u1	0	-2.00	40.00
17 :	1----------n-	u1000------1----1--------u100111	0	-1.00	38.00
18 :	0----------	01100------1----0--------01111u0	0	0.00	37.00
19 :	----------	n1010----------1110100	0	0.00	37.00
20 :	----------	u1000------1----------10000n1	0	-1.00	37.00
21 :	----------n-	n1101----------u010011	0	0.00	36.00
22 :	----------	11101----------1100010	0	-2.00	36.00
23 :	----------n-	u1011------0----------u110101	0	0.00	34.00
24 :	----------	n1001----------0010110	0	-2.00	34.00
25 :	----------u-	00010----------n001011	0	0.00	32.00
26 :	----------	u0001------1----------01110u0	0	-1.00	32.00
27 :	----------	10111----------0011001	0	0.00	31.00
28 :	----------	n1110----------1101001	0	0.00	31.00
29 :	----------	u0000----------0010100	0	0.00	31.00
30 :	----------	01000----------0001001	0	0.00	31.00
31 :	----------	01011----------1000101	0	0.00	31.00
32 :	----------	00101----------1010111	0	0.00	31.00
33 :	----------	11000----------0010001	0	0.00	31.00
34 :	----------	01110----------00000n0	0	-1.00	31.00
35 :	----------n-	10101----------u101001	0	0.00	30.00
36 :	----------	10011----------10110u1	0	-1.00	30.00
37 :	----------	n1000----------01100u0	0	-1.00	29.00
38 :	----------u-	n1001----------n010100	0	0.00	28.00
39 :	----------	n0001----------11000n0	0	-1.00	28.00
40 :	----------	u0101----------1001001	0	-1.00	27.00
41 :	----------	n0100----------0010111	0	-1.00	26.00
42 :	----------	u0000----------01100u1	0	-1.00	25.00
43 :	----------u-	00111----------n101101	0	0.00	24.00
44 :	----------	10001----------01011n0	0	-1.00	24.00
45 :	----------	n0011----------1010000	0	-1.00	23.00
46 :	----------	n0011----------1100111	0	-1.00	22.00
47 :	----------	n0011----------0011000	0	0.00	21.00
48 :	----------	11101----------10011u0	0	-1.00	21.00
49 :	----------u-	01010----------n001000	0	0.00	20.00
50 :	----------	01110----------11100n0	0	-1.00	20.00
51 :	----------	n0111----------0111000	0	-1.00	19.00
52 :	----------	n0001----------1101011	0	-1.00	18.00
53 :	----------	n0100----------11100u0	0	-1.00	17.00
54 :	----------u-	11000----------n000010	0	0.00	16.00
55 :	----------	00111----------00001n0	0	-1.00	16.00
56 :	----------	u1100----------10001u0	0	-2.00	15.00
57 :	----------u-	u0001----------n110000	0	-1.00	13.00
58 :	----------	n1000----------1101011	0	-2.00	12.00
59 :	----------u-	u1111----------n0000u1	0	-2.00	10.00
60 :	----------u-	n0010----------n0100n0	0	-1.00	8.00
61 :	----------	01100----------10100n1	0	-1.00	7.00
62 :	----------	11001----------0101000	0	0.00	6.00
63 :	----------	01100----------0000100	0	0.00	6.00
64 :	----------	n0011----------0101001	0	0.00	6.00
65 :	----------	00101----------0101000	0	0.00	6.00
66 :	----------	01011----------11101n0	0	-1.00	6.00
67 :	----------n-	11111----------u100000	0	0.00	5.00
68 :	----------	11110----------10100n1	0	-1.00	5.00
69 :	----------	n0100----------1010011	0	0.00	4.00
70 :	----------	n0010----------00011n0	0	-1.00	4.00
71 :	----------n-	n0100----------u100001	0	0.00	3.00
72 :	----------	10011----------10101u1	0	-1.00	3.00
73 :	----------	n1001----------0010111	0	0.00	2.00
74 :	----------	n0101----------1101110	0	0.00	2.00
75 :	----------	u1111----------11001n1	0	-1.00	2.00
76 :	----------n-	01100----------u111110	0	0.00	1.00
77 :	----------	00001----------11010n0	0	-1.00	1.00
78 :	----------	n0111----------1101000	0	0.00	0.00
79 :	----------	n0001----------0110011	0	0.00	0.00
80 :	----------				

FIG. 7.4 – Chemin différentiel pour le second bloc de l'attaque sur SHA-0 en $2^{33,6}$ appels à la fonction de compression [MP08].

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00001111010010111000011111000011				
-3 :	01000000110010010101000111011000				
-2 :	01100010111010111011001111111010				
-1 :	11101111110011011010101110001001				
00 :	01100111010001010010001100000001	n11u00000011110100101--10nuu1001	2	0.00	10.42
01 :	0nnn11111111000111000--1111n1100	u0nu11011111111---1111110u0u10nu	3	-3.00	12.42
02 :	n00n00101110100010001 0n11u10011u	01nu01001101011000 00---01111n110	3	-2.00	12.42
03 :	000u11101111100000nnn-0100101uu1	010111100101100--0-------n1011un	9	-8.00	13.42
04 :	01101111111001nnn011-0010u1101111	nn0n0000010000-110011000-unn10u1	2	-2.00	14.42
05 :	10unu1uu01u0000000u01001unnn1n10	u0nu11011100110000001----00u0010	4	-4.00	14.42
06 :	1n110n00000010100010100n1un10110	nnnn011010101000110011101uu001u1	0	0.00	14.42
07 :	u1u01u1100101000010n0111n1nn11n1n	un00110010110110001 0----0u1100n0	4	-4.00	14.42
08 :	1nnnnnnnnnnnnnn0100u00n00001110n	000100111010101000111-0110un0111	1	-1.00	14.42
09 :	11110111000000010101110111111un11	unn0010001110111 01-011100n1000n1	1	0.00	14.42
10 :	00010110111111111 0-1u0111nu1u111u	11n0110111010-11010111111nn10110	1	-1.00	15.42
11 :	u10111010001101000000001010011101	unu0000000000000-1-011000u00011u	2	0.00	15.42
12 :	1000111001001101---01n10nuu1111u	10u000101110-0-------1111u1n1101	8	-2.00	17.42
13 :	1100110011110--------nuuuu1uuun1	nn1111011011-1-1--------1u10100n	10	-2.00	23.42
14 :	n111110001--------------10n01011	unu0100111-1011--------00nuu10n1	9	-3.00	31.42
15 :	010010-11---------------011001u11	n0n11101010-1---1----------000nn	14	-1.00	37.42
16 :	n-011----------------------u0100	nn0101011111-1-0--------1000n00n0	0	-1.00	50.42
17 :	--1----------------------1--1n1	1un111000-0011----------1u01111u	0	-2.42	49.42
18 :	u-0--------------------------1--	nn00100000-1---0----------nu01u1	0	0.00	47.00
19 :	----------------------------	11u0110011-1-0----------1111011n	0	0.00	47.00
20 :	----------------------------	uu110000-00-1----------001111n0	0	-2.00	47.00
21 :	----------------------------u-	nun010011-1---1----------n1110u0	0	-2.00	45.00
22 :	----------------------------u-	1nu000000-0-1----------10n0111u0	0	-1.00	43.00
23 :	----------------------------	u100101-00-0--------------0110u1	0	-1.00	42.00
24 :	----------------------------	00000100-1---1----------00010111	0	0.00	41.00
25 :	----------------------------	10110000-1-0----------101001110	0	0.00	41.00
26 :	----------------------------	n10110-00-0--------------1101111	0	0.00	41.00
27 :	----------------------------	1110101-0---0----------0100111n0	0	-1.00	41.00
28 :	----------------------------n-	1001011-1----------10u001100	0	0.00	40.00
29 :	----------------------------	01100-10-0----------------111010	0	-2.00	40.00
30 :	----------------------------n-	u11011-0---0----------010u110111	0	0.00	38.00
31 :	----------------------------	n01110-1----------010001	0	-2.00	38.00
32 :	----------------------------n-	1111-11-0----------u110100	0	0.00	36.00
33 :	----------------------------	u1011-0---1----------00010000n	0	-2.00	36.00
34 :	----------------------------u	00110-1----------nu01011	0	0.00	34.00
35 :	----------------------------	n11-00-0----------010010nu	0	-1.00	34.00
36 :	----------------------------	0n11-1----------0100111n0	0	-2.00	33.00
37 :	----------------------------n-	un10-1----------u100001	0	-1.00	31.00
38 :	----------------------------	nu-10-0----------0011011u0	0	-1.00	30.00
39 :	----------------------------	u11-1----------0110000	0	0.00	29.00
40 :	----------------------------	n00-1----------11011010	0	-1.00	29.00
41 :	----------------------------	u-10-1----------11001110n1	0	-1.00	28.00
42 :	----------------------------n-	00-0----------0u110000	0	0.00	27.00
43 :	----------------------------	11-0----------11010u1	0	-1.00	27.00
44 :	----------------------------	x10-0----------1100111101	0	-1.00	26.00
45 :	----------------------------	u-1----------1110101	0	-1.00	25.00
46 :	----------------------------	n-0----------00001101	0	0.00	24.00
47 :	----------------------------	10-1----------11110n-	0	-1.00	24.00
48 :	----------------------------n-	-1----------1u100000	0	0.00	23.00
49 :	----------------------------	-0----------010111010	0	-2.00	23.00
50 :	----------------------------n-	n-1----------0u1111-1	0	-1.00	21.00
51 :	----------------------------	u----------010001u-	0	-2.00	20.00
52 :	----------------------------	0----------101110101-	0	-1.00	18.00
53 :	----------------------------	x0----------10110-11	0	-1.00	17.00
54 :	----------------------------	x----------1111111-1	0	0.00	16.00
55 :	----------------------------	----------001101-1	0	0.00	16.00
56 :	----------------------------	1----------11111-01-	0	0.00	16.00
57 :	----------------------------	----------00101-1-	0	0.00	16.00
58 :	----------------------------	----------000001-1-	0	0.00	16.00
59 :	----------------------------	----------010-01-0	0	0.00	16.00
60 :	----------------------------	----------11110-0--	0	0.00	16.00
61 :	----------------------------	----------0110-n-1	0	-1.00	16.00
62 :	----------------------------n--	----------0u0-01-0-	0	0.00	15.00
63 :	----------------------------	----------001-1x--	0	-1.00	15.00
64 :	----------------------------	----------0000-u--x	0	-2.00	14.00
65 :	----------------------------u---	----------n1-11-u-x	0	-2.00	12.00
66 :	----------------------------u--	----------1n1-0x--u	0	-1.00	10.00
67 :	----------------------------	----------101-u-xx-	0	-3.00	9.00
68 :	----------------------------u----	----------n0-00-n-xx	0	-3.00	6.00
69 :	----------------------------n---	----------u0-0xx-nx	0	-3.00	3.00
70 :	----------------------------x---				

FIG. 7.5 – Chemin différentiel pour le premier bloc de l'attaque sur SHA-1 réduit à 70 tours en $2^{36,5}$ appels à la fonction de compression [JP07a, JP07b, JP07c].

i	∇A_i	∇W_i	$L(i)$	$P_i(i)$	$N_e(i)$
-4 :	00111110001100010010011110011u10				
-3 :	01101010111101111001010011000000				
-2 :	00110011001111010000110111u0101				
-1 :	00100111110111010000101001nu101				
00 :	00011001010111100110101 1unn1n010	n10u1000101000000----0-10nnu0011	5	-3.00	8.60
01 :	1nnn10110110101001 01-u-n0n010n1u	n0nn00100100010-0-0011010u0n01nn	2	-2.00	10.60
02 :	nn1u00101010011 10uu1n00011101u0u	00nn0100101010001 01--0101000u010	2	-2.00	10.60
03 :	1unn00111011u1uu100111110u10111n	01100110010001111111110110-n0111nn	1	-1.00	10.60
04 :	0010100uu0n01000001u1010nn11uu0n	un1n0100101111100000100 10-nuu00n1	1	-1.00	10.60
05 :	n00101111000011u0000101n11011u01	n0uu10110011011010111----11u1000	4	-4.00	10.60
06 :	110011unnnn1111100n01110u101001n	uunu00101100111111101 0001un001n0	0	0.00	10.60
07 :	11100000001001n0100n0u00n1000nn00	uu00110000001010110001000n0111n1	0	0.00	10.60
08 :	n0100100111111111n0u110n0010n110n	00111001100111100111101110nn0010	0	0.00	10.60
09 :	u1100100000010111110011111u1n0u01	uuu1001001100011111-----0n1000u1	5	-1.00	10.60
10 :	1100011101010 10nuu1-0---110111u1	10u0010110110-0-0--------0uu00011	9	-8.00	14.60
11 :	u0110110011101100010--1-101n10n1	unn000000100000--0---1-11u00111n	6	-5.46	15.60
12 :	00101010110001110011--0-10n1n11n	00n1110010111 11----------n0n0000	10	-1.14	16.14
13 :	11010010101111--------u--1unnnu0	uu1110010011-1-0----------n11111n	11	-4.00	25.00
14 :	n00111100001------------u-00110u	uun0010100-1110----------nuu01u0	11	-4.00	32.00
15 :	100001000----------------1-010n01	u0u10110111-0-0-1---------0100uu	12	-1.00	39.00
16 :	u0011----------------------101110	nn1110010 10-0-0----------010n10u1	0	-2.00	50.00
17 :	--1--0----------------------01n1	1nu001101-1110----------1n01101n	0	-1.00	48.00
18 :	n-0--------------------------1--	nu10011010-0-0-0---------0nu01u1	0	0.00	47.00
19 :	--------------------------1--	11n0100011-0-0----------1100001u	0	0.00	47.00
20 :	------------------------------	un011000-1101-----------110111u1	0	-2.00	47.00
21 :	----------------------------u-	uun100010-0-0-0---------1n0110n0	0	-2.00	45.00
22 :	----------------------------n-	1un011010-0-1-----------0u1001u1	0	-1.00	43.00
23 :	------------------------------	n010000-11-0------------11001n0	0	-1.00	42.00
24 :	------------------------------	11101000-1-1-1---------001100100	0	0.00	41.00
25 :	------------------------------	00110011-0-0-----------000100000	0	0.00	41.00
26 :	------------------------------	u11111-01-0------------00100001	0	0.00	41.00
27 :	------------------------------	1011001-1-0-0-----------101011u0	0	-1.00	41.00
28 :	----------------------------u-	1110010-1-0------------1n010010	0	0.00	40.00
29 :	------------------------------	11000-00-0---------------0101000	0	-2.00	40.00
30 :	----------------------------n-	n00001-1---0------------0u101001	0	0.00	38.00
31 :	------------------------------	u01010-0----------------1111111	0	-2.00	38.00
32 :	----------------------------u-	1110-11-0---------------0n100000	0	0.00	36.00
33 :	------------------------------	u0110-0---1------------00010000n	0	-2.00	36.00
34 :	----------------------------u	01010-0----------------0nu01101	0	0.00	34.00
35 :	------------------------------	u00-10-0---------------1101010un	0	-1.00	34.00
36 :	------------------------------	0n00-0---0-------------0000101n0	0	-2.00	33.00
37 :	----------------------------n-	uu00-0----------------1u110010	0	-1.00	31.00
38 :	------------------------------	nu-10-0----------------1100000n1	0	-1.00	30.00
39 :	------------------------------	n11-0------------------01011111	0	0.00	29.00
40 :	------------------------------	n11-0----------------100101110	0	-1.00	29.00
41 :	------------------------------	u-00-0----------------0010111u0	0	-1.00	28.00
42 :	----------------------------u-	01-1------------------11n000010	0	0.00	27.00
43 :	------------------------------	00-1------------------000001n1	0	-1.00	27.00
44 :	------------------------------	x10-0----------------101100010	0	-1.00	26.00
45 :	------------------------------	u-1--------------------01001111	0	-1.00	25.00
46 :	------------------------------	n-0----------------110011011	0	0.00	24.00
47 :	------------------------------	01-1--------------------110001n-	0	-1.00	24.00
48 :	----------------------------n-	-1---------------------01u011000	0	0.00	23.00
49 :	------------------------------	-1-------------------0010011111	0	-2.00	23.00
50 :	----------------------------n-	u-1--------------------01u1111-0	0	-1.00	21.00
51 :	------------------------------	n----------------------1000010u-	0	-2.00	20.00
52 :	------------------------------	1----------------------001101000-	0	-1.00	18.00
53 :	------------------------------	x1---------------------000111-01	0	-1.00	17.00
54 :	------------------------------	x----------------------00011111-0	0	0.00	16.00
55 :	------------------------------	-----------------------0001010-0	0	0.00	16.00
56 :	------------------------------	0----------------------001001-01-	0	0.00	16.00
57 :	------------------------------	----------------------101110-0-	0	0.00	16.00
58 :	------------------------------	----------------------0001101-0-	0	0.00	16.00
59 :	------------------------------	----------------------1101-01-1	0	0.00	16.00
60 :	------------------------------	----------------------101011-0--	0	0.00	16.00
61 :	------------------------------	----------------------11011-u-0	0	-1.00	16.00
62 :	----------------------------u--	----------------------01n0-10-0-	0	0.00	15.00
63 :	------------------------------	----------------------1100-0x--	0	-1.00	15.00
64 :	------------------------------	----------------------10000-n--x	0	-2.00	14.00
65 :	--------------------------n---	----------------------1u1-10-n-x	0	-2.00	12.00
66 :	--------------------------n--	----------------------11u0-0x--n	0	-1.00	10.00
67 :	------------------------------	----------------------0100-n-xx-	0	-3.00	9.00
68 :	-------------------------n----	----------------------0u1-11-u-xx	0	-3.00	6.00
69 :	--------------------------u---	----------------------0n0-0xu-ux	0	-3.00	3.00
70 :	--------------------------u---				

FIG. 7.6 – Chemin différentiel pour le second bloc de l'attaque sur SHA-1 réduit à 70 tours en 2^{39} appels à la fonction de compression [JP07a, JP07b, JP07c].

TROISIÈME PARTIE

Cryptanalyses de nouvelles fonctions de hachage

CHAPITRE 8
Cryptanalyse de GRINDAHL

Sommaire

À la suite des récentes avancées dans le domaine de la cryptanalyse des fonctions de hachage de la famille SHA ou MD, de nouvelles fonctions de hachage ont été proposées. Parmi ces algorithmes figure GRINDAHL, conçu par Knudsen, Rechberger et Thomsen et publié récemment à FSE 2007 [KRT07]. L'une de ses particularités est qu'il suit les mêmes principes de construction que RIJNDAEL [DR02], et présente une efficacité légèrement meilleure que SHA-256. Nous montrons dans ce chapitre que la version 256 bits de GRINDAHL n'est pas résistante à la recherche de collisions. En effet, avec une attaque nécessitant approximativement 2^{112} appels à la fonction de hachage, il est possible de générer deux messages distincts qui aboutissent au même haché. Ces travaux ont fait l'objet d'une publication à la conférence ASIACRYPT 2007 [Pey07] qui a obtenu le prix du meilleur article.

8.1 Description de GRINDAHL

GRINDAHL est une famille de fonctions de hachage fondée sur la stratégie *Concaténation-Permutation-Troncature*. La permutation utilise les principes de l'algorithme de chiffrement par blocs RIJNDAEL [DR02], connu pour être celui choisi lors du processus d'élection de l'Advanced Encryption Standard (AES) [N-aes] du NIST. Deux algorithmes sont définis : une version produisant des hachés de taille 256 bits et une autre version produisant des hachés de taille 512 bits. De plus, il existe un mode *fonction de compression* n'acceptant que des entrées de taille fixe, à utiliser avec n'importe quel algorithme d'extension de domaine. Nous donnons dans cette

section une rapide description de la fonction de hachage GRINDAHL avec 256 bits de sortie. Pour une description plus détaillée, on se reportera à [KRT07].

Soit $n = 256$ le nombre de bits de sortie de la fonction de hachage H, utilisant un *état interne* s de 48 octets (384 bits), et soit M le message à hacher (préalablement complété par un suffixe de façon appropriée). M est tout d'abord divisé en m blocs $M_1, \ldots, M_m$ de 4 octets chacun (32 bits). À chaque itération k, le bloc de message M_k sera utilisé pour mettre à jour l'état interne s_{k-1}. Nous appelons *état interne étendu* $\hat{s}_k$ la concaténation du bloc de message M_{k+1} et de l'état interne s_k :

$$\hat{s}_k = M_{k+1} || s_k.$$

Nous avons ainsi $|\hat{s}_k| = (4 + 48) \times 8 = 416$ bits. Nous notons $\text{TRONC}_t^l(x)$ les t bits de poids faible de x et $\text{TRONC}_t^m(x)$ les t bits de poids fort de x. Soit $P : \{0,1\}^{416} \longmapsto \{0,1\}^{416}$ une permutation non linéaire, et soit s_0 *l'état interne initial* définit par $s_0 = \{0\}^{384}$.

Chaque itération k, avec $0 < k < m$, est définie par $s_k = \text{TRONC}_{384}^m(P(\hat{s}_{k-1}))$. Pour la dernière itération, la troncature est omise : $\hat{s}_m = P(\hat{s}_{m-1})$. Finalement, on applique huit *tours à blanc* $\hat{s}_k = P(\hat{s}_{k-1})$, pour $m < k \leqslant m+8$, et la sortie de la fonction de hachage est $\text{TRONC}_{256}^l(\hat{s}_{m+8})$.

La description n'est pas complète, car P n'a pas encore été définie. Cette permutation utilise les principes de conception de RIJNDAEL et l'état étendu $\hat{s}$ est donc considéré comme une matrice d'octets. Cependant, au lieu de manipuler une matrice $(4, 4)$ d'octets comme dans le cas de RIJNDAEL nous aurons une matrice α de 4 lignes et 13 colonnes pour la version 256 bits de GRINDAHL. L'élément de la matrice α situé sur la i-ième ligne et la j-ième colonne est un octet noté $\alpha_{i,j}$. Ainsi, nous obtenons :

$$\alpha = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,12} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,12} \\ \alpha_{2,0} & \alpha_{2,1} & \cdots & \alpha_{2,12} \\ \alpha_{3,0} & \alpha_{3,1} & \cdots & \alpha_{3,12} \end{pmatrix}.$$

Chaque octet peut être vu comme un élément du corps GF(2^8). En divisant l'état interne étendu $\hat{s}$ en 52 octets $x_0, \ldots, x_{51}$, il est possible de définir la conversion de $\hat{s}$ à α par $\alpha_{i,j} = x_{i+4\times j}$ (cette application possède un inverse naturel). Ainsi, avant chaque itération, la première colonne de la matrice α est écrasée par les 4 octets du bloc de message arrivant en entrée. La permutation P est alors définie par

$$P(\alpha) = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \circ \text{AddConstant}(\alpha).$$

MixColumns. Cette transformation est exactement celle définie pour l'algorithme RIJNDAEL. Plus précisément, nous appliquons une transformation à chaque colonne de la matrice de l'état interne étendu. Aux quatre octets d'entrée $(\alpha_{0,j}, \alpha_{1,j}, \alpha_{2,j}, \alpha_{3,j})$, nous faisons correspondre quatre nouveaux octets $(\alpha'_{0,j}, \alpha'_{1,j}, \alpha'_{2,j}, \alpha'_{3,j})$:

$$\begin{cases} \alpha'_{0,j} &= 2 \times \alpha_{0,j} + \alpha_{3,j} + \alpha_{2,j} + 3 \times \alpha_{1,j} \\ \alpha'_{1,j} &= 2 \times \alpha_{1,j} + \alpha_{0,j} + \alpha_{3,j} + 3 \times \alpha_{2,j} \\ \alpha'_{2,j} &= 2 \times \alpha_{2,j} + \alpha_{1,j} + \alpha_{0,j} + 3 \times \alpha_{3,j} \\ \alpha'_{3,j} &= 2 \times \alpha_{3,j} + \alpha_{2,j} + \alpha_{1,j} + 3 \times \alpha_{0,j} \end{cases}$$

les lois d'addition et de multiplication étant celles du corps GF(2^8).

ShiftRows. Cette transformation décale les octets sur les lignes de la matrice de façon cyclique, le nombre de positions de décalage dépendant de la ligne considérée. Ainsi, la i-ième ligne est décalée de ρ_i positions sur la droite, avec $\rho_0 = 1$, $\rho_1 = 2$, $\rho_2 = 4$ et $\rho_3 = 10$:

$$\alpha'_{i,j} = \alpha_{i,(\rho_i + j) \bmod 13}.$$

SubBytes. Il s'agit de la seule partie non linéaire de la permutation. Elle est définie exactement comme la fonction SubBytes pour l'algorithme RIJNDAEL : tous les éléments de l'état interne étendu sont mis à jour par application d'une boîte de substitution $\alpha'_{i,j} = SBOX(\alpha_{i,j})$, donnée dans le tableau 8.1.

b \ a	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

TAB. 8.1 – Boîte de substitution $SBOX$ utilisée dans RIJNDAEL et GRINDAHL. À tout octet $\alpha_{i,j} = 16 \times b + a$, on fait correspondre l'octet $\alpha'_{i,j}$ donné par le tableau. Les valeurs sont notées en hexadécimal.

AddConstant. Cette fonction est simplement définie par $\alpha_{3,12} \longleftarrow \alpha_{3,12} \oplus \texttt{01}$, où `01` est la valeur hexadécimale de 1.

Une vue schématique de la version 256 bits de GRINDAHL est donnée dans la figure 8.1. La version 512 bits de GRINDAHL est fondée sur les mêmes principes que celle de 256 bits, à la différence près que l'état interne étendu est plus grand (8 lignes au lieu de 4).

Le mode *fonction de compression* pour GRINDAHL-256 consiste simplement à hacher 40 blocs de message de 4 octets pour chaque appel à la fonction de compression. Pour la suite, sauf indication contraire, l'état interne sera considéré dans sa forme matricielle.

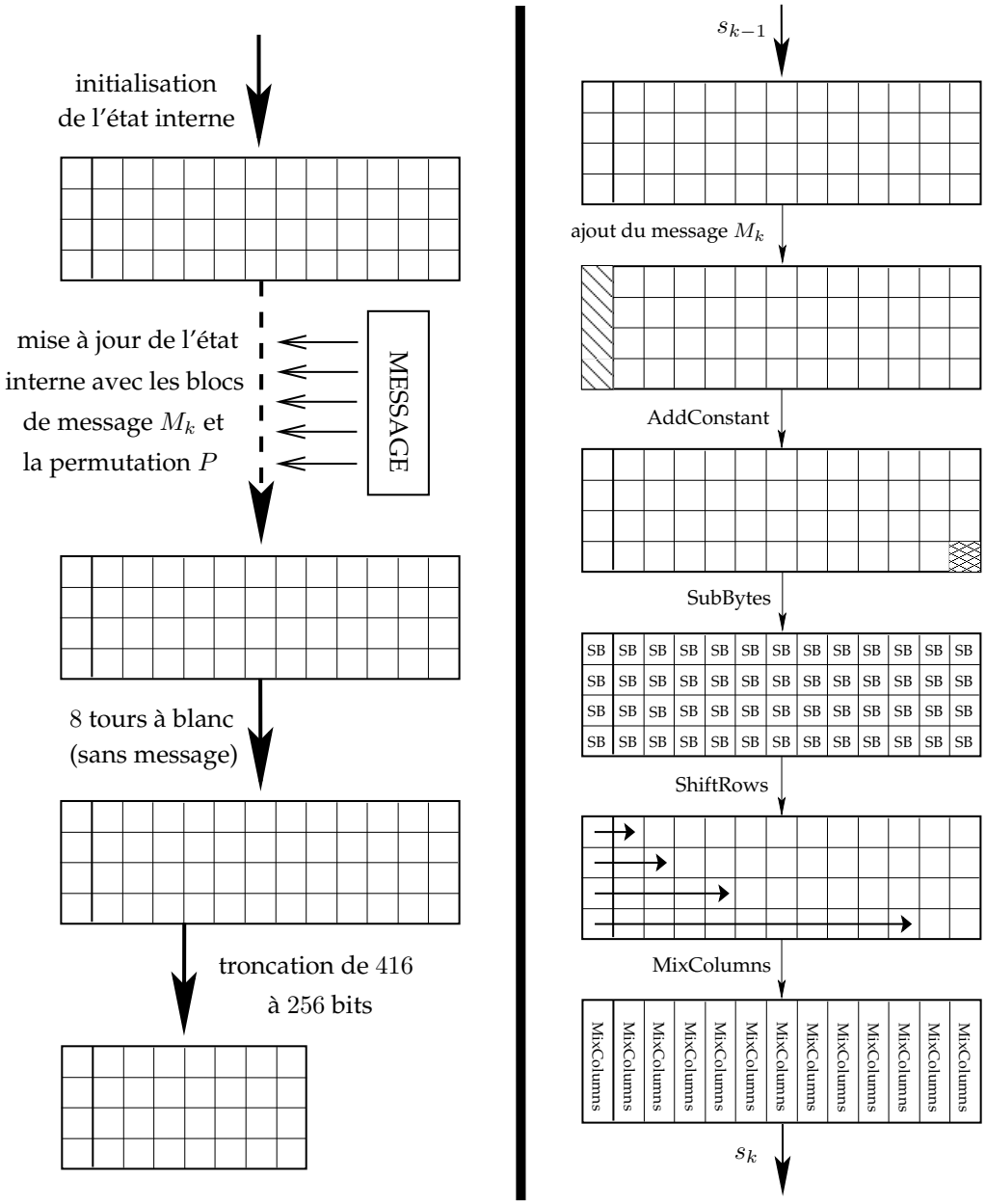

FIG. 8.1 – Vue schématique de la fonction de hachage GRINDAHL. La partie de gauche représente la vue générale de l'algorithme tandis que celle de droite représente la permutation interne P utilisée. Chaque cellule correspond à un octet de l'état interne.

8.2 Analyse générale

Dans cette partie, nous chercherons tout d'abord un moyen de trouver un bon chemin différentiel pour la version 256-bit de GRINDAHL. Plus exactement, nous cherchons un chemin de k itérations commençant par s_0 et tel que deux messages différents M et M' (mais de même longueur m blocs pour éviter d'éventuels problèmes dus au rembourrage) aboutissent au même haché, c'est-à-dire $\text{TRONC}^l_{256}(\hat{s}^M_{m+8}) = \text{TRONC}^l_{256}(\hat{s}^{M'}_{m+8})$. Notre objectif est de mettre au point des attaques recherchant des collisions ou des secondes préimages. Trouver un chemin différentiel incluant les tours à blanc semble être très difficile puisqu'aucun mot de message n'y est inséré, ce qui laisse très peu de contrôle à l'attaquant sur cette partie. Cependant, même si le nombre de bits sur lesquels l'on tente de trouver une collision est alors plus grand (l'état interne est plus grand que le haché final), le problème semble beaucoup plus simple lorsque l'on essaye de trouver une *collision interne* : un chemin différentiel excluant les tours à blanc, c'est-à-dire tel que $\hat{s}^M_m = \hat{s}^{M'}_m$. Nous expliquons ici comment trouver un tel chemin.

8.2.1 Les différences tronquées

Dans l'article original [KRT07], les concepteurs de GRINDAHL mentionnent une faiblesse possible de l'algorithme, découverte par un membre anonyme du comité de programme. La technique de cryptanalyse est assez naturelle : l'attaquant ne s'occupe pas des valeurs réelles des différences présentes dans les octets de l'état interne, mais regarde seulement s'il y a une différence ou non. Nous appelons ce type de différences *différences tronquées* en référence aux différences tronquées très similaires utilisées par Knudsen dans [Knu94]. Cette façon de voir l'algorithme permet de simplifier son analyse, du fait notamment que les fonctions AddConstant et SubBytes peuvent à présent être omises, car elles sont transparentes du point de vue des différences tronquées. Ensuite, l'attaquant tente de trouver un chemin composé de différences tronquées (appelé *chemin différentiel tronqué*) pour lequel à chaque itération le nombre d'octets actifs (octets avec une différence tronquée non nulle) est faible. Durant ce chemin différentiel tronqué, les différences tronquées ne peuvent être effacées qu'à deux des étapes d'une itération : lors de la transformation MixColumns ou lors de la troncature à la fin de l'itération. Par une utilisation intelligente de la fonction MixColumns le nombre de différences tronquées dans une colonne peut être réduit et leurs positions changées. Il est cependant impossible d'effacer toutes les différences tronquées d'une colonne à la fois. Une différence tronquée est aussi effacée si elle se trouve dans la première colonne de la matrice α à la fin d'une itération, et ce, du fait de la troncature. Cette deuxième possibilité est prise en compte directement lors de l'établissement du chemin différentiel final, et non durant la recherche d'une paire de messages valide. Cependant, pour l'élimination des différences tronquées lors de l'application de MixColumns, il est possible de jouer sur les blocs de message insérés à chaque itération pour forcer le comportement recherché de la fonction (voir section 8.2.2). Dans ce cas, les octets du message agissent comme des octets *actifs/passifs* en ce sens qu'ils n'affectent pas certaines parties de l'état interne pour un nombre limité d'itérations (voir section 8.2.3). La faisabilité de cette méthode fut laissée à l'état de problème ouvert et nous montrons en section 8.2.4 qu'il existe un meilleur moyen pour trouver des collisions sur GRINDAHL.

8.2.2 Analyse de la fonction MixColumns

La transformation MixColumns utilisée dans GRINDAHL est la même que celle définie dans RIJNDAEL. Comme elle est dérivée d'un code MDS (Maximum Distance Separable), une propa-

gation maximale des différences est assurée. Plus précisément, la somme du nombre des octets actifs en entrée et en sortie est toujours supérieure ou égale à 5 (ou égale à 0 dans le cas où aucun octet n'est actif en entrée), ce qui implique que le nombre de différences tronquées non nulles en entrée et en sortie est aussi supérieur ou égal à 5.

Plus formellement, soit $V = (A, B, C, D)$ un vecteur d'entrée de 4 octets A, B, C et D. Soit $W = (A', B', C', D')$ un vecteur de sortie de 4 octets A', B', C' et D'. On note $MC : V \longmapsto W$ où $MC : (A, B, C, D) \longmapsto (A', B', C', D')$ la fonction MixColumns et l'on note $D_i(V_1, V_2)$ la fonction retournant 1 si les valeurs du i-ième octet des vecteurs V_1 et V_2 sont distinctes, et 0 dans le cas contraire. Enfin, $ND(V_1, V_2)$ retourne le nombre d'octets distincts, c'est-à-dire $ND(V_1, V_2) = \#\{i | D_i(V_1, V_2) = 1\}$. Nous obtenons ainsi que si $W_1 = MC(V_1)$ et $W_2 = MC(V_2)$ avec $V_1 \neq V_2$, alors

$$ND(V_1, V_2) + ND(W_1, W_2) \geqslant 5.$$

Une autre propriété intéressante de cette fonction est que pour tout octet d'entrée et tout octet de sortie la fonction partielle associée est une permutation quelle que soit la valeur des trois autres octets d'entrée. Donc avec $V_1 \neq V_2$ tirés uniformément et aléatoirement dans $\{0, 1\}^{4\times 8}$, $W_1 = MC(V_1)$ et $W_2 = MC(V_2)$, nous avons pour tout $1 \leqslant i \leqslant 4$:

$$P_D = P[D_i(W_1, W_2) = 0] = \frac{256^3 - 1}{256^4 - 1} \simeq 2^{-8}, \tag{8.1}$$

$$\overline{P_D} = P[D_i(W_1, W_2) = 1] = 1 - P_D \simeq 1 - 2^{-8}. \tag{8.2}$$

Notre but ici est de calculer la probabilité qu'un masque de différentielles tronquées fixé en entrée donne un masque de différentielles tronquées fixé en sortie, ce qui nous servira plus tard à calculer la probabilité de succès d'un chemin différentiel tronqué. Par exemple, on souhaite savoir quelle est la probabilité que deux mots d'entrée V_1 et V_2 distincts sur leurs deux premiers octets donnent en sortie deux mots distincts sur leurs trois premiers octets après application de MixColumns. Ces probabilités peuvent être calculées de deux façons : formellement ou empiriquement, en testant exhaustivement toutes les valeurs d'entrée. Puisque la fonction MixColumns est linéaire, considérer des valeurs ou des différences revient au même (durant le test, au lieu de chercher des différences ou des non-différences, on cherche des valeurs nulles ou non nulles). Nous donnons dans le tableau 8.2 une approximation de ces probabilités.

D_I \ D_O	0	1	2	3	4
0	0	- ∞	- ∞	- ∞	- ∞
1	- ∞	- ∞	- ∞	- ∞	0
2	- ∞	- ∞	- ∞	-8	0
3	- ∞	- ∞	-16	-8	0
4	- ∞	-24	-16	-8	0

TAB. 8.2 – Probabilité approchée que deux mots d'entrée de 4 octets chacun, avec D_I octets distincts sur des positions prédéfinies, donnent après application de MixColumns deux mots de sortie de 4 octets chacun avec D_O octets distincts sur des positions prédéfinies. Les valeurs sont notées en échelle logarithmique de base 2.

8.2.3 Existence des octets de contrôle

Modifier des octets du message influera évidemment très rapidement sur l'état interne, mais pas immédiatement. Pour chaque octet modifié du message M_k, nous donnons dans le tableau 8.3 les colonnes de s (dans sa représentation matricielle α) affectées par cette modification après 1, 2 et 3 itérations. À partir de 4 itérations, tout octet du message affecte entièrement l'état interne. Cette propriété d'octets *actifs/passifs* va nous permettre d'attaquer différentes colonnes de différentes itérations de façon plus ou moins indépendante. En fait, nous allons contrôler indépendamment le comportement de certaines transitions de MixColumns grâce aux octets actifs/passifs.

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k		✓											
B_k			✓										
C_k					✓								
D_k											✓		

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k			✓	✓		✓						✓	
B_k				✓	✓		✓						✓
C_k		✓				✓	✓		✓				
D_k		✓						✓				✓	✓

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k	✓		✓	✓	✓	✓	✓	✓	✓	✓			✓
B_k	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		
C_k			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
D_k	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓

TAB. 8.3 – Influences sur les colonnes de l'état interne étendu d'une modification d'un octet du bloc de message $M_k = (A_k, B_k, C_k, D_k)$ entrant à l'itération k. Nous représentons une colonne affectée (ou active) par ✓ et le cas contraire par un vide. Le premier tableau montre l'influence sur s_{k-1}, le deuxième sur s_k et le troisième sur s_{k+1}.

8.2.4 Stratégie générale

Nous disposons à présent de tous les outils nécessaires pour construire un chemin différentiel tronqué et évaluer sa probabilité de succès. La question qui demeure est celle de la méthode de construction. L'intuition naturelle d'un cryptanalyste (comme suggéré par le membre anonyme du comité de programme de la conférence FSE 2007) est de chercher à maintenir un nombre faible de différences tronquées non nulles durant le chemin, pour ainsi augmenter sa probabilité de succès. Cependant, trouver un tel chemin semble être très complexe, et il est possible de s'en convaincre par les propriétés données dans l'article original [KRT07] :

Propriété 1 *Une collision interne pour* GRINDAHL*-256 requiert au moins* 5 *itérations.*

Cette propriété peut être vérifiée grâce à une recherche exhaustive de type *« rencontre au milieu »* comme expliqué dans l'article original. Cependant, avec une légère amélioration de

cet algorithme, il est possible de prouver qu'une collision interne pour GRINDAHL-256 requiert au moins 6 itérations.

Propriété 2 *Tout chemin différentiel commençant et finissant avec des différences nulles dans l'état interne contient au moins une itération dans laquelle au moins la moitié des octets de l'état interne sont actifs.*

La deuxième propriété revient à dire qu'il est impossible de trouver un chemin différentiel de poids faible pour GRINDAHL, nécessaire pour espérer une bonne probabilité de succès. La seule autre option pour un attaquant serait un chemin différentiel très court, possibilité exclue par la première propriété.

Ces deux propriétés forment le principal argument de sécurité de GRINDAHL quant à la recherche de collisions.

Une dernière observation intéressante pour l'attaquant est qu'en introduisant des différences dans l'état interne, après quelques itérations, nous arrivons très rapidement à une paire d'états contenant uniquement des différences tronquées non nulles (tous les octets sont actifs). De plus, cette situation de paire d'états *« partout différents »* est quasiment stable : la probabilité qu'une paire de colonnes, comptant uniquement des différences tronquées non nulles, donne après application de MixColumns à nouveau une paire de colonnes comportant uniquement des différences tronquées non nulles est approximativement égale à $P_A = (1 - 2^{-8})^4$. Ce qui nous donne une probabilité égale à $P_A^{12} \simeq 2^{-0,27}$ pour qu'un tel évènement se produise simultanément pour les douze colonnes de l'état interne. Cela semble indiquer qu'il est facile d'obtenir et de conserver une paire d'états partout différents. L'idée de départ de notre attaque est contre-intuitive pour un cryptanalyste puisque nous allons essayer de construire un chemin différentiel aboutissant à une collision en partant d'une paire d'états partout différents. L'avantage de cette approche est que la très grande probabilité P_A^{12} nous permet de disposer d'autant de paires d'états partout différents que l'on souhaite.

8.2.5 Trouver un chemin différentiel tronqué

Chercher un chemin différentiel tronqué commençant par une paire d'états internes partout différents est assez facile. Une méthode possible est de remonter le schéma en arrière de façon quasi exhaustive. En effet, dans GRINDAHL les différences tronquées se propagent en arrière à la même vitesse qu'en avant. Plus précisément, pour chercher une collision à la fin de l'itération k, nous essayons tous les masques possibles de différences tronquées pour les blocs de message insérés aux itérations k, $k-1$, etc., et toutes les transitions de différentielles tronquées possibles par application de la fonction MixColumns, jusqu'à atteindre à une paire d'états partout différents. Cet algorithme peut être grandement amélioré à l'aide d'une technique *« d'abandon rapide »* : on calcule une borne inférieure sur le coût du chemin que l'on est en train de construire (en prenant en compte le contrôle apporté par les octets actifs/passifs, voir section 8.3) et l'on arrête la branche de recherche actuelle si la complexité de l'attaque atteint ou dépasse 2^{128} opérations (borne idéale pour une fonction de hachage de 256 bits de sortie). On arrête aussi la recherche si l'on va trop loin en termes de nombre d'itérations : dans certains cas particuliers, la complexité globale de l'attaque peut rester constante même en augmentant indéfiniment le nombre d'itérations, et l'algorithme de recherche d'un chemin différentiel tronqué bouclerait alors indéfiniment.

Évidemment, toujours ajouter des différences tronquées dans tous les blocs de message insérés est le moyen le plus rapide d'arriver à une paire d'états partout différents. Cependant,

durant la recherche d'une paire de messages vérifiant le chemin différentiel tronqué, nous allons utiliser les octets de message insérés comme *octets de contrôle* pour attaquer indépendamment certaines parties du chemin et ainsi améliorer la probabilité de succès. Il peut donc être préférable de ne pas ajouter trop vite de différences et d'augmenter ainsi le nombre d'itérations du chemin différentiel tronqué. Cela aura pour effet d'augmenter le nombre de blocs de message insérés, et donc de fournir plus d'octets de contrôle. On peut voir cette technique comme une *dilution* du chemin. Par exemple, il est possible de trouver un chemin commençant par une paire d'états partout différents et menant à une collision en seulement 4 itérations, avec une probabilité de succès d'approximativement 2^{-312}. Néanmoins, un autre chemin d'une taille de 8 itérations et avec une probabilité de succès de 2^{-440} pourrait être meilleur dans notre cas. Effectivement, dans le deuxième cas, même si la probabilité de succès a été divisée par un facteur 2^{138}, nous avons en contrepartie inséré 8 paires de mots de message au lieu de 4 dans le premier cas. Ainsi, nous obtenons environ $2 \times 4 \times 4 \times 8 = 256$ degrés de liberté en plus (4 paires de mots de message de 4 octets chacun), ce qui peut être un gain plus important que la perte quant à la probabilité de succès. Il existe évidemment une limite : à partir d'un certain point, continuer d'ajouter des itérations n'améliore plus la situation, mais l'aggrave presque toujours.

8.3 Trouver une collision

Dans cette partie nous présentons une attaque par collision pour la version 256 bits de GRINDAHL, fondée sur l'analyse précédente.

8.3.1 Le chemin différentiel tronqué

Avant de décrire l'attaque permettant de trouver des collisions, la figure 8.2 donne le chemin différentiel tronqué généré par un programme implantant la technique expliquée en section 8.2.5. Ce chemin est l'un des candidats donnant la meilleure complexité d'attaque possible. Nous notons k la dernière itération de notre chemin différentiel tronqué, ce qui correspond à la dernière ligne dans la figure 8.2. Tout d'abord, on peut observer que toutes les transitions de différences tronquées de MixColumns sont valides (vérifient la propriété MDS). Ces transitions se produisent avec une certaine probabilité, et en multipliant ces probabilités entre-elles, on obtient la probabilité finale du chemin différentiel tronqué, approximativement égale à $2^{-55\times 8} = 2^{-440}$. Cette dernière semble très faible, mais nous insérons aussi beaucoup de blocs de message, ce qui va nous permettre d'attaquer plusieurs parties de façon indépendante.

Notre but est maintenant de trouver une paire de messages vérifiant le chemin différentiel tronqué attendu. Pour cela, nous ne chercherons pas à satisfaire les contraintes du chemin différentiel itération par itération, mais nous allons plutôt traiter les mots de message de 4 octets un par un. En d'autres termes, nous allons fixer les 4 octets d'une paire de mots de message de telle façon que les nouvelles transitions différentielles de MixColumns imposées soient celles attendues dans le chemin différentiel tronqué. Si c'est le cas, nous continuons à la prochaine paire de mots de message, jusqu'à obtenir la collision.

Dans le tableau 8.4 sont récapitulés les dépendances des transitions MixColumns en les mots de message insérés et utilisés comme octets de contrôle pour le chemin différentiel tronqué de la figure 8.2. Le coût de chaque transition est indiqué (voir section 8.2.2) ainsi que les octets de contrôle insérés à chaque itération (voir section 8.2.3). La deuxième colonne du tableau donne la position des colonnes de l'état interne étendu pour lesquelles nous forçons

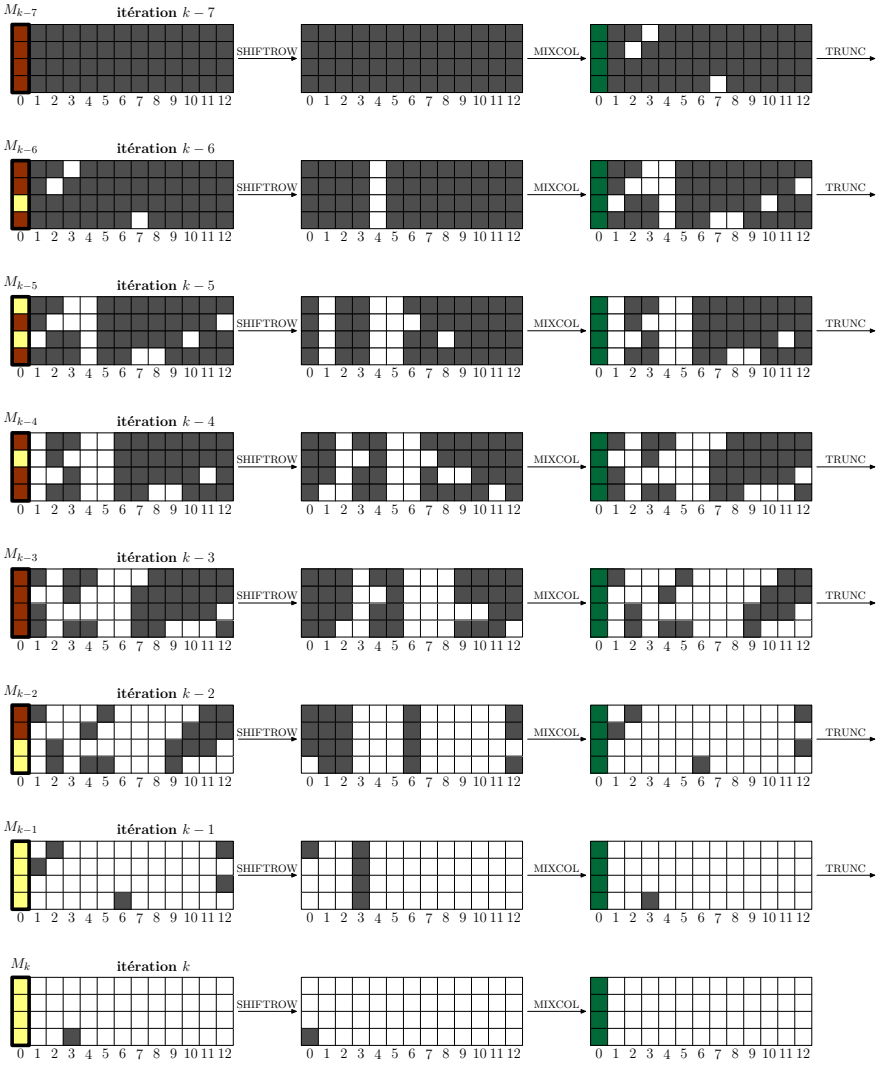

FIG. 8.2 – Chemin différentiel tronqué en 8 itérations et commençant par une paire d'états partout différents. L'état interne étendu est vu dans sa représentation matricielle, chaque cellule représentant un octet. Les cellules grisées correspondent à une différence tronquée non nulle pour l'octet concerné, les blanches correspondent à une différence tronquée nulle. Chaque ligne d'états internes étendus représente une itération. Le schéma simplifié est décrit : seules les transformations ayant une influence sur les différentielles tronquées sont considérées, les fonctions AddConstant et SubBytes étant donc omises. La première colonne donne l'état interne étendu juste après sa mise à jour à l'aide d'un mot de message de 4 octets, et la deuxième colonne donne le même état après application de la transformation ShiftRows. Enfin, la troisième colonne correspond à l'état interne étendu juste après application de la fonction MixColumns. Chaque première colonne de 4 octets de la première colonne d'états étendus représente les mots de message insérés à chaque itération, qui seront utilisés comme octets de contrôle. Il faut noter que dans chaque ligne, la première colonne de 4 octets de la dernière colonne d'états étendus (après application de MixColumns) peut prendre n'importe quelle valeur de masque de différences tronquées puisque ces octets seront écrasés par le prochain mot de message inséré.

une transition différentielle d'une valeur de différence non-nulle vers une valeur nulle durant la transformation MixColumns, et la première colonne du tableau indique à quelle itération cela intervient. Pour chaque transition, la troisième colonne du tableau correspond au coût en termes de nombre d'octets de contrôle (pour un coût c, la transition a une probabilité égale à $2^{-c\times 8}$). Enfin, chacune des sept colonnes restantes du tableau représente une paire de mots de message, qui sera utilisée comme octets de contrôle : les lettres a ou A, b ou B, c ou C et d ou D représentent respectivement le premier, deuxième, troisième et quatrième octet des 4 octets de message insérés. Une lettre en majuscule signifie que l'on a deux octets de contrôle (à cause d'une différence d'entrée tronquée non nulle, il est possible de faire varier les deux octets de la paire) et une lettre en minuscule signifie que l'on n'a qu'un seul octet de contrôle (il n'y a aucune différence pour cet octet de message). Dans le corps du tableau, un tiret ou une croix indique si la transition MixColumns désignée par la ligne correspondante est affectée par l'octet de contrôle désigné par la colonne correspondante. Nous avons divisé ces dépendances en deux groupes pour plus de simplicité : les croix représentent pour chaque transition MixColumns les dépendances du dernier mot de message impliqué (les tirets indiquant les autres dépendances). Enfin, la dernière ligne calcule le coût pour l'attaquant en termes de nombre d'octets pour chaque mot de message fixé (la somme de ces coûts est égale à la complexité finale de l'attaque). Le calcul de ce coût est expliqué en section 8.3.2.

En lisant le tableau, nous obtenons finalement qu'il nous faille tester $2^{14\times 8} = 2^{112}$ paires d'états internes partout différents pour avoir une bonne probabilité d'aboutir à une collision. L'attaque est décrite plus en détail dans la section suivante.

8.3.2 L'attaque par collision

Première étape :

À partir de la valeur initiale prédéfinie pour l'état interne, calculer quelques itérations avec beaucoup de différences tronquées dans les blocs de message insérés pour rapidement arriver à une paire d'états internes partout différents, notée A. Cette étape peut être omise lors du calcul total de la complexité, son coût étant largement négligeable.

Deuxième étape :

À partir de la paire d'états internes partout différents A, engendrer $2^{14\times 8} = 2^{112}$ paires d'états internes partout différents $A_1, \ldots, A_{2^{112}}$. Cette étape nécessite $2^{112} \times 2^{0,27} = 2^{112,27}$ itérations.

Troisième étape :

Fixer les octets de contrôle itération par itération : pour les blocs de message ajoutés au début des itérations $k-8$, $k-7$ et $k-6$ de notre chemin différentiel tronqué du tableau 8.4, nous avons plus d'octets de contrôle que nécessaire. En effet, nous avons 8 (respectivement 8 et 7) octets disponibles pour les messages insérés à l'itération $k-8$ (respectivement $k-7$ et $k-6$), alors que seulement 2 (respectivement 7 et 7) octets de degrés de liberté sont requis, comme l'indique le calcul dans le tableau 8.4 de la somme des coûts des lignes où une croix apparaît dans la colonne considérée. De façon plus générale, pour chaque paire de mots de message insérés (M_{k-i}, M'_{k-i}), ces octets seront utilisés pour ajuster le comportement des transitions MixColumns où les croix apparaissent à la colonne M_{k-i} dans le tableau 8.4 : puisque les croix représentent le dernier mot de message impliqué dans cette transition, les dépendances qui le

it.	col.	coût	blocs de message insérés																									
			$k-8$				$k-7$				$k-6$				$k-5$				$k-4$				$k-3$				$k-2$			
			A	B	C	D	A	B	C	D	A	B	c	D	a	B	c	D	A	b	C	D	A	B	C	D	A	B	c	d
k-7	2	1	–					×																						
	3	1	×	×																										
	7	1				×																								
k-6	1	1		–		–			–	–	×																			
	2	1	–		–	–	–					×																		
	3	2	–	–	–	–	×	×																						
	7	1	–	–	–					×																				
	8	1	–	–	–	–			×																					
	10	1		–	–									×																
	12	1	–		–	–		×		×																				
k-5	2	1	–	–	–	–	–		–	–	–					×														
	3	1	–	–	–	–	–	–	–	–	×	×																		
	8	1	–	–	–	–	–	–	–	–			×																	
	9	1	–	–	–	–	×	×	×	×																				
	11	1	–	–	–	–			–	–	×		×	×																
k-4	1	1	–	–	–	–	–	–	–	–		–		–			–	–	×											
	3	1	–	–	–	–	–	–	–	–	–	–	–	–	×	×														
	4	2	–	–	–	–	–	–	–	–	–	–		–		–					×									
	7	1	–	–	–	–	–	–	–	–	–	–	–					×												
	9	1	–	–	–	–	–	–	–	–	×	×	×	×																
	10	1	–	–	–	–	–	–	–	–		–	–									×								
	11	1	–	–	–	–	–	–	–	–			–	–	×		×	×												
	12	1	–	–	–	–	–	–	–	–	–		–	–		×		×												
k-3	1	3	–	–	–	–	–	–	–	–	–	–	–	–		–		–			–	–	×							
	2	2	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–	–					×						
	4	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–		–					×					
	5	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	×		×									
	9	2	–	–	–	–	–	–	–	–	–	–	–	–	×	×	×	×												
	10	2	–	–	–	–	–	–	–	–	–	–	–	–		–	–									×				
	11	1	–	–	–	–	–	–	–	–	–	–	–	–			–	–	×		×	×								
	12	2	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–		×		×								
k-2	1	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–		–			–	–	×			
	2	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–	–					×		
	6	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–			×	×					
	12	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–		×		×				
k-1	3	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	×	×		
Contraintes			**2**				**7**				**7**				**7**				**9**				**14**				**9**			
Octets de contôle			**8**				**8**				**7**				**6**				**7**				**8**				**4**			
COUT			**0**				**0**				**0**				**1**				**2**				**6**				**5**			

TAB. 8.4 – Dépendances des blocs de message utilisés comme octets de contrôle et insérés lors du chemin différentiel tronqué de la figure 8.2 pour une collision à la fin de l'itération k.

précèdent (représentées par des tirets) sont déjà fixées à ce moment de l'attaque. Pour chaque mot de message, le coût total est égal à la somme des coûts de toutes les transitions MixColumns impliquées, diminuée du nombre d'octets de contrôle disponibles et utilisables dans M_{k-i}. Ainsi, à ce point de l'attaque, nous maintenons toujours 2^{112} paires d'états internes vérifiant le chemin différentiel tronqué. Pour les mots de message insérés durant l'itération $k-5$, nous avons 6 octets de contrôle pour 7 octets de conditions sur les transitions, nous gardons

ainsi une paire d'états internes sur 2^8 et nous avançons au $(k-4)$-ième mot de message avec 2^{104} paires d'états internes valides. Nous continuons l'attaque de la même manière pour les trois derniers mots de message $k-4$, $k-3$ et $k-2$, qui fournissent respectivement 7, 8 et 4 octets de contrôle [‡] pour 9, 14 et 9 octets de conditions respectivement. Nous avons donc une bonne probabilité de trouver une paire d'états internes (et sa paire de messages correspondante) vérifiant entièrement le chemin différentiel tronqué en commençant avec $2^{14\times 8} = 2^{112}$ éléments différents.

Quatrième étape :

Ajouter un $(k+1)$-ième bloc de message avec des différences tronquées nulles pour forcer l'écrasement des différences non nulles dans la première colonne de l'état interne étendu après la dernière itération k, le tout sans en introduire de nouvelles. Ceci permet d'éviter d'entrer directement dans les tours à blanc (pour lesquels la troncature est omise) juste après le chemin différentiel, auquel cas la première colonne ne serait pas écrasée et les différences tronquées non nulles toujours présentes.

8.3.3 Discussion de l'attaque

Dans un souci de clarté, nous explicitons ici plus en détail comment manipuler les octets de contrôle en étudiant un exemple : la manière dont l'attaquant peut fixer la paire de messages introduite à l'itération $k-5$ (septième colonne du tableau 8.4). Les précédents mots de message ayant déjà été fixés durant le début de l'attaque, nous devons juste réaliser les contraintes indiquées par une croix de la colonne considérée dans le tableau 8.4. Certaines transitions différentielles de MixColumns doivent se comporter comme attendu dans le chemin différentiel tronqué, et cela a un coût (voir tableau 8.2). Par exemple, dans la deuxième colonne de la $(k-5)$-ième itération, nous attendons une transition de 4 différences tronquées à 3 différences tronquées et cela se produira avec probabilité 2^{-8}, donc avec un coût d'un octet. Cependant, pour faire se produire cette transition, nous pouvons utiliser le mot de message inséré à l'itération $k-5$ (et plus exactement son deuxième octet) pour ainsi varier l'instance de transition, et ce jusqu'à arriver à celle attendue (il y a plusieurs façons de procéder pour cette étape, et ceci est étudié dans le prochain paragraphe). Il y a en fait une bonne probabilité de trouver 2^8 paires d'octets valides, car nous avons deux octets de contrôle pour un octet de condition. Nous continuons le même processus pour la transition de la septième colonne de l'itération $k-4$ avec le quatrième octet du mot de message : encore une fois, nous disposons de deux octets de contrôle pour un octet de conditions et donc de 2^8 paires d'octets valides. On identifie alors les éléments du produit des deux ensembles de taille 2^8 venant d'être calculés pour lesquels la transition MixColumns de la douzième colonne de l'itération $k-4$ est vérifiée, ce qui nous coûte un octet de conditions et nous laisse donc 2^8 paires valides pour le deuxième et le quatrième octet du message inséré à l'itération $k-5$. On fixe alors le premier octet du message pour réaliser la transition de la troisième colonne de l'itération $k-4$: puisque nous consommons un octet de contrôle pour un octet de conditions, nous maintenons toujours 2^8 paires valides. Enfin, avec le dernier octet du message à fixer (le troisième), nous cherchons à réaliser une bonne transition pour la neuvième colonne de l'itération $k-3$: nous ne disposons que d'un octet de contrôle pour deux octets de conditions, mais comme nous avions maintenu 2^8 paires valides, cela nous

[‡] pour le cas du mot de message $k-2$, nous avons seulement 4 octets de contrôle et non 6 comme pourrait le suggérer le tableau 8.4 : puisque c et d ne sont impliqués dans aucune transition MixColumns du chemin différentiel tronqué, ils ne peuvent être considérés comme octets de contrôle.

fournit un octet de contrôle en plus. Pour résumer, à la fin de cette étape nous avons une bonne probabilité de trouver une paire du mot de message arrivant à l'itération $k-5$ vérifiant les transitions citées. Cependant, nous n'avons pas encore pris en compte la transition de la onzième colonne de la $k-4$-ième itération, ce qui nous coûte un octet de conditions. Finalement, pour cet exemple, fixer le mot de message nous coûtera 2^8 essais, car nous avions un total de six octets de contrôle utilisables pour sept octets de conditions. En répétant ce raisonnement pour tous les mots de message insérés à chaque itération du chemin différentiel tronqué, nous aboutissons à la complexité totale de 2^{112} pour trouver une collision.

On pourrait penser que même s'il faut essayer 2^{112} paires d'états partout différents, l'opération de base qui consiste à faire varier les octets de contrôle peut être coûteuse. En effet, en prenant l'exemple précédent, certaines étapes demandent de tester 2^8 ou 2^{16} valeurs de mots de message, chacune nécessitant un calcul de la fonction SubBytes, ou d'une ou deux itérations (cela dépend de la colonne de l'état interne étendu où la transition a lieu). Même si cela représente toujours une attaque valide, la complexité en est légèrement augmentée. Cet argument est vrai si l'attaquant utilise une méthode de recherche naïve. Cependant, des précalculs peu coûteux permettent de réduire le coût de recherche en procédant uniquement à de rapides consultations de tables en mémoire. Par exemple, avec un précalcul de complexité 2^{32} en temps et en mémoire, il est possible d'engendrer toutes les informations requises pour très rapidement effectuer la recherche des bons éléments durant la troisième étape de l'attaque par collision.

On peut aussi se demander pourquoi la complexité des transitions de 4 différences tronquées vers 4 différences tronquées n'est pas comptabilisée. De telles transitions se produisent toujours avec une très grande probabilité $P_A = (1-2^{-8})^4 \simeq 2^{-0,02}$ et c'est pourquoi elles ont un effet négligeable sur la complexité totale de l'attaque. De plus, le mode fonction de compression utilise 40 itérations pour un appel à la fonction de compression, ce qui montre que notre attaque est même plus rapide que 2^{112} appels à la fonction de hachage.

Nous avons vérifié qu'une attaque de ce type de complexité au plus 2^{120} appels à la fonction de hachage existe pour toutes les constantes de rotation apportant la meilleure diffusion possible (au sens du critère retenu par les concepteurs pour choisir ces constantes), ce qui semble indiquer que l'état interne de GRINDAHL n'est pas assez grand.

La version 512 bits de GRINDAHL peut probablement être attaquée de la même manière. Mais la recherche de chemins différentiels est alors beaucoup trop coûteuse, car l'état interne devient très grand. Cependant, cette recherche nécessite une capacité de calcul bien moindre que celle requise par la borne de l'attaque générique du paradoxe des anniversaires, même si elle est hors de portée des ordinateurs actuels.

8.4 Améliorations et autres attaques

L'attaque que nous venons de présenter peut être améliorée si l'on dispose de beaucoup de mémoire, comme le montre Khovratovich [Kho08]. Grâce à une utilisation fine des messages testés, il est possible de créer des structures en entrée qui vont multiplier naturellement les instances. Ceci est possible, du fait que notre attaque utilise des différentielles tronquées. Ainsi, presque toute paire issue d'un groupe de n messages pourra vérifier un chemin composé de différences tronquées, alors que nous n'avions considéré que $n/2$ paires parmi les n messages. Cette amélioration aboutit à une attaque en collision contre la version 256 bits de GRINDAHL avec une puissance de calcul de 2^{100} appels à la fonction et de complexité 2^{84} en mémoire.

La méthode d'attaque en collision contre GRINDAHL présentée dans ce chapitre peut être légèrement modifiée pour cryptanalyser la famille de fonctions de hachage RADIOGATÚN, fondée sur la théorie des fonctions éponges [BDP08] et récemment proposée par Bertoni *et al.* [BDP06]. Nous expliquons dans [Pey08] qu'il semble possible de trouver des collisions en un temps inférieur à celui de l'attaque générique du paradoxe des anniversaires, en utilisant des différences très spécifiques et une méthode de recherche de chemins différentiels assez semblable à celle explicitée dans ce chapitre.

Enfin, GRINDAHL et RADIOGATÚN possédant une structure similaire, nous pouvons introduire le concept de *fonctions éponges étendues*, incluant ces deux schémas. En collaboration avec Stefan Lucks et Michael Gorski, nous avons montré dans [Pey08, GLP08] qu'une technique de cryptanalyse issue des algorithmes de chiffrement par blocs, les *attaques glissantes* [BW99], peut s'appliquer aux fonctions de hachage utilisant des fonctions éponges comme composant principal. Plus précisément, nous montrons qu'il est possible d'attaquer certaines de ces fonctions lorsqu'on les utilise pour calculer un MAC, l'une des principales applications des fonctions de hachage. Ces techniques s'appliquent aux deux versions de GRINDAHL ainsi qu'à une version très légèrement modifiée de RADIOGATÚN.

CHAPITRE 9
Cryptanalyse de FORK-256

Sommaire

FORK-256 est une nouvelle famille de fonctions de hachage de 256 bits de sortie conçue par Hong *et al.* [HCS05, HCS06]. Une de ses principales caractéristiques (et aussi l'origine de son nom) est que, comme la famille de fonctions de hachage RIPEMD, elle utilise plusieurs branches parallèles à l'intérieur de sa fonction de compression. La version proposée, plus rapide que SHA-256, requiert quatre branches parallèles. Une première cryptanalyse d'une version réduite à deux branches fut rapidement publiée par Mendel *et al.* [MLP07], puis Matusiewicz *et al.* [MCP06] présentèrent d'autres vulnérabilités. Suite à un travail indépendant, fusionné avec les résultats de Matusiewicz, Contini et Pieprzyk, nous avons publié un article à la conférence FSE 2007 présentant la première attaque par collision pour la fonction entière [MPB07]. Par la suite, une amélioration de notre attaque fut proposée par Contini *et al.* [CMP07]. Enfin, l'algorithme a été mis à jour par Hong *et al.* [HCS07] pour corriger ces défauts, mais cette modification a immédiatement été suivie par la publication d'une nouvelle vulnérabilité [Saa07a].

9.1 Description de FORK-256

FORK-256 est une famille de fonctions de hachage fondée sur une construction relativement classique : elle utilise le schéma de Merkle-Damgård avec une fonction de compression h

qui fait correspondre à 256 bits de variable de chaînage H_i et à 512 bits de message M, la nouvelle variable de chaînage H_{i+1} de 256 bits. Nous donnons ici une description rapide de cet algorithme ; pour obtenir les caractéristiques complètes, on pourra se reporter à [HCS05, HCS06].

À chaque itération i, la fonction de compression maintient un ensemble de quatre branches parallèles $\{\text{BRANCHE}_j\}$ avec $j \in 1, 2, 3, 4$, chacune d'elles utilisant un ordonnancement différent des 16 mots de 32 bits de message $M = \{m_0, \ldots, m_{15}\}$ défini par une permutation σ_j. La variable de chaînage, composée de 8 mots de 32 bits chacun $H_i = (\overline{A_0}, \overline{B_0}, \overline{C_0}, \overline{D_0}, \overline{E_0}, \overline{F_0}, \overline{G_0}, \overline{H_0})$ est insérée dans les 4 branches à la fois. Après avoir calculé la sortie de chacune des branches parallèles $h_j = \text{BRANCHE}_j(H_i, M)$, on en déduit la nouvelle variable de chaînage par :

$$H_{i+1} = H_i + [(h_1 + h_2) \oplus (h_3 + h_4)] \quad ,$$

où les opérations $+$ et $\oplus$ sont effectuées mot de 32 bits par mot de 32 bits. Ceci est expliqué dans la figure 9.1.

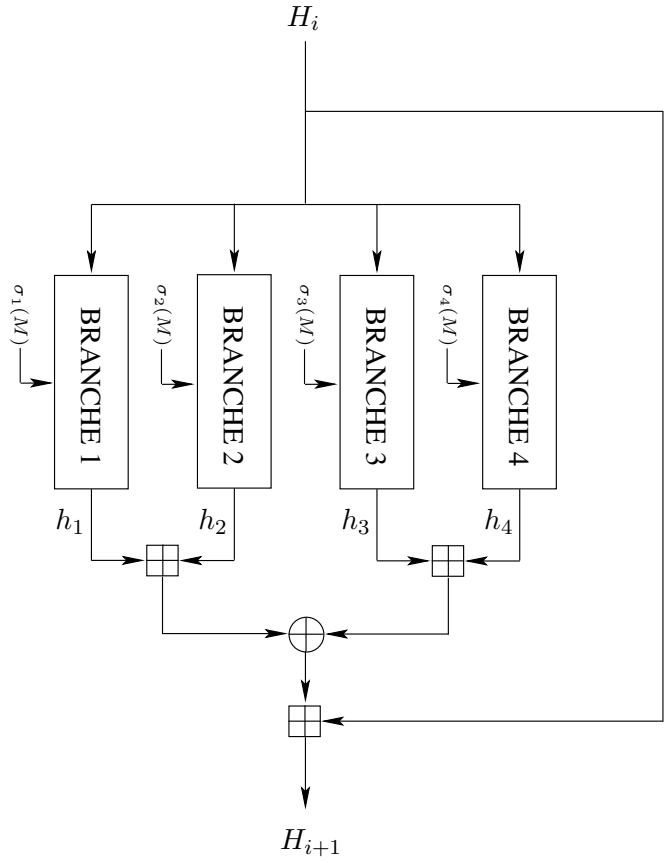

FIG. 9.1 – Squelette de la fonction de compression de FORK-256.

Chaque branche BRANCHE_j est composée de 8 étapes. À chaque tour $k = 1, \ldots, 8$, la fonction de branche met à jour sa propre copie d'un état interne de huit mots de 32 bits, tous initialisés par la variable de chaînage H_i. $R_{j,k}$ représente la valeur du registre $R \in \{A, \ldots, H\}$ dans la j-ième branche après l'étape k. Les mots $A_{j,0}$, …, $H_{j,0}$ sont initialisés avec les valeurs correspondantes des 8 mots de la variable de chaînage entrante $\overline{A_0}$, …, $\overline{H_0}$. Une étape k applique la transformation décrite dans la figure 9.2, à savoir pour $1 \leqslant j \leqslant 4$:

$$
\begin{aligned}
X_f &= f(A_{j,k-1} + m_{\sigma_j(2k-2)}) \\
X_g &= g(A_{j,k-1} + m_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)}) \\
Y_g &= g(E_{j,k-1} + m_{\sigma_j(2k-1)}) \\
Y_f &= f(E_{j,k-1} + m_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)}) \\
A_{j,k} &= (H_{j,k-1} + Y_g^{\lll 21}) \oplus Y_f^{\lll 17} \\
B_{j,k} &= A_{j,k-1} + m_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)} \\
C_{j,k} &= (B_{j,k-1} + X_f) \oplus X_g \\
D_{j,k} &= (C_{j,k-1} + X_f^{\lll 5}) \oplus X_g^{\lll 9} \\
E_{j,k} &= (D_{j,k-1} + X_f^{\lll 17}) \oplus X_g^{\lll 21} \\
F_{j,k} &= E_{j,k-1} + m_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)} \\
G_{j,k} &= (F_{j,k-1} + Y_g) \oplus Y_f \\
H_{j,k} &= (G_{j,k-1} + Y_g^{\lll 9}) \oplus Y_f^{\lll 5}
\end{aligned}
$$

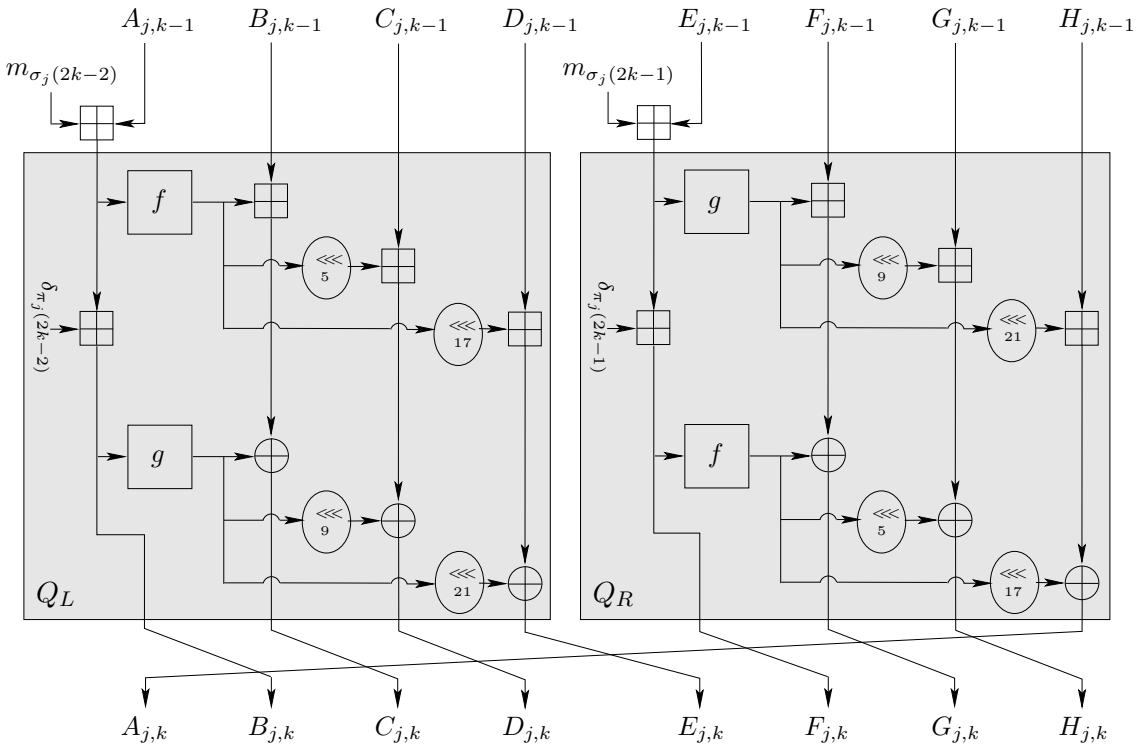

FIG. 9.2 – Transformation lors du tour k pour la branche j de FORK-256. Les Q-structures sont encadrées.

Les fonctions f et g sont respectivement définies par $f(x) = x + \left(x^{\lll 7} \oplus x^{\lll 22}\right)$ et $g(x) = x \oplus \left(x^{\lll 13} + x^{\lll 27}\right)$. Enfin, les constantes $\delta_0, \ldots, \delta_{15}$ sont données dans le tableau 9.1, et les permutations σ_j et π_j sont définies dans le tableau 9.2.

δ_0	δ_1	δ_2	δ_3
0x428a2f98	Ox71374491	Oxb5c0fbcf	Oxe9b5dba5

δ_4	δ_5	δ_6	δ_7
0x3956c25b	Ox59f111f1	Ox923f82a4	Oxab1c5ed5

δ_8	δ_9	δ_{10}	δ_{11}
0xd807aa98	Ox12835b01	Ox243185be	Ox550c7dc3

δ_{12}	δ_{13}	δ_{14}	δ_{15}
0x72be5d74	Ox80deb1fe	Ox9bdc06a7	Oxc19bf174

TAB. 9.1 – Constantes $\delta_0, \ldots, \delta_{15}$ utilisées pour FORK-256.

j	permutation de messages $\sigma_j(\cdot)$																permutation de constantes $\pi_j(\cdot)$															
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	14	15	11	9	8	10	3	4	2	13	0	5	6	7	12	1	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3	7	6	10	14	13	2	9	12	11	4	15	8	5	0	1	3	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
4	5	12	1	8	15	0	13	11	3	10	9	2	7	14	4	6	14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1

TAB. 9.2 – Permutations des mots de message et des constantes utilisées dans chaque branche j de FORK-256.

9.2 Observations préliminaires sur FORK-256

Comme nous l'avons expliqué dans la section précédente, FORK-256 maintient quatre branches parallèles opérant sur le même état initial et utilisant les mêmes blocs de message, mais dans un ordre différent. Cela semble être l'une des principales forces de FORK-256 et de fait, les premières cryptanalyses publiées étaient limitées à deux des quatre branches [MLP07]. En d'autres termes, la difficulté pour cryptanalyser FORK-256 provient essentiellement du fait que les mêmes blocs de message sont les entrées des quatre branches, mais dans un ordre permuté. Ainsi, alors qu'attaquer une ou deux branches semble relativement facile, l'effet des différences est difficile à éliminer dans les branches restantes. On peut cependant identifier quelques caractéristiques différentielles intéressantes de FORK-256.

k	$\Delta A_{j,k}$	$\Delta B_{j,k}$	$\Delta C_{j,k}$	$\Delta D_{j,k}$	$\Delta E_{j,k}$	$\Delta F_{j,k}$	$\Delta G_{j,k}$	$\Delta H_{j,k}$	Δm_L	Δm_R	Prob.
0	d	d	d	d	d	d	d	d	$-d$	$-d$	
1	d	0	d	d	d	0	d	d	$-d$	$-d$	P_d^6
2	d	0	0	d	d	0	0	d	$-d$	$-d$	P_d^4
3	d	0	0	0	d	0	0	0	$-d$	$-d$	P_d^2
4	0	0	0	0	0	0	0	0			1

TAB. 9.3 – Une caractéristique différentielle pour FORK-256 sur 4 tours et aboutissant à une collision locale. Le tableau donne le chemin différentiel dans une branche ainsi que sa probabilité de succès à chaque tour. Enfin, m_L (respectivement m_R) représente le mot de message ajouté à droite (respectivement à gauche) de chaque tour dans une branche.

La première, identifiée indépendamment par Muller [Mul06p] et Mendel *et al.* [MLP07],

surmonte le problème des quatre branches en appliquant la même différence modulaire $-d$ à chaque bloc de message. Si après que la quatrième étape est terminée, l'état interne présente la différence d sur chacun de ses 8 mots de 32 bits, nous obtenons une collision sur chacune des branches à la fin des huit tours. Ce comportement, résumé dans le tableau 9.3, rend complètement inefficace l'utilisation de quatre branches avec une permutation sur l'ordre des messages pour contrer la cryptanalyse différentielle : la même différence est appliquée à chaque mot de message et le même motif survient dans les quatre branches simultanément. La probabilité P_d représente la probabilité que la différence d se propage sans modification durant un tour, compte tenu de ce que la différence modulaire doit passer à travers un $\oplus$. En effet, les différences modulaires ne se propagent pas sans modification lorsqu'un $\oplus$ est traversé (de la même façon que les différences binaires ne se propagent pas sans modification à travers un $+$). Cette probabilité peut être calculée de manière exacte pour toute différence d donnée. Une étude plus générale de ce problème a été réalisée par Lipmaa, Wallén et Dumas [LWD04] et nous donnons ici une version plus faible de leurs résultats, adaptée à nos besoins :

Propriété 3 *Soit d un mot de 32 bits, la probabilité*

$$P_d = Pr_{x,y}\left[\left((x+d)\oplus y\right) = \left(x\oplus y\right) + d\right]$$

où les éléments x et y sont deux mots de 32 bits, peut être exprimée comme le produit matriciel suivant :

$$P_d = L \times M_{d_{31}} \times M_{d_{30}} \times \cdots \times M_{d_0} \times C,$$

où d_i représente le i-ième bit de d et L, C, M_0, et M_1 sont définis par :

$$M_0 = \frac{1}{4}\begin{pmatrix} 4 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad M_1 = \frac{1}{4}\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad {}^{T}C = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Il faut noter que la différence d se propage sans modification durant un tour lorsqu'elle est située dans le registre A ou le registre E de l'état interne, mais avec une probabilité P_d dans le cas contraire. La probabilité finale du chemin différentiel du tableau 9.3 est donc égale à P_d^{12} pour chaque branche.

Une autre méthode pour éviter le problème des différents ordonnancements des mots de message sur les quatre branches consiste à appliquer une différence sur la variable de chaînage ou sur la variable de chaînage et le message plutôt que sur le message seul. On est donc amené à rechercher pour la fonction de compression h une pseudo-collision $\mathrm{h}(H_i, M) = \mathrm{h}(H'_i, M')$ ou une pseudo-presque collision $\mathrm{h}(H_i, M) \simeq \mathrm{h}(H'_i, M')$ où $(H'_i, M') \neq (H_i, M)$. Dans le cas de FORK-256, les différences dans chaque mot de l'état interne ne se diffusent pas de manière identique, comme on peut le remarquer sur la figure 9.2. Plus précisément, seules les différences présentes dans les registres A et E vont se propager aux autres registres à la prochaine étape. Les autres différences (dans les registres B, C, D, F, G et H) sont simplement décalées

k	$\Delta A_{j,k}$	$\Delta B_{j,k}$	$\Delta C_{j,k}$	$\Delta D_{j,k}$	$\Delta E_{j,k}$	$\Delta F_{j,k}$	$\Delta G_{j,k}$	$\Delta H_{j,k}$	Δm_L	Δm_R	Prob.
0	0	d_0	0	0	0	0	0	0	0	0	
1	0	0	d_1	0	0	0	0	0	0	0	
2	0	0	0	d_2	0	0	0	0	0	0	
3	0	0	0	0	d_3	0	0	0	0	0	
4	0	0	0	0	0	d_4	0	0	0	0	P'
5	0	0	0	0	0	0	d_5	0	0	0	
6	0	0	0	0	0	0	0	d_6	0	0	
7	d_7	0	0	0	0	0	0	0			

TAB. 9.4 – Un chemin différentiel de sept étapes aboutissant à une pseudo-presque collision pour FORK-256. Le tableau donne le chemin différentiel dans une branche ainsi que sa probabilité de succès à chaque tour. Enfin, m_L (respectivement m_R) représente le mot de message ajouté à droite (respectivement à gauche) de chaque tour dans une branche.

d'un registre vers la droite (et potentiellement modifiées). Ainsi, en appliquant une différence au deuxième mot de la variable de chaînage (correspondant au registre B), cette différence va se propager vers la droite sans se diffuser aux autres registres durant trois tours. Pendant le quatrième tour, la différence d_3 va très probablement se diffuser à partir du registre E dans trois des autres registres internes (F, G, et H), et ce, dans les quatre branches. Cependant, nous montrons dans la prochaine section qu'il est possible d'éviter la diffusion d'une différence lorsqu'elle est présente dans les registres A ou E, ce qui se produirait, si la diffusion d'une différence vers plusieurs mots de sortie pouvait être modélisée par le tirage au sort de différences de sortie indépendantes, avec probabilité $P' \simeq 2^{-3\times 32} = 2^{-96}$. On peut constater que le raisonnement pour cette caractéristique différentielle sera identique si l'on applique la différence d_0 sur le registre F au lieu du registre B. De plus, la même remarque que pour le premier motif est toujours valable : si l'on souhaite qu'une différence se propage sans être modifiée, cela ne se produira qu'avec probabilité P_d (sauf pour le quatrième tour où la probabilité est égale à 1 puisque la différence est alors située sur le registre A ou E).

9.3 Les microcollisions

La transformation exécutée à chaque étape dans chaque branche, décrite en section 9.1, peut être découpée en trois parties : ajout des mots de message, deux structures parallèles Q_L et Q_R, et une permutation finale des registres internes. Ces parties sont identifiées dans la figure 9.2. Q_L et Q_R sont les principaux éléments apportant confusion et diffusion dans la fonction de compression.

Dans cette section, nous donnons une méthode pour trouver, quelle que soit la branche considérée, des chemins différentiels de la forme $(\Delta A, 0, 0, 0) \rightarrow (\Delta A, 0, 0, 0)$ à travers une structure Q_L et montrons que cela s'applique de la même manière au cas d'une structure Q_R. L'idée générale est de chercher des valeurs du registre A et les valeurs appropriées en entrée des registres B, C, D pour que la différence en sortie de B, C, D soit égale à zéro, malgré des différences non nulles en sortie des fonctions f et g. De telles situations se produisent si l'on obtient trois *microcollisions* simultanées, i.e. si dans chacun des registres B, C et D, la différence en sortie de la fonction g annule exactement la différence en sortie de la fonction f.

Soit $y = f(x)$, $y' = f(x')$ et $z = g(x + \delta)$, $z' = g(x' + \delta)$. Nous obtenons une microcollision

pour le registre B si l'équation

$$(y + B) \oplus z = (y' + B) \oplus z' \tag{9.1}$$

est vérifiée pour des valeurs de y, y', z, z' données et une constante B. Notre but est de trouver l'ensemble des constantes B pour lesquelles l'équation (9.1) est satisfaite.

Nous réutilisons les trois différentes manières de représenter une différence entre deux nombres x et x' de w bits chacun, déjà introduites dans le chapitre 5. Les relations entre ces représentations seront utiles pour la suite. On peut observer que si une différence binaire signée vaut $(r_0, r_1, \ldots, r_w)$ où $r_i \in \{-1, 0, 1\}$, alors la différence binaire correspondante est égale à $(|r_0|, |r_1|, \ldots, |r_w|)$: pour chaque bit, chaque différence positive ou négative est considérée comme une différence, et une différence nulle reste une différence nulle. La relation entre une différence additive et une différence binaire signée est plus complexe : la différence additive $\partial(x, x')$ correspondant à une différence binaire signée $\Delta^{\pm}(x, x') = (r_0, \ldots, r_w)$, avec $r_i \in \{-1, 0, 1\}$, vaut $\partial(x, x') = \sum_{i=0}^{w} 2^i \cdot r_i$. Il faut noter que cette correspondance n'est pas univoque. Par exemple, en considérant des mots de 4 bits, nous avons $\Delta^{\pm}(11, 2) = (1, 0, 0, 1)$, $\Delta^{\pm}(14, 5) = (1, 0, 1, -1)$, et $\Delta^{\pm}(12, 3) = (1, 1, -1, -1)$. Toutes ces différences binaires signées correspondent à la même différence additive $\partial(x, x') = 9$. Il est aussi possible de passer d'une paire de valeurs à une autre en ajoutant une constante appropriée, par exemple $(12, 3)$ = $(11 + 1, 2 + 1)$. Cette addition préserve la différence additive, mais peut modifier la différence binaire signée.

En réécrivant l'équation (9.1) sous la forme $(y + B) \oplus (y' + B) = z \oplus z'$, on peut facilement voir que la différence binaire signée $\Delta^{\pm}(y + B, y' + B)$ ne peut avoir un chiffre non nul qu'aux positions où la différence binaire $\Delta^{\oplus}(z, z')$ vaut 1. Cela permet de diminuer l'ensemble des représentations binaires signées possibles qui vérifient la différence binaire à $2^{h_w(\Delta^{\oplus}(z,z'))}$ éléments, où $h_w(x)$ représente le poids de Hamming d'une différence binaire signée x de w bits. Mais puisqu'à une différence binaire signée correspond une unique différence additive, il y a aussi seulement $2^{h_w(\Delta^{\oplus}(z,z'))}$ différences additives $\Delta^{\pm}(y + B, y' + B)$ (et donc seulement $2^{h_w(\Delta^{\oplus}(z,z'))}$ différences additives $\partial(y, y')$ puisqu'une différence additive est préservée lorsque l'on ajoute une constante B) qui vérifient la différence binaire $\Delta^{\oplus}(z, z')$ donnée.

Ainsi, pour vérifier si une différence additive particulière $\partial(y, y') = y - y'$ peut potentiellement satisfaire la différence binaire désirée, il faut résoudre le problème suivant : étant donné $\partial(y, y') = y - y'$, $-2^{32} < \partial(y, y') < 2^{32}$ et un ensemble de positions $I = \{k_0, k_1, \ldots, k_m\} \subset \{0, \ldots, 31\}$ déterminées par les bits non nuls de $\Delta^{\oplus}(z, z')$, décider s'il est possible de trouver une représentation binaire signée $r = (r_0, \ldots, r_{31})$ correspondant à $\partial(y, y')$ et telle que

$$\partial(y, y') = \sum_{i=0}^{m} 2^{k_i} \cdot r_{k_i} \quad \text{où } r_{k_i} \in \{-1, 1\} \ . \tag{9.2}$$

En remplaçant r_{k_i} par $2 \cdot t_i - 1$ où $t_i \in \{0, 1\}$, cette équation peut être mise sous la forme équivalente

$$\partial(y, y') + \sum_{i=0}^{m} 2^{k_i} = 2^{k_0+1} t_0 + 2^{k_1+1} t_1 + \cdots + 2^{k_m+1} t_m \ , \tag{9.3}$$

Décider s'il existe des nombres t_i qui satisfont l'équation (9.3) est une instance du problème appelé *problème du sac à dos* [Kar72]. Il est de plus connu que lorsque l'instance de ce problème est *supercroissante* (ce qui est le cas ici puisque tous les poids sont des puissances de 2, ce qui est notre cas), il est facile de le résoudre. Cela nous donne ainsi une condition nécessaire pour

une microcollision qui est facile à calculer : la différence $\partial(y, y') = y - y'$ doit pouvoir être représentée comme dans l'équation (9.2), car dans le cas contraire aucune constante B telle que recherchée n'existe et il n'y pas de solution à l'équation (9.1). Il est aussi possible de montrer que cette condition est suffisante : si l'on peut trouver une solution au problème de l'équation (9.2), alors il existe une constante B modifiant la différence binaire signée de telle façon que l'on obtienne finalement la différence binaire voulue.

Puisqu'une solution à une instance supercroissante du problème du sac à dos est unique, la solution de (9.2) est aussi unique. Cela signifie que l'on connaît l'unique différence binaire signée $\Delta^{\pm}(u, u + \partial(y, y')) = (r_0, \ldots, r_{31})$ qui est compatible avec la différence binaire $\Delta^{\oplus}(z, z')$ et donnant la différence additive $\partial(y, y')$. Cependant, une unique différence binaire signée correspond à plusieurs paires $(u, u+\partial(y, y'))$. Si à une position particulière $j \in I$, nous avons $r_j = -1$, alors nous sommes certains qu'en cette position la valeur du j-ième bit de u doit changer de 1 à 0. De la même manière, si nous avons $r_j = 1$, le j-ième bit de u doit changer de 0 à 1. Les bits restants de u (correspondants aux positions contenant des 0 dans $\Delta^{\pm}(u, u + \partial(y, y'))$) peuvent avoir n'importe quelle valeur. Ainsi, il nous est très facile de déterminer l'ensemble $\mathcal{U}$ des valeurs u recherchées. Il est de plus évident que $\mathcal{U}$ contient toujours au moins un élément.

À présent, puisque $u = y + B$ pour tout $u \in \mathcal{U}$, l'ensemble $\mathcal{B}$ de toutes les constantes B qui satisfont l'équation (9.1) est simplement $\mathcal{B} = \{u \oplus y : u \in \mathcal{U}\}$. Ce raisonnement montre aussi que si l'on peut obtenir une microcollision sur l'un des registres de l'état interne, il y a $|\mathcal{B}| = 2^{32-h_w(z\oplus z')}$ constantes aboutissant à une microcollision si le bit de poids fort de $z \oplus z'$ est nul et $2^{32-h_w(z\oplus z')+1}$ si ce bit est égal à 1. Cette distinction est due au fait que si $31 \in I$, nous n'avons pas besoin de changer u_{31} d'une manière particulière (c'est-à-dire soit $1 \to 0$ ou soit $0 \to 1$), tout changement est satisfaisant puisque nous n'introduisons plus de retenues.

Finalement, puisqu'aucune propriété des fonctions f et g n'a été utilisée durant le raisonnement précédent, celui-ci reste valable non seulement pour la recherche de microcollisions dans Q_R, mais aussi pour toute structure de ce type utilisant n'importe quelle fonction à la place de f ou de g.

9.4 Une première tentative de recherche d'un chemin différentiel

Dans la section 9.2, nous avons étudié le chemin différentiel du tableau 9.4 qui comporte une différence additive d_0 sur le registre B en entrée des quatre branches. Si l'on souhaite que $d_0 = d_1 = d_2 = d_3$ simultanément pour chaque branche, cela se produira avec probabilité P_d^{12} puisqu'une différence additive peut être modifiée après application d'une autre opération que l'addition ou la soustraction. Ainsi, le facteur 12 provient du fait que l'on force la constance de la différence additive durant trois tours sur les quatre branches simultanément. Nous devons donc à présent forcer l'apparition de microcollisions pour le quatrième tour de chaque branche. Pour cela, à l'entrée de ce quatrième tour, nous devons fixer les registres (E, F, G, H) à certaines valeurs qui sont facilement calculables grâce à la méthode explicitée dans la précédente section. Nous expliquons ici comment fixer les registres à ces valeurs. Nous ne nous occupons pas de la propagation des différences pour le reste des tours puisqu'aucune microcollision ne devra être forcée après le quatrième tour (d_4, d_5 et d_6 peuvent prendre un valeur arbitraire).

9.4.1 Une pseudo-presque collision au septième tour

Notre principal outil ici sera un bon ordonnancement dans la détermination des mots de message pour pouvoir imposer aux quadruplets (E, F, G, H) de chaque branche les valeurs voulues. Pour cela, nous allons tout d'abord étudier les relations entre les mots de message et les mots de la variable de chaînage d'une part et les quadruplets de chaque branche d'autre part.

Avant d'entrer dans les détails, et pour simplifier l'étude des ces relations dans chaque branche j, il faut remarquer que fixer la valeur du quadruplet $(E_{j,3}, F_{j,3}, G_{j,3}, H_{j,3})$ est équivalent à forcer celle du quadruplet $(E_{j,3}, F_{j,3}, F_{j,2}, F_{j,1})$. Ceci peut être facilement vérifié en parcourant les tours à l'envers dans les branches de FORK-256, ce qui nous donne les équations suivantes :

$$\begin{aligned} (a) \quad F_{j,2} &= \big(G_{j,3} \oplus f(F_{j,3})\big) - g(F_{j,3} - \delta_{\pi_j(5)}), \\ (b) \quad G_{j,2} &= \big(H_{j,3} \oplus f(F_{j,3})^{\lll 5}\big) - g(F_{j,3} - \delta_{\pi_j(5)})^{\lll 9}, \\ (c) \quad F_{j,1} &= \big(G_{j,2} \oplus f(F_{j,2})\big) - g(F_{j,2} - \delta_{\pi_j(3)}). \end{aligned} \tag{9.4}$$

Ainsi, l'équation (a) permet de directement fixer $F_{j,2}$, puis $F_{j,1}$ est déduit des équations (a), (b) et (c). En prenant en compte cette remarque, le tableau 9.5 donne toutes les relations à étudier. Les quadruplets à fixer à des valeurs précalculées composent la colonne de gauche et chaque ligne indique la dépendance des mots de ces quadruplets en des mots de message ou de variable de chaînage.

En vue du tableau 9.5, nous proposons l'algorithme suivant pour assigner séquentiellement les valeurs des mots de message et de variable de chaînage de telle manière que les quadruplets dans chaque branche soient égaux aux valeurs prédéfinies. Certains ajustements sont ajoutés pour aider la différence introduite initialement dans le registre B de chaque branche à se propager sans être modifiée durant les 3 premiers tours. Il est par exemple possible de s'assurer que cela se produit en forçant à zéro la valeur d'entrée de la fonction g.

1. *Initialiser.* Choisir A_0 de façon aléatoire.
2. *Ajuster.* Assigner à m_0, m_5, m_7 et m_{14} les valeurs suivantes : $m_0 = -(A_0 + \delta_{\pi_1(0)})$, $m_{14} = -(A_0 + \delta_{\pi_2(0)})$, $m_7 = -(A_0 + \delta_{\pi_3(0)})$, $m_5 = -(A_0 + \delta_{\pi_4(0)})$.
3. *Forcer les mots $G_{j,3}$.* Choisir D_0 tel que $F_{3,2}$ ait la valeur correcte. Ensuite, choisir m_3, m_9, et m_8 un par un, tels que $F_{1,2}$, $F_{2,2}$, et $F_{4,2}$ atteignent la bonne valeur.
4. *Forcer les mots $H_{j,3}$.* (Si cette étape a déjà été exécutée 2^{32} fois, retourner à l'étape 1.) Choisir H_0 de façon aléatoire. Ajuster m_{11} et m_1 pour éviter que la différence introduite ne soit modifiée durant la deuxième étape dans les branches 2 et 4 respectivement. Ensuite, fixer les mots $\overline{E_0}$, m_{15}, m_6, et m_{12} tels que $F_{1,1}$, $F_{2,1}$, $F_{3,1}$, et $F_{4,1}$ respectivement atteignent la bonne valeur.
5. *Forcer les mots $F_{j,3}$.* Choisir $\overline{C_0}$ tel que $F_{4,3}$ ait la valeur correcte. Ensuite, fixer m_{10} et m_2 un à un, tels que $F_{2,3}$ et $F_{3,3}$ respectivement aient la bonne valeur. À cet instant de l'attaque, $F_{1,3}$ est complètement défini. Si sa valeur est celle attendue, continuer à l'étape suivante, sinon retourner à l'étape 4.
6. *Forcer les mots $E_{1,3}$, $E_{2,3}$, et $E_{4,3}$.* (Si cette étape a déjà été exécutée 2^{32} fois, retourner à l'étape 1.) Choisir G_0 de façon aléatoire. Fixer $\overline{B_0}$ tel que $E_{4,3}$ ait la valeur correcte

	variable de chaînage								blocs de message m_i															
	$\overline{A_0}$	$\overline{B_0}$	$\overline{C_0}$	$\overline{D_0}$	$\overline{E_0}$	$\overline{F_0}$	$\overline{G_0}$	$\overline{H_0}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$E_{4,3}$	–	x		–	–		–	–		–				–			–			x	–			–
$E_{3,3}$	–	x		–	–		–	–							–	–			–		x	–	–	
$E_{2,3}$	–	x		–	–		–	–					x				–	–		–			–	–
$E_{1,3}$	–	x		–	–		–	–	–	–	–	–	–			x								
$F_{4,3}$	–		x		–			–	x	–				–							–			
$F_{3,3}$	–		x		–			–			x				–	–			–					
$F_{2,3}$	–		x		–			–											x	–			–	–
$F_{1,3}$	–		x		–			–	–	–	–			x										
$G_{4,3} \leftrightarrow F_{4,2}$	–			x										–			x							
$G_{3,3} \leftrightarrow F_{3,2}$	–			x												–							x	
$G_{2,3} \leftrightarrow F_{2,2}$	–			x														x					–	
$G_{1,3} \leftrightarrow F_{1,2}$	–			x					–			x												
$H_{4,3} \leftrightarrow F_{4,1}$					x																x			
$H_{3,3} \leftrightarrow F_{3,1}$					x										x									
$H_{2,3} \leftrightarrow F_{2,1}$					x																			x
$H_{1,3} \leftrightarrow F_{1,1}$					x					x														

TAB. 9.5 – Relations entre les quadruplets à fixer dans chaque branche et les mots de message et de variable de chaînage. Les symboles « – » et « x » représentent chacun un degré de liberté pour fixer la valeur d'un des mots des quadruplets, lorsque toutes les autres dépendances de la ligne correspondante ont déjà été fixées. Dans le cas d'une dépendance « x », le mot peut être *directement* fixé par le degré de liberté, c'est-à-dire la relation entre ces deux mots est simple une fois les autres dépendances fixées. Au contraire, une dépendance « – » impose à l'attaquant de parcourir les valeurs du mot apportant le degré de liberté jusqu'à aboutir à la valeur recherchée pour le mot à fixer du quadruplet correspondant. Enfin, $U \leftrightarrow V$ signifie que l'on souhaite forcer une valeur sur le mot V pour indirectement fixer le mot U grâce aux équations (a), (b) et (c).

puis fixer m_4 tel que $E_{2,3}$ atteigne la bonne valeur. À cet instant de l'attaque, $E_{1,3}$ est complètement défini. Si sa valeur est celle attendue, continuer à l'étape suivante, sinon retourner à l'étape 6.

7. *Forcer les mots* $E_{3,3}$. (Si cette étape a déjà été exécutée 2^{32} fois, retourner à l'étape 1.) Choisir m_{13} de façon aléatoire. À cet instant de l'attaque, $E_{3,3}$ est complètement défini. Si sa valeur est celle attendue, afficher la solution trouvée, sinon retourner à l'étape 7.

Cet algorithme exécute quelques recherches exhaustives *indépendantes* de taille 2^{32}. Pour cela, aucune mémoire n'est requise et la complexité moyenne est de 2^{32} exécutions d'un quart de FORK-256, et donc 2^{30} exécutions de la fonction de compression. Cependant, pour que l'attaque réussisse, la différence d doit se propager sans être modifiée jusqu'au quatrième tour. Puisque la probabilité d'un tel événement pour une étape d'une branche est égale à P_d, et

puisque nous avons déjà forcé cet événement pour la première étape (étape 2 de l'algorithme) et pour deux des branches de la deuxième étape (étape 4 de l'algorithme), la probabilité totale est égale à P_d^6.

Nous pouvons aussi remarquer que le mot F_0 de la variable de chaînage ne modifie pas les valeurs fixées pour les quadruplets de chaque branche. Ainsi, pour chaque solution fournie par notre algorithme, nous pouvons assigner à F_0 n'importe laquelle des 2^{32} valeurs possibles. En d'autres termes, notre obtenons pour chaque solution 2^{32} paires $\{(M, \mathrm{IV}), (M, \mathrm{IV}')\}$ telles qu'après application des sept premiers tours de FORK-256, une différence apparaît seulement dans les registres $A_{j,7}$ de chaque branche.

Pour conclure, nous obtenons 2^{32} solutions pour une complexité équivalente à $2^{30} \cdot P_d^{-6}$ exécutions de la fonction de compression, et le coût moyen du calcul d'une paire est approximativement de $1/4 \cdot P_d^{-6}$ exécutions de FORK-256.

9.4.2 Choisir la différence

Dans les paragraphes précédents, nous avons montré qu'une différence additive intéressante doit satisfaire deux conditions. La première est que des microcollisions puissent se produire simultanément durant le quatrième tour des quatre branches et la deuxième est que la probabilité P_d que cette différence se propage sans être modifiée soit aussi grande que possible. Puisque minimiser le poids de Hamming de la représentation de cette différence maximise P_d, nous cherchons tout d'abord des différences additives de poids 1 ou 2. Finalement, le meilleur candidat trouvé est la différence additive $d = \texttt{0x00000404}$ qui nous donne une probabilité P_d approximativement égale à 2^{-3}.

Enfin, comme expliqué précédemment, nous pouvons calculer des valeurs cibles pour les quadruplets de chaque branche pour cette différence additive $d = \texttt{0x00000404}$:

$$
\begin{array}{llllllllll}
E_{1,3} & = \texttt{0x030e9c3f}, & E_{2,3} & = \texttt{0x7e24de5c}, & E_{3,3} & = \texttt{0x00fa4d1e}, & E_{4,3} & = \texttt{0x20b7363f}, \\
F_{1,3} & = \texttt{0xa4115fb0}, & F_{2,3} & = \texttt{0x10276030}, & F_{3,3} & = \texttt{0x35edee6e}, & F_{4,3} & = \texttt{0xefc6172f}, \\
G_{1,3} & = \texttt{0x22c18168}, & G_{2,3} & = \texttt{0x4db27e00}, & G_{3,3} & = \texttt{0xd81cdc6c}, & G_{4,3} & = \texttt{0x8c2c7c00}, \\
H_{1,3} & = \texttt{0x1816822c}, & H_{2,3} & = \texttt{0x27e004db}, & H_{3,3} & = \texttt{0xcdc6bd82}, & H_{4,3} & = \texttt{0xc7bff8c3}.
\end{array}
$$

9.4.3 Pseudo-presque collisions pour la fonction de compression

Une version réduite à sept tours.

Nous nous focalisons tout d'abord sur une version réduite à sept tours (au lieu de huit) de FORK-256 : le huitième tour est omis, mais les additions finales du rebouclage et la permutation finale du huitième tour sont conservées. Les différences $A_{j,7}$ en sortie du septième tour (seules restantes après exécution de notre algorithme) vont se déplacer en $B_{j,8}$ puisque la permutation finale du huitième tour n'a pas été retirée. Après addition du rebouclage, qui comporte une différence seulement en $B_{j,0}$, il semble que l'on puisse trouver une pseudo-collision pour cette fonction de compression. Néanmoins, toutes ces différences auront leur bit non nul de poids le plus faible à la même position, définie par la position du bit non nul de poids le plus faible dans $d = \texttt{0x00000404}$. Dans notre cas, la position est le troisième bit de poids faible. On additionnera donc de telles différences pour les 4 branches et aussi pour le rebouclage, ce qui donnera un nombre impair de termes et imposera la valeur 1 au troisième bit de poids

faible de la différence du deuxième mot de sortie de la fonction de compression. Obtenir une pseudo-collision par cette méthode est donc impossible, le meilleur résultat possible étant une pseudo-presque collision avec seulement un bit de différence en sortie. Nous avons alors estimé la probabilité d'une telle attaque comme suit : après avoir choisi des valeurs aléatoires (vérifiant néanmoins le chemin différentiel du tableau 9.4) pour les registres internes de chaque branche après application du quatrième tour, nous exécutons le reste de la fonction de compression de cette version réduite de FORK-256 et vérifions le type de pseudo-presque collisions obtenues. La probabilité d'une pseudo-presque collision avec seulement un bit de différence en sortie a pu être estimée à 2^{-15} (sur 2^{32} expérimentations, 127665 sorties étaient des pseudo-presque collisions avec seulement un bit de différence). Enfin, puisque notre algorithme nous fournit 2^{32} candidats valides avec $2^{30}/P_d^6 = 2^{49}$ exécutions de la fonction de compression, la complexité pour trouver 2^{17} paires distinctes de pseudo-presque collisions avec seulement un bit de différence en sortie est approximativement égale à 2^{49} appels à la fonction de compression de FORK-256 (d'où une complexité moyenne de $2^{49+15-32} = 2^{32}$ par solution).

La fonction de compression complète de FORK-256**.**

L'algorithme précédemment décrit nous fournit 2^{32} paires de variables de chaînage et de mots de message pour lesquelles la sortie de la fonction de compression de FORK-256 est identique sur quatre des huit mots pour une complexité de 2^{49} exécutions. Il reste en effet à annuler un bit de différence (sur une position fixée) dans le second registre et les différences sur trois mots de 32 bits (dans les troisième, quatrième et cinquième registres). La probabilité d'une pseudo-presque collision avec 1 bit de différence sur le second mot a été évaluée à 2^{-15}, et celle d'annuler les différences sur l'un des trois registres suivants a été évaluée à 2^{-31} pour la différence initiale $d = \texttt{0x0000404}$. Ainsi, la complexité finale pour trouver une pseudo-presque collision avec 1 bit de différence pour la fonction de compression de FORK-256 est égale à $2^{49+93+15-32} = 2^{125}$. De la même manière, la probabilité de trouver une pseudo-presque collision avec 2 bits de différence sur le deuxième registre est égale à 2^{-10}, ce qui nous donne une complexité finale de $2^{49+93+10-32} = 2^{120}$ pour calculer une telle collision sur la sortie complète de la fonction de compression de FORK-256.

Résultats expérimentaux.

Dans la figure 9.3, nous donnons comme exemple une pseudo-presque collision avec 22 bits de différence, obtenue par l'algorithme décrit en section 9.4.1 avec les valeurs cibles des quadruplets et la différence d de la section 9.4.2. Cette solution donne aussi une pseudo-presque collision avec 2 bits de différence pour la version de FORK-256 réduite à sept tours.

9.5 Trouver des chemins différentiels pour FORK-256

9.5.1 Principe général

Dans cette section, nous retournons au problème initial de trouver des chemins différentiels pour les quatre branches de FORK-256, et donner une solution la plus générale possible. En supposant que l'on puisse éviter la propagation des différences dans les structures Q_L et Q_R (c'est-à-dire que l'on sait comment trouver des microcollisions) et que les différences additives restent inchangées dans les registres B, C, D et F, G, H (si $P_d = 1$), alors les seuls emplacements où des modifications de différences peuvent intervenir sont les registres A et E, après addition

Variable de chaînage entrante H_i

$A_0 =$ 0x8406e290	$B_0 =$ 0x5988c6af	$C_0 =$ 0x76a1d478	$D_0 =$ 0x0eb60cea
$E_0 =$ 0xf5c5d865	$F_0 =$ 0x458b2dd1	$G_0 =$ 0x528590bf	$H_0 =$ 0xc3bf98a1
$A'_0 =$ 0x8406e290	$B'_0 =$ 0x5988cab3	$C'_0 =$ 0x76a1d478	$D'_0 =$ 0x0eb60cea
$E'_0 =$ 0xf5c5d865	$F'_0 =$ 0x458b2dd1	$G'_0 =$ 0x528590bf	$H'_0 =$ 0xc3bf98a1

Message

$m_0 =$ 0x396eedd8	$m_1 =$ 0x0e8c2a93	$m_2 =$ 0xb961f8a4	$m_3 =$ 0xf0a06fc6
$m_4 =$ 0x9935952b	$m_5 =$ 0xe01d16c9	$m_6 =$ 0xddc60aa4	$m_7 =$ 0x0ac1d8df
$m_8 =$ 0xc6fef1d8	$m_9 =$ 0x4c472ca6	$m_{10} =$ 0x58d9322d	$m_{11} =$ 0x2d087b65
$m_{12} =$ 0x7c8e1a26	$m_{13} =$ 0x71ba5da1	$m_{14} =$ 0xba5d2bfc	$m_{15} =$ 0x1988f929

Variable de chaînage sortante H_{i+1}

$A_0 =$ 0x9897c70a	$B_0 =$ 0x4e18862d	$C_0 =$ 0xb4725ac1	$D_0 =$ 0xcfc9f92c
$E_0 =$ 0x9aa0637d	$F_0 =$ 0xae772570	$G_0 =$ 0x74dd4af1	$H_0 =$ 0xcd444dd7
$A'_0 =$ 0x9897c70a	$B'_0 =$ 0x4e1880f9	$C'_0 =$ 0x1e677302	$D'_0 =$ 0x4c65096 6
$E'_0 =$ 0xf4792bf4	$F'_0 =$ 0xae772570	$G'_0 =$ 0x74dd4af1	$H'_0 =$ 0xcd444dd7

FIG. 9.3 – Une pseudo-presque collision, sans différence sur le message, avec 22 bits de différence pour la fonction de compression de FORK-256. Les valeurs sont données en notation hexadécimale et les octets comportant des différences sont soulignés.

modulaire d'une différence dans un des mots de message. Ainsi, les valeurs différentielles des registres lors des tours peuvent être vues comme des fonctions linéaires (au sens de l'addition modulaire) des registres de la variable de chaînage $(\overline{A_0}, \ldots, \overline{H_0})$ et des mots de message $m_0, \ldots, m_{15}$.

Si l'on considère le cas le plus général et que l'on suppose, de façon très optimiste, que deux différences modulaires identiques peuvent s'annuler, nous travaillons alors dans $\mathbb{F}_2$ et les différences dans tous les registres sont des combinaisons $\mathbb{F}_2$-linéaires des différences $\Delta\overline{A_0}, \ldots, \Delta\overline{H_0}$ et $\Delta m_0, \ldots, \Delta m_{15}$ (qui sont maintenant vues comme des éléments de $\mathbb{F}_2$). Les différences de sortie $(\Delta A, \ldots, \Delta H)$ de la fonction de compression (après rebouclage) sont aussi des combinaisons linéaires de celles de $S = (\Delta\overline{A_0} \ldots, \Delta\overline{H_0}, \Delta m_0, \ldots, \Delta m_{15})$, ce qui peut être représenté par une application $\mathbb{F}_2$-linéaire $(\Delta A, \ldots, \Delta H) = L_{out}(S)$. Cela signifie que l'on peut trouver l'ensemble $\mathcal{S}_c$ de tous les vecteurs S de différences en entrée n'aboutissant à aucune différence en sortie de la fonction de compression, simplement en cherchant le noyau de cette application $\mathcal{S}_c = \ker(L_{out})$.

Pour minimiser la complexité de l'attaque finale, nous pouvons rajouter la condition que l'on souhaite trouver des chemins différentiels de poids le plus petit possible. Puisqu'une dif-

férence dans un registre à chaque tour est une fonction linéaire des différences $\Delta\overline{A_0}\ldots,\Delta\overline{H_0}$, $\Delta m_0,\ldots,\Delta m_{15}$ et que l'on a seulement 2^{24} possibilités, une approche logique consiste à énumérer toutes ces possibilités et à garder les solutions minimisant le nombre de registres contenant une différence non nulle. Les différences dans les registres autres que A et E ne contribuent que peu à la complexité totale de l'attaque, car elles ne demandent pas de forcer des microcollisions, ce qui est le plus coûteux. Ainsi, en ne prenant en compte que les différences dans les registres A et E, nous ne considérons que celles qui risquent de poser problème.

9.5.2 Généralisation de la recherche

Il est possible de généraliser l'approche décrite précédemment. Suivant que l'on force ou non une microcollision sur l'un des registres en sortie de Q_L ou Q_R, nous avons huit façons différentes de modéliser la propagation d'une différence à travers de telles structures. En utilisant le modèle linéaire supposant que les différences s'annulent, nous pouvons exprimer les différences de sortie de chaque structure Q_L :

$$\begin{aligned} \Delta A_{i+1} &= \Delta A_i \ , & \Delta C_{i+1} &= \Delta C_i + q_C \cdot \Delta A_i \ , \\ \Delta B_{i+1} &= \Delta B_i + q_B \cdot \Delta A_i \ , & \Delta D_{i+1} &= \Delta D_i + q_D \cdot \Delta A_i \ . \end{aligned}$$

où $q_B, q_C, q_D \in \mathbb{F}_2$ sont des coefficients fixés caractérisant les structures Q_L. Le raisonnement est identique pour les structures Q_R. Ainsi, nous avons 8^{64} modèles linéaires possibles pour FORK-256 si l'on autorise des microcollisions plus variées. Cette liberté va nous permettre de diminuer le nombre de structures Q actives, mais va cependant ajouter des conditions supplémentaires pour annuler les différences arrivant en entrée des différentes parties de la structure.

Les résultats finaux de notre recherche de chemins différentiels pour la fonction de compression de FORK-256 sont résumés dans le tableau 9.6. On peut observer qu'en introduisant un modèle plus complexe de propagation des différences à travers les structures Q, on diminue grandement le nombre de microcollisions nécessaires par rapport au cas où une microcollision devrait nécessairement apparaître simultanément sur les trois derniers registres en sortie de la structure Q. Le tableau montre aussi qu'il est possible d'obtenir une collision en n'ajoutant qu'une seule différence dans m_{12} et en n'ayant que six structures Q dans lesquelles des microcollisions devront être forcées. Nous étudions dans la prochaine section comment utiliser ce chemin différentiel pour calculer des presque collisions, mais aussi des collisions pour la fonction de compression de FORK-256.

9.6 Collisions pour la fonction de compression de FORK-256

Dans cette section, nous montrons comment utiliser le chemin différentiel ne comportant que des différences sur m_{12} et trouvé par la méthode de la section 9.5 pour obtenir des différences très faibles en sortie de la fonction de compression de FORK-256. Nous donnons alors deux stratégies différentes pour trouver des collisions pour cette fonction de compression, plus rapidement que la borne théorique obtenue par le paradoxe des anniversaires dans le cas d'une fonction de hachage idéale.

Notre attaque exploite le fait que si l'on introduit des différences additives seulement dans m_{12}, si l'on arrive à trouver des microcollisions dans le premier et le cinquième tour de la quatrième branche et dans le quatrième tour de la troisième branche, et enfin si l'on parvient à

Scénario	Branches	m	Différences dans	Structures Q actives
Pseudo-collisions	1,2,3,4	5	$\overline{H_0}, m_2, m_{11}$	12 :000, 25 :000, 35 :001, 41 :001, 51 :010
Collisions	1,2,3,4	6	m_{12}	13 :000, 31 :001, 40 :000, 47 :100, 50 :000, 57 :000
Pseudo-collisions	1,2,3	2	$\overline{B_0}, m_{12}$	8 :100, 24 :0
	1,2,4	3	$\overline{H_0}, m_{11}$	3 :000, 51 :010, 60 :000
	1,3,4	3	$\overline{H_0}, m_2$	35 :001, 44 :000, 51 :000
	2,3,4	3	$\overline{D_0}, m_9$	36 :010, 43 :000, 52 :000
Collisions	1,2,3	3	m_0, m_3, m_9	1 :001, 20 :010, 39 :100
	1,2,4	4	m_1, m_2	2 :001, 9 :000, 25 :100, 51 :000
	1,3,4	5	m_9	10 :000, 39 :001, 42 :001 43 :010, 59 :000
	2,3,4	5	m_3, m_9	20 :010, 27 :000, 39 :000 57 :000, 59 :010

TAB. 9.6 – Nombre minimal m de structures Q actives selon différents scénarios de chemins différentiels pour la fonction de compression de FORK-256. Les structures Q sont numérotées de 1 à 64 où 1 correspond à Q_L dans le premier tour de la première branche et 64 à Q_R dans le dernier tour de la quatrième branche. La notation N :bcd signifie que dans la structure Q numéro N, la différence d'entrée de A (E dans le cas de Q_R) se propagera aux registres B, C et D respectivement (F, G et H dans le cas de Q_R) si $b = 1$, $c = 1$, $d = 1$ respectivement. Par exemple, le chemin différentiel de la figure 9.4 peut être encodé comme 13 :110, 31 :111, 40 :000, 47 :111, 50 :000, 57 :000.

éviter la propagation de la différence de $A_{1,6}$ à $E_{1,7}$ dans la première branche, alors la différence en sortie sera confinée aux registres B, C, D, et E, c'est-à-dire à seulement 128 bits des 256 bits de sortie de la fonction de hachage. Le chemin différentiel considéré est explicité dans la figure 9.4.

Comme indiqué précédemment, le nombre de bits affectés en sortie peut être réduit par un choix judicieux de la différence additive utilisée dans m_{12} : les différences maximisant la position p du bit non nul de poids le plus faible permettront de confiner les différences sur les positions supérieures ou égales à p dans le registre B de la sortie de la fonction de compression.

Dans la prochaine section, nous explicitons une première méthode permettant de trouver des presque collisions et des collisions, ne nécessitant qu'un faible coût en mémoire. Ensuite, dans la section suivante, nous montrons comment tirer parti d'une certaine quantité de mémoire pour précalculer des tables qui seront utiles durant le déroulement de l'attaque, pour diminuer la complexité totale en temps.

9.6.1 Trouver des collisions avec peu de mémoire

L'attaque se décompose en deux phases. Durant la première étape, nous devons simultanément forcer des microcollisions au premier tour et au cinquième tour de la quatrième branche et au quatrième tour de la troisième branche pour une différence additive injectée dans le message m_{12}. Durant la seconde étape, nous utilisons les mots de message m_4 et m_9, laissés libres

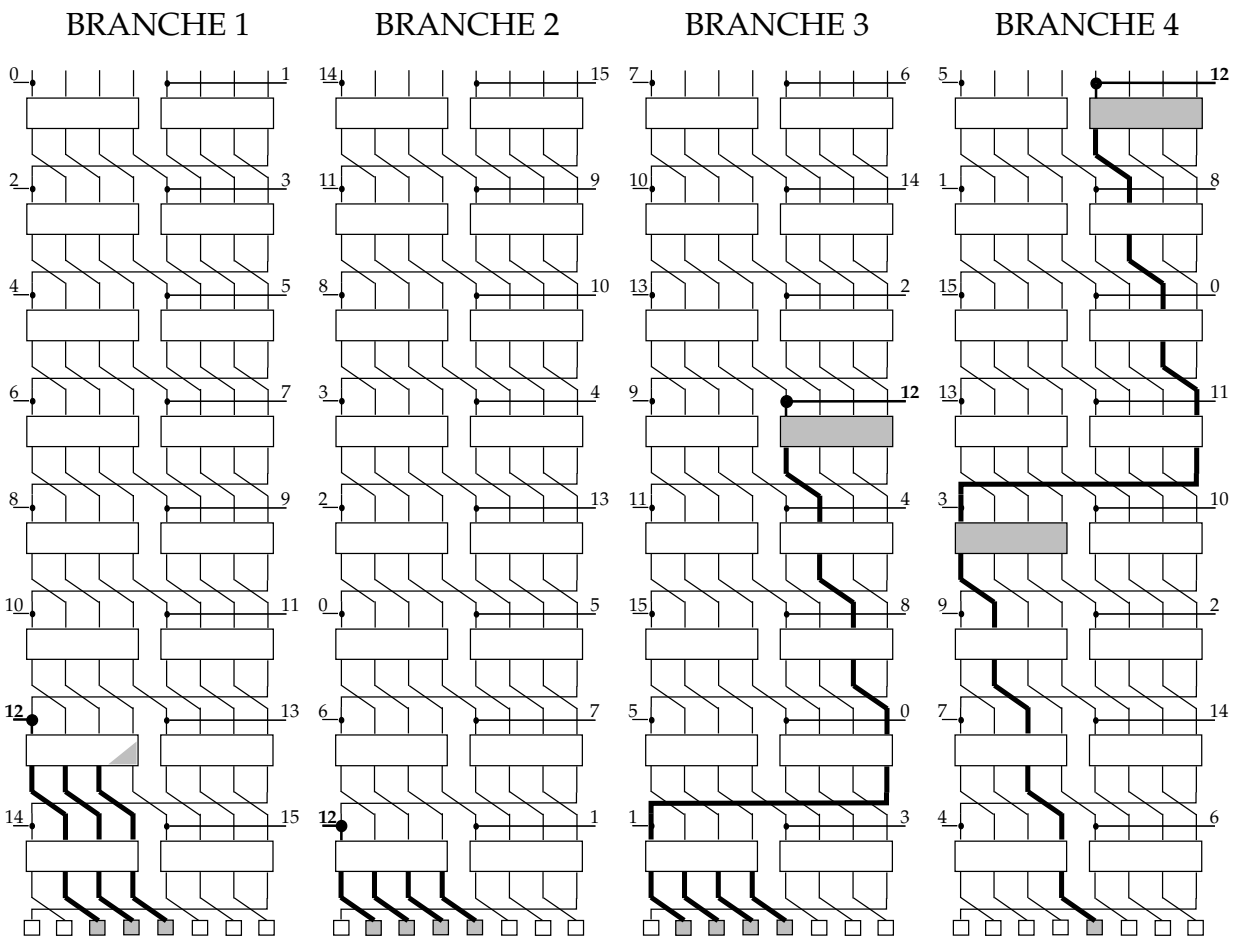

FIG. 9.4 – Chemin différentiel utilisé pour calculer des collisions pour la fonction de compression de FORK-256. Les lignes en gras montrent la propagation des différences. Les structures Q nécessitant des microcollisions sont grisées. Les nombres indiquent l'ordonnancement des mots de message.

à ce point de l'attaque, pour éviter la propagation de la différence présente dans $A_{1,6}$ à $E_{1,7}$. Cette microcollision, uniquement sur un seul registre, intervient durant le septième tour de la première branche.

Trouver des microcollisions dans la troisième et la quatrième branche.

On suppose ici qu'une différence additive d adéquate a déjà été choisie. Nous procédons alors comme suit :

1. *Quatrième branche, premier tour.* Choisir x_1 tel que la paire $(x_1, x_1 + d)$ puisse aboutir à des microcollisions dans Q_R pour le premier tour dans la quatrième branche. Fixer alors $m_{12} = x_1 - \overline{E_0}$ et assigner les valeurs correctes à $\overline{F_0}$, $\overline{G_0}$, et $\overline{H_0}$ pour réaliser les microcollisions sur les trois registres.

2. *Quatrième branche, cinquième tour.* Donner des valeurs aléatoires à m_5, m_1, m_8, m_{15}, m_0, m_{13}, et m_{11}. Calculer ensuite la première moitié de la quatrième branche, jusqu'au cin-

quième tour, et trouver une paire de valeurs $(x_2, x_2 + d^*)$ (où d^* est la différence additive présente dans le registre $A_{4,4}$) pouvant aboutir à des microcollisions dans Q_L. Calculer alors les valeurs ρ_1, ρ_2 et ρ_3 à imposer en entrée des trois registres $B_{4,4}$, $C_{4,4}$ et $D_{4,4}$ pour obtenir les microcollisions. Si aucune solution n'existe, recommencer cette étape, sinon
– Fixer $m_3 = x_2 - A_{4,4}$.
– Fixer $m_{13} = \rho_1 - A_{4,3} - \delta_8$ tel que $B_{4,4}$ soit égal à la valeur ρ_1.
– Fixer $m_{15} = [\rho_2 \oplus g(B_{4,4})] - f(B_{4,4} - \delta_8) - A_{4,2} - \delta_{10}$ tel que $C_{4,4} = \rho_2$.
– Fixer m_1 tel que $D_{4,4} = \rho_3$.
Il faut à présent ajuster m_0 et m_{11} pour compenser les modifications de m_1 et de m_{15}.

3. *Troisième branche.* Trouver une paire de valeurs $(x_3, x_3 + d)$ permettant des microcollisions dans Q_R au niveau de la quatrième étape de la troisième branche et calculer les constantes correspondantes λ_1, λ_2, et λ_3 que doivent atteindre les registres $F_{3,3}$, $G_{3,3}$ et $H_{3,3}$. De la même manière que précédemment, fixer m_2 tel que $F_{3,3} = \lambda_1$, m_{14} tel que $G_{3,3} = \lambda_2$, et m_6 tel que $H_{3,3} = \lambda_3$. Il faut à présent ajuster m_{10} pour compenser les modifications de m_6. Il faut aussi compenser la modification de m_{14}, mais m_{13} ne peut être ajusté étant donné que ce mot de message a déjà été fixé dans la quatrième branche. À la place, nous sélectionnons $\overline{B_0}$ de telle manière que $E_{3,3}$ soit égal à la bonne valeur x_3.

4. *Quatrième branche.* la modification de $\overline{B_0}$ est la seule des modifications effectuées durant l'étape de la troisième branche pouvant influer sur les microcollisions déjà forcées dans la quatrième branche. En effet, la valeur du registre $A_{4,4}$ sera changée. Cependant, cela peut être corrigé en ajustant encore une fois m_{11}.

À l'issue de cette procédure, nous obtenons une paire de mots de message et de mots de variable de chaînage vérifiant les deux dernières branches du chemin différentiel présenté dans la figure 9.4. Pour le reste de l'attaque, nous allons utiliser les mots de message m_4 et m_9 qui n'ont pas encore été fixés.

Une microcollision dans la première branche.

Nous devons à présent traiter la première et de la deuxième branche. La deuxième branche ne demande aucun travail de la part de l'attaquant : m_{12} apparaît dans le dernier tour de la deuxième branche et cela n'induit des différences que dans les registres B, C, D, et E de la sortie de la fonction de compression. La première branche est plus problématique, car nous devons y forcer une microcollision dans $D_{1,6} \rightarrow E_{1,7}$ durant le septième tour. Il ne semble pas y avoir de meilleur moyen pour y parvenir que de tester aléatoirement des instances des mots de message m_4 et m_9 jusqu'à obtenir cette microcollision. La probabilité de succès de cette approche dépend largement de la différence additive utilisée, et le tableau 9.7 nous montre les deux meilleurs candidats obtenus par expérimentation.

Analysons à présent la complexité de la recherche de cette microcollision en termes de nombre d'appels à la fonction de compression de FORK-256. Étant donné une différence additive, soit η le nombre de valeurs de $A_{1,6}$ valides pour obtenir une microcollision. Les valeurs valides de $A_{1,6}$ sont les valeurs x pour lesquelles il existe une constante qui aboutit à une microcollision pour la paire $(x, x + d)$. Par exemple, pour la différence additive $d =$`22f80000`, nous avons $\eta = 2^{21.7}$ valeurs de $A_{1,6}$ valides.

Notre algorithme concernant la première branche est alors comme suit :

1. *Initialiser.* Fixer m_4 à 0.

différence d	η	probabilité observée
`0xdd080000`	$2^{21.7}$	$2^{-24.6}$
`0x22f80000`	$2^{21.7}$	$2^{-24.6}$

TAB. 9.7 – Meilleures différences additives d trouvées par expérimentation et leur probabilité respective de fournir une microcollision dans $D_{1,6} \rightarrow E_{1,7}$ durant le septième tour de la première branche. Le nombre de valeurs d'entrée de $A_{1,6}$ pouvant aboutir à une microcollision est donné dans la colonne η.

2. *Précalculer une table.* Calculer tous les registres internes jusqu'au septième tour. Alors, pour chaque valeur valide x, poser $A_{1,6} = x$ et revenir en arrière pour obtenir la valeur $H_{1,5}$ correspondante, puis mémoriser le résultat dans une table T.
3. *Chercher un m_9 valide.* Pour toutes les valeurs possibles de m_9, calculer la valeur de $H_{1,5}$ correspondante et chercher si ce résultat apparaît dans la table T. Si c'est le cas, aller à l'étape 4. Une fois toutes les valeurs de m_9 testées, incrémenter m_4 et revenir à l'étape 2.
4. *Vérifier.* Si la valeur actuelle de m_9 donne une microcollision dans $D_{1,6} \rightarrow E_{1,7}$ alors, afficher la solution trouvée, sinon retourner à l'étape 3.

L'étape 2 représente $1/64$ d'un calcul complet de la fonction de compression de FORK-256 pour toutes les η valeurs valides. Ainsi, sa complexité est de $\eta/64 = 2^{15.7}$ calculs de FORK-256. L'étape 3 représente $1/64$ d'un calcul complet de la fonction de compression de FORK-256 pour les 2^{32} valeurs de m_9 testées. Donc, sa complexité est de 2^{26} calculs de FORK-256. Enfin, puisque l'étape 4 réussit avec une probabilité $2^{-24.6}$ (voir tableau 9.7), nous obtenons $2^{7.4}$ solutions pour une complexité de 2^{26}. Le coût moyen de notre algorithme pour trouver une solution est donc d'approximativement $2^{18.6}$ évaluations de FORK-256.

Complexité totale de l'attaque.

On peut vérifier expérimentalement que pour la différence additive $d =$ `0xdd080000` en entrée, la distribution des différences en sortie de la fonction de compression de FORK-256 sur les 108 bits affectés (il y a en tout 109 bits pouvant contenir une différence, mais la différence sur le bit 19 du registre B sera toujours annulée) est très proche de la distribution uniforme. Ainsi, nous pouvons espérer trouver une collision en engendrant 2^{108} paires vérifiant le chemin différentiel de la figure 9.4. Enfin, puisque chaque solution nous coûte en moyenne $2^{18.6}$ calculs de la fonction de compression de FORK-256, nous obtenons une attaque par collision de complexité $2^{126.6}$, ce qui est très proche de la borne théorique dans le cas d'une fonction de compression idéale. L'attaque nécessite une mémoire d'approximativement $2 \cdot 2^{22}$ mots de 32 bits pour le stockage des tables précalculées.

9.6.2 Amélioration de l'attaque à l'aide de tables précalculées

Dans cette section, nous montrons comment améliorer l'attaque précédemment décrite par l'utilisation de tables précalculées. Considérons le mécanisme de diffusion lorsqu'aucune différence n'est impliquée. À chaque tour, les huit registres sont traités quatre par quatre : à gauche nous avons (A, B, C, D) puis (E, F, G, H) à droite. Prenons l'exemple de (E, F, G, H), l'autre cas étant identique. Le bloc de message m arrivant sur le registre E permet de fixer le registre

de sortie F à n'importe quelle valeur désirée, mais quelle est son action sur les trois autres registres de sortie, par exemple le registre H ? En moyenne, pour tout registre d'entrée G, il existe une valeur du bloc de message telle que le registre H soit égal à une valeur prédéfinie. Ainsi, pour toute valeur prédéfinie du registre de sortie H, on peut construire une table T_H contenant les valeurs (G, y) telles que $\psi_G(y) = H$. La construction de la table, exécutée avant la recherche de collisions, demande moins de 2^{32} appels à la fonction de compression de FORK-256 et 2^{32} blocs de mémoire. En construisant 2^{32} tables de ce type (une pour chaque valeur possible de H), pour toute paire (G, H) il devient possible avec forte probabilité de trouver un bloc de message tel que le registre d'entrée G aboutisse effectivement au registre de sortie H à travers l'exécution du tour. Le coût total de construction des tables est de 2^{64} en temps et en mémoire, mais l'accès à une table est en temps constant.

Pour notre attaque, nous utiliserons plusieurs tables. La première, T_{10}, sera utilisée pour contrôler la transition $C_{3,1} \rightarrow D_{3,2}$ grâce à m_{10}, c'est-à-dire $m_{10} = T_{10}(C_{3,1}, D_{3,2})$. Une autre famille de tables, $T_{9,a}$, sera utilisée pour déterminer quelle valeur de m_9 produira la transition recherchée $E_{1,4} \rightarrow A_{1,6}$ étant donné m_{11} fixé, c'est-à-dire $m_9 = T_{9,a}(E_{1,4}, m_{11})$ doit satisfaire $A_{1,6} = a$, où a est une valeur fixée (il y a 253 valeurs de a pour lesquelles la probabilité d'une microcollision est de 2^{-8}).

Comme dans l'attaque précédente, nous utilisons le chemin différentiel de la figure 9.4 en injectant une différence seulement dans le mot de message m_{12}, et en cherchant des microcollisions dans les zones grisées. Nous allons choisir les blocs de message suivant un certain ordonnancement pour satisfaire ces contraintes, mais contrairement à l'attaque précédente, nous choisissons à l'avance les instances de microcollisions dans la troisième et la quatrième branche. Nous devons tout d'abord nous assurer que la différence additive dans le registre $A_{4,4}$ sera la même que celle injectée m_{12}. De plus, pour la valeur de différence $d = \texttt{0xdd080000}$ que nous allons utiliser, nous ne considérons que les 253 valeurs de a pour lesquelles nous avons la plus forte probabilité que la différence d ne se diffuse pas de $A_{1,6}$ à $E_{1,7}$, c'est-à-dire qu'une seule microcollision se produise. La valeur $a - m_{12} = \texttt{0xe8db2d4b}$ en est un exemple.

1. *Initialiser.* Fixer m_{12}, F_0, G_0, et H_0 pour obtenir une microcollision dans le premier tour de la quatrième branche.

2. *Quatrième branche.* Choisir m_1 pour fixer $B_{4,2}$ à la valeur recherchée. Choisir m_5 de façon aléatoire puis ajuster m_8 pour que la différence d se propage sans être modifiée. Choisir m_{13} pour fixer $B_{4,4}$ à la valeur recherchée. Ajuster m_{11} pour que la différence d se propage sans être modifiée et choisir m_3 pour obtenir la bonne valeur de $A_{4,4}$.

3. *Troisième branche.* Choisir m_6 pour fixer $F_{3,1}$ à la valeur recherchée. Choisir m_7 de façon aléatoire puis choisir m_{14} pour fixer $F_{3,2}$ à la valeur recherchée. Utiliser la table T_{10} pour trouver m_{10} tel que $E_{3,3}$ soit à la bonne valeur (ce qui est possible, car m_5, m_7, m_{13}, et m_{14} ont déjà été fixés). Choisir m_2 pour obtenir la bonne valeur de $F_{3,3}$.

4. *Première branche.* Choisir m_4 de façon aléatoire, puis utiliser la table $T_{9,a}$ pour une certaine valeur a et ainsi décider quelle valeur de m_9 aboutira à $a = A_{1,6}$. Cette valeur a permet d'espérer une microcollision sur $E_{1,7}$ avec une probabilité de 2^{-8}. S'il n'y a pas de microcollisions pour la valeur de a choisie, recommencer cette étape avec un autre candidat. Puisque nous avons 253 candidats potentiels, nous avons une bonne probabilité d'en trouver un valide.

La complexité de cet algorithme est proche d'un seul appel à la fonction de compression de FORK-256, mais nécessite une phase de précalcul d'approximativement 2^{64} en temps et en

mémoire. Puisque 108 bits au plus diffèrent en sortie de la fonction de compression en utilisant la différence additive `0xdd080000`, notre algorithme nécessite environ 2^{108} calculs de FORK-256 pour trouver une collision.

Bibliographie

[AB96] R.J. Anderson and E. Biham. TIGER : A Fast New Hash Function. In D. Gollmann, editor, *Fast Software Encryption – FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer-Verlag, 1996.

[ANP07] E. Andreeva, G. Neven, B. Preneel and T. Shrimpton. Seven-Property-Preserving Iterated Hashing : ROX. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer-Verlag, 2007.

[AFS05] D. Augot, M. Finiasz and N. Sendrier. A Family of Fast Syndrome Based Cryptographic Hash Functions. In E. Dawson and S. Vaudenay, editors, *Progress in Cryptology – MYCRYPT 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 64–83. Springer-Verlag, 2005.

[AM07] J.P. Aumasson and W. Meier. Analysis of Multivariate Hash Functions. In K. Nyberg, editor, *Information Security and Cryptology – ICISC 2007* volume 4817 of *Lecture Notes in Computer Science*, pages 309–323. Springer-Verlag, 2007.

[AMP08] J.P. Aumasson, W. Meier and R.C.W. Phan. The Hash Function Family LAKE. In K.H. Nam and G. Rhee, editors, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[AMP04] G. Avoine, J. Monnerat and T. Peyrin. Advances in Alternative Non-adjacent Form Representations. In A. Canteaut and K. Viswanathan, editors, *Progress in Cryptology – INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2004.

[BR00] P. Barreto and V. Rijmen. The Whirlpool Hashing Function. First version in 2000 but revised in May 2003. http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html.

[Bel06] M. Bellare. New Proofs for NMAC and HMAC : Security Without Collision Resistance. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer-Verlag, 2006.

[BCK96] M. Bellare, R. Canetti and H. Krawczyk. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[BR06] M. Bellare and T. Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer-Verlag, 2006.

[BR93] M. Bellare and P. Rogaway. Random Oracles are Practical : A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[BPS06] 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. http://csrc.nist.gov/groups/ST/hash/second_workshop.html.

[BDP06] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. RadioGatun, a Belt-and-Mill Hash Function. In Proceedings of *Second NIST Cryptographic Hash Workshop*, 2006. http://csrc.nist.gov/groups/ST/hash/second_workshop.html.

[BDP08] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. On the Indifferentiability of the Sponge Construction. In N.P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 2008.

[BC04] E. Biham and R. Chen. Near-Collisions of SHA-0. In M.K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer-Verlag, 2004.

[BCJ05] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 36–57. Springer-Verlag, 2005.

[BD06] E. Biham and O. Dunkelman. A Framework for Iterative Hash Functions : HAIFA. In Proceedings of *Second NIST Cryptographic Hash Workshop*, 2006. http://csrc.nist.gov/groups/ST/hash/second_workshop.html.

[BPR07] O. Billet, M.J.B. Robshaw and T. Peyrin. On Building Hash Functions From Multivariate Quadratic Equations. In J. Pieprzyk, H. Ghodosi and E. Dawson, editors, *Information Security and Privacy – ACISP 2007*, in volume 4586 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 2007.

[BRS02] J. Black, P. Rogaway, and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer-Verlag, 2002.

[BB91] B. den Boer and A. Bosselaers An Attack on the Last Two Rounds of MD4. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 194–203. Springer-Verlag, 1991.

[BB93] B. den Boer and A. 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-Verlag, 1993.

[BCC07] E. Bresson, B. Chevallier-Mames, C. Clavier, B. Debraize, P.A. Fouque, L. Goubin, A. Gouget, G. Leurent, P.Q. Nguyen, P. Paillier, T. Peyrin and S. Zimmer. Revisiting Security Relations Between Signature Schemes and their Inner Hash Functions. In Proceedings of *ECRYPT Hash Workshop*, 2007. http://events.iaik.tugraz.at/HashWorkshop07/program.html.

[BPS90] L. Brown, J. Pieprzyk, and J. Seberry. LOKI - a Cryptographic Primitive for Authentication and Secrecy Applications. In J. Pieprzyk and J. Seberry, editors, *Advances in Cryptology – AUSCRYPT 1990*, volume 453 of *Lecture Notes in Computer Science*, pages 229–236. Springer-Verlag, 1990.

[BW99] A. Biryukov and D. Wagner. Slide Attacks. In L.R. Knudsen, editor, *Fast Software Encryption – FSE 1999*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 1999.

[Buc65] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, University of Innsbruck, 1965.

[CR06] C. De Cannière and C. Rechberger. Finding SHA-1 Characteristics : General Results and Applications. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2006.

[CMR07] C. De Cannière, F. Mendel and C. Rechberger. Collisions for 70-Step SHA-1 : On the Full Cost of Collision Search. In C.M. Adams, A. Miri and M.J. Wiener, editors, *Selected Areas in Cryptography – SAC 2007*, volume 4876 of *Lecture Notes in Computer Science*, pages 56–73. Springer-Verlag, 2007.

[CJ98] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[Coc07] M. Cochran. Notes on the Wang et al. 2^{63} SHA-1 Differential Path. ePrint archive, 2007. http://eprint.iacr.org/2007/474.pdf.

[CLS06] S. Contini, A.K. Lenstra and R. Steinfeld. VSH, an Efficient and Provable Collision Resistant Hash Function. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 165–182. Springer-Verlag, 2006.

[CMP07] S. Contini, K. Matusiewicz and J. Pieprzyk. Extending FORK-256 Attack to the Full Hash Function. In S. Qing, H. Imai and G. Wang, editors, *Information Security and Cryptology – ICISC 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 296–305. Springer-Verlag, 2007.

[CMP08] S. Contini, K. Matusiewicz, J. Pieprzyk, R. Steinfeld, G. Jian, L. San and H. Wang. Cryptanalysis of LASH. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[CY06] S. Contini and Y.L. Yin. Forgery and Partial Key-Recovery Attacks on HMAC and NMAC Using Hash Collisions. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 37–53. Springer-Verlag, 2006.

[CPM90] D. Coppersmith, S. Pilpel, C.H. Meyer, S.M. Matyas, M.M. Hyden, J. Oseas, B. Brachtl, and M. Schilling. Data Authentication Using Modification Dectection Codes Based on a Public One Way Encryption Function. U.S. Patent No. 4,908,861, March 13, 1990.

[CDM05] J.S. Coron, Y Dodis, C Malinaud and P Puniya. Merkle-Damgard Revisited : How to Construct a Hash Function. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer-Verlag, 2005.

[DR02] J. Daemen and V. Rijmen The Design of Rijndael. Springer-Verlag, 2002.

[Dam89] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989.

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

[Dea99] R.D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[DG01] C. Debaert and H. Gilbert. The RIPEMD and RIPEMD Improved Variants of MD4 Are Not Collision Free. In M. Matsui, editor, *Fast Software Encryption – FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 2001.

[Dob96a] H. Dobbertin. Cryptanalysis of MD4. In D. Gollmann, editor, *Fast Software Encryption – FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag, 1996.

[Dob96b] H. Dobbertin. Cryptanalysis of MD5 compress. In Rump Session of *Advances in Cryptology – EUROCRYPT 1996*, 1996. ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf.

[Dob97] H. Dobbertin. RIPEMD with Two-Round Compress Function is Not Collision-Free. In volume 10(1) of *Journal of Cryptology*, pages 51–70. Springer-Verlag, 1997.

[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.

[DP07] O. Dunkelman and B. Preneel. Generalizing the Herding Attack to Concatenated Hashing Schemes. In Proceedings of *ECRYPT Hash Workshop*, 2007.

[FLN07] P-A. Fouque, G. Leurent and P.Q. Nguyen. Full Key-Recovery Attacks on HMAC / NMAC-MD4 and NMAC-MD5. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 13–30. Springer-Verlag, 2007.

[GH03] H. Gilbert and H. Handschuh. Security Analysis of SHA-256 and Sisters. In M. Matsui and R.J. Zuccherato, editors, *Selected Areas in Cryptography – SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer-Verlag, 2003.

[GGH96] O. Goldreich, S. Goldwasser and S. Halevi. Collision-Free Hashing from Lattice Problems. In volume 42(3) of *Electronic Colloquium on Computational Complexity (ECCC)*, 1996.

[GLP08] M. Gorski, S. Lucks and T. Peyrin. Slide Attacks on Hash Functions. To appear in J. Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[HN00] H. Handschuh and D. Naccache. SHACAL. *Submission to the NESSIE project*, 2000. http://www.cryptonessie.org.

[HN02] H. Handschuh and D. Naccache. SHACAL : A Family of Block Ciphers. *Submission to the NESSIE project*, 2002. http://www.cryptonessie.org.

[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*, CE-15, pages 75–78, 2003.

[Hir04] S. Hirose. Provably Secure Double-block-length Hash Functions in a Black-box Model. In C. Park and S. Chee, editors, *Information Security and Cryptology – ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 330–342. Springer-Verlag, 2004.

[Hir06] S. Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In M.J.B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer-Verlag, 2006.

[HS06] J.J. Hoch and A. Shamir. Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. In M.J.B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 2006.

[HCS05] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon and S. Chee. A New Dedicated 256-Bit Hash Function : FORK-256. In Proceedings of *First NIST Cryptographic Hash Workshop*, 2005. http://csrc.nist.gov/groups/ST/hash/first_workshop.html.

[HCS06] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon and S. Chee. A New Dedicated 256-Bit Hash Function : FORK-256. In M.J.B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 195–209. Springer-Verlag, 2006.

[HCS07] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon and S. Chee. New FORK-256. ePrint archive, 2007. http://eprint.iacr.org/2007/185.pdf.

[IMP08] 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.

[Jou04] A. Joux. Multi-collisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 2004.

[JP07a] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 2007.

[JP07b] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Proceedings of *ECRYPT Hash Workshop*, 2007. http://events.iaik.tugraz.at/HashWorkshop07/program.html.

[JP07c] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. Presentation during *Advances in Cryptology – CRYPTO 2007*, August 2007.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KP00] P. Kasselman and W. Penzhorn. Cryptanalysis of reduced version of HAVAL. In volume 36(1) of *6th Electronics letters*, pages 30–31. 2000.

[KK06] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer-Verlag, 2006.

[KKS00] J. Kelsey, T. Kohno and B. Schneier. Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent. In B. Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 75–93. Springer-Verlag, 2000.

[KS05] J. Kelsey and B. Schneier. Second Preimages on n-bit Hash Functions for Much Less Than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer-Verlag, 2005.

[KBP06] J. Kim, A. Biryukov, B. Preneel and S. Hong. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract). In R. De Prisco and M. Yung, editors, *Security and Cryptography for Networks – SCN 2006*, volume 4116 of *Lecture Notes in Computer Science*, pages 242–256. Springer-Verlag, 2006.

[Kli06] V. Klima. Tunnels in Hash Functions : MD5 Collisions Within a Minute. ePrint archive, 2006. http://eprint.iacr.org/2006/105.pdf.

[KS02] A. Klimov and A. Shamir. A New Class of Invertible Mappings. In B. Kaliski, C. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470–483. Springer-Verlag, 2002.

[KS04] A. Klimov and A. Shamir. Cryptographic Applications of T-functions. In M. Matsui and R.J. Zuccherato, editors, *Selected Areas in Cryptography – SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 2003.

[Knu94] L.R. Knudsen. Truncated and Higher Order Differentials. In B. Preneel, editor, *Fast Software Encryption – FSE 1994*, in volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 1995.

[Knu05] L.R. Knudsen. SMASH - A Cryptographic Hash Function. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, 2005.

[Knu08] L.R. Knudsen. Hash functions and SHA-3. Invited Talk of *Fast Software Encryption – FSE 2008*, 2008. http://fse2008.epfl.ch/docs/slides/day_1_sess_2/Knudsen-FSE2008.pdf.

[KL94] L.R. Knudsen and X. Lai. New Attacks on All Double Block Length Hash Functions of Hash Rate 1, Including the Parallel-DM. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 410–418. Springer-Verlag, 1994.

[KM05] L.R. Knudsen and F. Muller. Some Attacks Against a Double Length Hash Proposal. In B. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 462–473. Springer-Verlag, 2005.

[KP96] L.R. Knudsen and B. Preneel. Hash Functions Based on Block Ciphers and Quaternary Codes. In K. Kim and T. Matsumoto, editors, *Advances in Cryptology – ASIACRYPT 1996*, volume 1163 of *Lecture Notes in Computer Science*, pages 77–90. Springer-Verlag, 1996.

[KP97] L.R. Knudsen and B. Preneel. Fast and Secure Hashing Based on Codes. In B.S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 485–498. Springer-Verlag, 1997.

[KP02] L.R. Knudsen and B. Preneel. Construction of Secure and Fast Hash Functions Using Nonbinary Error-Correcting Codes. In volume 48(9) of *IEEE Transactions on Information Theory*, pages 2524–2539, 2002.

[KRT07] L.R. Knudsen, C. Rechberger and S.S. Thomsen. Grindahl - A family of hash functions. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, 2007.

[Kho08] D. Khovratovich. Cryptanalysis of hash functions with structures. In *ECRYPT Hash Workshop*, 2008. http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Khovratovich.pdf.

[LM92] X. Lai and J.L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT 1992*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer-Verlag, 1992.

[LWH93] X. Lai, C. Waldvogel, W. Hohl, and T. Meier. Security of Iterated Hash Functions Based on Block Ciphers. In D.R. Stinson, editor, *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes in Computer Science*, pages 379–390. Springer-Verlag, 1993.

[LW05] A.K. Lenstra and B. de Weger. On the Possibility of Constructing Meaningful Hash Collisions for Public Keys. In C. Boyd and J.M.G. Nieto, editors, *Information Security and Privacy – ACISP 2005*, in volume 3574 of *Lecture Notes in Computer Science*, pages 267–279. Springer-Verlag, 2005.

[Leu07] G. Leurent. Message Freedom in MD4 and MD5 Collisions : Application to APOP. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in 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*, to appear. Springer-Verlag, 2008.

[LL05] J. Liang and X. Lai. Improved Collision Attack on Hash Function MD5. ePrint archive, 2005. http://eprint.iacr.org/2005/425.pdf.

[LWD04] H. Lipmaa, J. Wallén and P. Dumas. On the Additive Differential Probability of Exclusive-Or. In B.K. Roy and W. Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 2004.

[Luc05] S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In B.K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer-Verlag, 2005.

[MP08] S. Manuel and T. Peyrin. Collisions on SHA-0 in one hour. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[MCP06] K. Matusiewicz, S. Contini and J. Pieprzyk. Weaknesses of the FORK-256 compression function. ePrint archive, 2006. http://eprint.iacr.org/2006/317.pdf.

[MPB07] K. Matusiewicz, T. Peyrin, O. Billet, S. Contini and J. Pieprzyk. Cryptanalysis of FORK-256. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, 2007.

[MRH04] U.M. Maurer, R. Renner and C. Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In M. Naor, editor, *Theory of Cryptography – TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer-Verlag, 2004.

[MT07] U.M. Maurer and S. Tessaro. Domain Extension of Public Random Functions : Beyond the Birthday Barrier. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 187–204. Springer-Verlag, 2007.

[MLP07] F. Mendel, J. Lano and B. Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In M. Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2007.

[MPR06] F. Mendel, N. Pramstaller, C. Rechberger and V. Rijmen. Analysis of Step-Reduced SHA-256. In M.J.B. Robshaw, editor, *Fast Software Encryption – CFSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer-Verlag, 2006.

[MRR07] F. Mendel, C. Rechberger and V. Rijmen. Update on SHA-1. In Rump Session of *Advances in Cryptology – CRYPTO 2007*, August 2007.

[MR07] F. Mendel and V. Rijmen. Cryptanalysis of the Tiger Hash Function. In C-S. Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 536–550. Springer-Verlag, 2007.

[Handbook] A.J. Menezes, S.A. Vanstone, and P.C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[Mer89] R.C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1989.

[Mul06p] F. Muller. Personal communication, 2006.

[MP05] F. Muller and T. Peyrin. Linear Cryptanalysis of the TSC Family of Stream Ciphers. In B.K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 373-394. Springer-Verlag, 2005.

[MP06] F. Muller and T. Peyrin. Cryptanalysis of T-Function-Based Hash Functions. In M.S. Rhee and B. Lee, editors, *Information Security and Cryptology – ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 267–285. Springer-Verlag, 2006.

[NSS06] Y. Naito, Y. Sasaki, T. Shimoyama, J. Yajima, N. Kunihiro and K. Ohta. Improved Collision Search for SHA-0. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2006.

[NLS05] M. Nandi, W. Lee, K. Sakurai, and S. Lee. Security Analysis of a 2/3-rate Double Length Compression Function in Black-box Model. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 243–254. Springer-Verlag, 2005.

[N-sha0] National Institute of Standards and Technology. FIPS 180 : Secure Hash Standard, May 1993. http://csrc.nist.gov.

[N-sha1] National Institute of Standards and Technology. FIPS 180-1 : Secure Hash Standard, April 1995. http://csrc.nist.gov.

[N-sha2] National Institute of Standards and Technology. FIPS 180-2 : Secure Hash Standard, August 2002. http://csrc.nist.gov.

[N-sha2b] National Institute of Standards and Technology. FIPS 180-2 : Secure Hash Standard - Change notice 1, February 2004. http://csrc.nist.gov.

[N-sha3] National Institute of Standards and Technology. FIPS 180-3 : Secure Hash Standard http://csrc.nist.gov.

[N-aes] National Institute of Standards and Technology. FIPS 197 : Advanced Encryption Standard, November 2001. http://csrc.nist.gov.

[NB08] I. Nikolic and A. Biryukov. Collisions for step-reduced SHA-256. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[OpenSLL] OPENSSL. The Open Source toolkit for SSL/TLS, 2007. http://www.openssl.org/source/.

[PSC02] S. Park, S.H. Sung, S. Chee and Jongin Lim. On the Security of Reduced Versions of 3-Pass HAVAL. In L.M. Batten and J. Seberry, editors, *Information Security and Privacy – ACISP 2002*, in volume 2384 of *Lecture Notes in Computer Science*, pages 406–419. Springer-Verlag, 2002.

[Pey07] T. Peyrin. Cryptanalysis of Grindahl. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer-Verlag, 2007.

[Pey08] T. Peyrin. Security Analysis of Extended Sponge Functions. In *ECRYPT Hash Workshop*, 2008. http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Peyrin.pdf.

[PGM06] T. Peyrin, H. Gilbert, F. Muller and M.J.B. Robshaw. Combining Compression Functions and Block Cipher-Based Hash Functions. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 315–331. Springer-Verlag, 2006.

[PV05] T. Peyrin and S. Vaudenay. The Pairing Problem with User Interaction. In R. Sasaki, S. Qing, E. Okamoto and H. Yoshiura, editors, *International Conference on Information Security – SEC 2005*, pages 251–266. Springer-Verlag, 2005.

[PRR05] N. Pramstaller, C. Rechberger and V. Rijmen. Breaking a New Hash Function Design Strategy Called SMASH. In B. Preneel and S.E. Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 233–244. Springer-Verlag, 2005.

[Pre93] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.

[PBG89] B. Preneel, A. Bosselaers, R. Govaerts, and J. Vandewalle. Collision-free Hash Functions Based on Block Cipher Algorithms. In *Proceedings 1989 International Carnahan Conference on Security Technology*, pages 203–210. IEEE, 1989. IEEE catalog number 89CH2774-8.

[PGV93] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers : A Synthetic Approach. In D.R. Stinson, editor, *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer-Verlag, 1993.

[QD89] J.J. Quisquater and J.P. Delescaille. How Easy is Collision Search. New Results and Applications to DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 408–413. Springer-Verlag, 1989.

[QG89] J.-J. Quisquater and M. Girault. 2n-bit Hash-functions Using n-bit Symmetric Block Cipher Algorithms. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 102–109. Springer-Verlag, 1989.

[RIPE95] RIPE, Integrity Primitives for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040), In A. Bosselaers and B. Preneel, editors, volume 1007 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[RFCmd4] Ronald L. Rivest. RFC 1320 : The MD4 Message-Digest Algorithm, April 1992. http://www.ietf.org/rfc/rfc1320.txt.

[RFCmd5] Ronald L. Rivest. RFC 1321 : The MD5 Message-Digest Algorithm, April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[RS04] P. Rogaway and T. Shrimpton. Cryptographic Hash-Function Basics : Definitions, Implications, and Separations for Preimage Resistance, Seconde-Preimage Resistance, and Collision Resistance. In B.K. Roy and W. Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer-Verlag, 2004.

[Rog06] P. Rogaway. Formalizing Human Ignorance. In P.Q. Nguyen, editor, *Progress in Cryptology – VIETCRYPT 2006*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer-Verlag, 2006.

[Saa06] M.J.O. Saarinen. Security of VSH in the Real World. In R. Barua and T. Lange, editors, *Progress in Cryptology – INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 95–103. Springer-Verlag, 2006.

[Saa07a] M.J.O. Saarinen. A Meet-in-the-Middle Collision Attack Against the New FORK-256. In K. Srinathan, C.P. Rangan and M. Yung, editors, *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 10–17. Springer-Verlag, 2007.

[Saa07b] M.J.O. Saarinen. Linearization Attacks Against Syndrome Based Hashes. In K. Srinathan, C.P. Rangan and M. Yung, editors, *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 1–9. Springer-Verlag, 2007.

[SS08] S.K. Sanadhya and P. Sarkar. Attacking Reduced Round SHA-256. ePrint archive, 2008. http://eprint.iacr.org/2008/142.pdf.

[Saphir] Projet RNRT SAPHIR : Sécurité et Analyse des Primitives de Hachage Innovantes et Récentes. http://www.crypto-hash.fr.

[SWO07] Y. Sasaki, L. Wang, K. Ohta and N. Kunihiro. New Message Difference for MD4. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in 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://eprint.iacr.org/2007/101.pdf.

[SP07] Y. Seurin and T. Peyrin. Security Analysis of Constructions Combining FIL Random Oracles. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, 2007.

[Sha08] A. Shamir. SQUASH - a New MAC With Provable Security Properties for Highly Constrained Devices Such As RFID Tags. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[Sha49] C. Shannon. Communication Theory of Secrecy Systems. In volume 28(4) of *Bell System Technical Journal*, pages 656–715, 1949.

[SLW07a] M. Stevens, A.K. Lenstra and B. de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In M. Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2007.

[SLW07b] 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/.

[SKP07] M. Sugita, M. Kawazoe, L. Perret and H. Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer-Verlag, 2007.

[VW99] P.C. Van Oorschot and M.J. Wiener. Parallel Collision Search with Cryptanalytic Applications. In volume 12(1) of *Journal of Cryptology*, pages 1–28. Springer-Verlag, 1999.

[VBP03] 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.

[Wag99] D. Wagner. The Boomerang Attack. In L.R. Knudsen, editor, *Fast Software Encryption – FSE 1999*, in volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer-Verlag, 1999.

[Wag02] D. Wagner. A Generalized Birthday Problem. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 2002.

[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*, to appear. Springer-Verlag, 2008.

[WFL04] X. Wang, D. Feng, X. Lai and H. Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. ePrint archive, 2004. http://eprint.iacr.org/2004/199.pdf.

[WLF05] X. Wang, X. Lai, D. Feng, H. Chen and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.

[WYY05a] X. Wang, A.C. Yao, and F. Yao. Cryptanalysis on SHA-1. In Proceedings of *NIST Cryptographic Hash Workshop*, 2005.

[WYY05b] X. Wang, Y.L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer-Verlag, 2005.

[WYY05c] X. Wang, Y.L. Yin, and H. Yu. New Collision Search for SHA-1. In Rump Session of *Advances in Cryptology – CRYPTO 2005*, August 2005.

[WY05] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2005.

[WYY05d] X. Wang, H. Yu and Y.L. Yin. Efficient Collision Search Attacks on SHA-0. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2005.

[Wat08] D. Watanabe. A note on the security proof of Knudsen-Preneel construction of a hash function. http://csrc.nist.gov/groups/ST/hash/documents/WATANABE_kp_attack.pdf.

[Win84] R.S. Winternitz. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, *Lecture Notes in Computer Science*, pages 88–90, 1984.

[YIN08a] J. Yajima, T. Iwasaki, Y. Naito, Y. Sasaki, T. Shimoyama, N. Kunihiro and K. Ohta. A Strict Evaluation Method on the Number of Conditions for the SHA-1 Collision Search. In M. Abe and V. Gligor, editors, *ACM Symposium on Information, Computer and Communications Security – ASIACCS 2008*, *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[YIN08b] J. Yajima, T. Iwasaki, Y. Naito, Y. Sasaki, T. Shimoyama, T. Peyrin, N. Kunihiro and K. Ohta. A Strict Evaluation Method on the Number of Conditions for SHA-1 Collision Search. To appear in volume E92-A (1) of *IEICE Trans. Fundamentals*, January 2009.

[YSN07] J. Yajima, Y. Sasaki, Y. Naito, T. Iwasaki, T. Shimoyama, N. Kunihiro and K. Ohta. A New Strategy for Finding a Differential Path of SHA-1. In J. Pieprzyk, H. Ghodosi and E. Dawson, editors, *Information Security and Privacy – ACISP 2007*, in volume 4586 of *Lecture Notes in Computer Science*, pages 45–58. Springer-Verlag, 2007.

[YS05] J. Yajima and T. Shimoyama. Wang's sufficient conditions of MD5 are not sufficient. ePrint archive, 2005. http://eprint.iacr.org/2005/263.pdf.

[YBC04] 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.

[YW007] H. Yoshida, D. Watanabe, K. Okeya, J. Kitahara, H. Wu, O. Küçük and B. Preneel. MAME : A Compression Function with Reduced Hardware Requirements. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 148–165. Springer-Verlag, 2007.

[YWY06] 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*, in volume 4047 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2006.

[Yuv79] G. Yuval. How to Swindle Rabin. In volume 3(3) of *Cryptologia*, pages 187–189, 1979.

[ZPS92] Y. Zheng, J. Pieprzyk and J. Seberry. HAVAL - A One-Way Hashing Algorithm with Variable Length of Output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology – ASIACRYPT 1992*, volume 718 of *Lecture Notes in Computer Science*, pages 83–104. Springer-Verlag, 1992.

Bibliographie personelle

[Indocrypt-04] G. Avoine, J. Monnerat and T. Peyrin. Advances in Alternative Non-adjacent Form Representations. In A. Canteaut and K. Viswanathan, editors, *Progress in Cryptology – INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2004.

[ACISP-07] O. Billet, M.J.B. Robshaw and T. Peyrin. On Building Hash Functions From Multivariate Quadratic Equations. In J. Pieprzyk, H. Ghodosi and E. Dawson, editors, *Information Security and Privacy – ACISP 2007*, in volume 4586 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 2007.

[HashWorkshop-07a] E. Bresson, B. Chevallier-Mames, C. Clavier, B. Debraize, P.A. Fouque, L. Goubin, A. Gouget, G. Leurent, P.Q. Nguyen, P. Paillier, T. Peyrin and S. Zimmer. Revisiting Security Relations Between Signature Schemes and their Inner Hash Functions. In Proceedings of *ECRYPT Hash Workshop*, 2007. http://events.iaik.tugraz.at/HashWorkshop07/program.html.

[Asiacrypt-08] M. Gorski, S. Lucks and T. Peyrin. Slide Attacks on Hash Functions. To appear in J. Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[Crypto-07] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 2007.

[HashWorkshop-07b] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Proceedings of *ECRYPT Hash Workshop*, 2007. http://events.iaik.tugraz.at/HashWorkshop07/program.html.

[FSE-08] S. Manuel and T. Peyrin. Collisions on SHA-0 in one hour. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, to appear. Springer-Verlag, 2008.

[FSE-07a] K. Matusiewicz, T. Peyrin, O. Billet, S. Contini and J. Pieprzyk. Cryptanalysis of FORK-256. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, 2007.

[Asiacrypt-05] F. Muller and T. Peyrin. Linear Cryptanalysis of the TSC Family of Stream Ciphers. In B.K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 373-394. Springer-Verlag, 2005.

[ICISC-06] F. Muller and T. Peyrin. Cryptanalysis of T-Function-Based Hash Functions. In M.S. Rhee and B. Lee, editors, *Information Security and Cryptology – ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 267–285. Springer-Verlag, 2006.

[Asiacrypt-07] T. Peyrin. Cryptanalysis of Grindahl. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer-Verlag, 2007.

[Asiacrypt-06] T. Peyrin, H. Gilbert, F. Muller and M.J.B. Robshaw. Combining Compression Functions and Block Cipher-Based Hash Functions. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 315–331. Springer-Verlag, 2006.

[SEC-05] T. Peyrin and S. Vaudenay. The Pairing Problem with User Interaction. In R. Sasaki, S. Qing, E. Okamoto and H. Yoshiura, editors, *International Conference on Information Security – SEC 2005*, pages 251–266. Springer-Verlag, 2005.

[FSE-07b] Y. Seurin and T. Peyrin. Security Analysis of Constructions Combining FIL Random Oracles. In A. Biryukov, editor, *Fast Software Encryption – FSE 2007*, in volume 4593 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, 2007.

[IEICE-09] J. Yajima, T. Iwasaki, Y. Naito, Y. Sasaki, T. Shimoyama, T. Peyrin, N. Kunihiro and K. Ohta. A Strict Evaluation Method on the Number of Conditions for SHA-1 Collision Search. To appear in volume E92-A (1) of *IEICE Trans. Fundamentals*, January 2009.

ANNEXE A

Spécification des fonctions de compression de la famille MD-SHA

Dans cette section de l'appendice, nous donnons tous les éléments nécessaires à la spécification complète des fonctions de compression de la famille MD-SHA. Tout d'abord, pour chaque schéma, nous donnons les paramètres associés à la fonction de compression (dans le cas des membres de la famille RIPEMD, ces paramètres correspondent à une seule branche) :

- n le nombre de bits de sortie,
- m le nombre de blocs de message traités par itération,
- w la taille en bits de chaque bloc,
- r le nombre de registres internes,
- t le nombre de tours,
- u le nombre d'étapes par tour,
- s le nombre total d'étapes.

L'expansion de message est exprimée par la définition des permutations π_j dans le cas d'une permutation des blocs de message à chaque tour, et par la formule de récurrence dans le cas d'une expansion de message récursive. Nous donnons ensuite les valeurs d'initialisation pour le premier bloc traité lors du premier appel à la fonction de compression durant le hachage itératif de Merkle-Damgård, puis la formule de mise à jour des registres cibles avec toutes les constantes ou fonctions nécessaires à l'implantation de la fonction. A_{i+1} représente le registre mis à jour à l'étape $i, 0 \leqslant i \leqslant s-1$ (dans le cas des membres de la famille RIPEMD, on distingue les registres cibles A_i^L de la branche de gauche des registres cibles A_i^R de celle de droite). Par souci de clarté pour la suite, nous introduisons maintenant certaines fonctions booléennes qui vont être souvent utilisées en tant que fonction Φ_j :

$$\begin{aligned} \text{XOR}(x,y,z) &:= x \oplus y \oplus z \\ \text{MAJ}(x,y,z) &:= xy \oplus xz \oplus yz \\ \text{IF}(x,y,z) &:= xy \oplus \bar{x}z = xy \oplus xz \oplus z \\ \text{ONX}(x,y,z) &:= (x \vee \bar{y}) \oplus z = xy \oplus y \oplus z \oplus 1 \end{aligned}$$

où x, y et z sont des mots de w bits, ces fonctions traitant les mots d'entrée bit à bit.

Nous donnons enfin l'initialisation de l'état interne par la variable de chaînage d'entrée $H = h_0, \ldots, h_{r-1}$, et le calcul final de la variable de chaînage de sortie $H' = h'_0, \ldots, h'_{r-1}$.

A.1 MD4 [RFCmd4]

Paramètres :

$n = 128$ $m = 16$ $w = 32$ $r = 4$ $t = 3$ $u = 16$ $s = 48$

Expansion de message :

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

$\pi_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 1$	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
Tour $j = 2$	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

Valeurs d'initialisation :

A_{-3} =0x67452301 A_{-2} =0x10325476 A_{-1} =0x98badcfe A_0 =0xefcdab89

Transformation d'étape :

$$A_{i+1} = (A_{i-3} + \Phi_j(A_i, A_{i-1}, A_{i-2}) + W_i + K_j)^{\lll s_i}.$$

Tour j	$\Phi_j(x, y, z)$	K_j
0	IF(x, y, z)	0x00000000
1	MAJ(x, y, z)	0x5a827999
2	XOR(x, y, z)	0x6ed9eba1

Tour j	$S_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	3	7	11	19	3	7	11	19	3	7	11	19	3	7	11	19
1	3	5	9	13	3	5	9	13	3	5	9	13	3	5	9	13
2	3	9	11	15	3	9	11	15	3	9	11	15	3	9	11	15

Entrée :

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

Sortie :

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

A.2 MD5 [RFCmd5]

Paramètres :

$n = 128$ $m = 16$ $w = 32$ $r = 4$ $t = 4$ $u = 16$ $s = 64$

Expansion de message :

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

$\pi_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 1$	1	6	11	0	5	10	15	4	9	14	3	8	13	2	7	12
Tour $j = 2$	5	8	11	14	1	4	7	10	13	0	3	6	9	12	15	2
Tour $j = 3$	0	7	14	5	12	3	10	1	8	15	6	13	4	11	2	9

Valeurs d'initialisation :

A_{-3} =0x67452301 A_{-2} =0x10325476 A_{-1} =0x98badcfe A_0 =0xefcdab89

Transformation d'étape :

$$A_{i+1} = A_i + (A_{i-3} + \Phi_j(A_i, A_{i-1}, A_{i-2}) + W_i + K_i)^{\lll s_i}.$$

Tour j	$\Phi_j(x,y,z)$	$S_{16\times j+k}$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	IF(x,y,z)	7	12	17	22	7	12	17	22	7	12	17	22	7	12	17	22
1	IF(z,x,y)	5	9	14	20	5	9	14	20	5	9	14	20	5	9	14	20
2	XOR(x,y,z)	4	11	16	23	4	11	16	23	4	11	16	23	4	11	16	23
3	ONX(x,z,y)	6	10	15	21	6	10	15	21	6	10	15	21	6	10	15	21

Valeurs K_i avec $i =$ (ligne $\times$ 4) + colonne			
0xd76aa478	0xe8c7b756	0x242070db	0xc1bdceee
0xf57c0faf	0x4787c62a	0xa8304613	0xfd469501
0x698098d8	0x8b44f7af	0xffff5bb1	0x895cd7be
0x6b901122	0xfd987193	0xa679438e	0x49b40821
0xf61e2562	0xc040b340	0x265e5a51	0xe9b6c7aa
0xd62f105d	0x02441453	0xd8a1e681	0xe7d3fbc8
0x21e1cde6	0xc33707d6	0xf4d50d87	0x455a14ed
0xa9e3e905	0xfcefa3f8	0x676f02d9	0x8d2a4c8a
0xfffa3942	0x8771f681	0x6d9d6122	0xfde5380c
0xa4beea44	0x4bdecfa9	0xf6bb4b60	0xbebfbc70
0x289b7ec6	0xeaa127fa	0xd4ef3085	0x04881d05
0xd9d4d039	0xe6db99e5	0x1fa27cf8	0xc4ac5665
0xf4292244	0x432aff97	0xab9423a7	0xfc93a039
0x655b59c3	0x8f0ccc92	0xffeff47d	0x85845dd1
0x6fa87e4f	0xfe2ce6e0	0xa3014314	0x4e0811a1
0xf7537e82	0xbd3af235	0x2ad7d2bb	0xeb86d391

Entrée :

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

Sortie :

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

A.3 RIPEMD-0 [RIPE95]

Paramètres par branche :

$n = 128$ $m = 16$ $w = 32$ $r = 4$ $t = 3$ $u = 16$ $s = 48$

Expansion de message :

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

$\pi_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 1$	7	4	13	1	10	6	15	3	12	0	9	5	14	2	11	8
Tour $j = 2$	3	10	2	4	9	15	8	1	14	7	0	6	11	13	5	12

Valeurs d'initialisation :

$$A^L_{-3} = A^R_{-3} = \texttt{0x67452301} \quad A^L_{-2} = A^R_{-2} = \texttt{0xefcdab89}$$
$$A^L_{-1} = A^R_{-1} = \texttt{0x98badcfe} \quad A^L_{0} = A^R_{0} = \texttt{0x10325476}$$

Transformation d'étape :

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

Tour j	$\Phi_j(x, y, z)$	K^L_j	K^R_j
0	IF(x, y, z)	`0x00000000`	`0x50a28be6`
1	MAJ(x, y, z)	`0x5a827999`	`0x00000000`
2	XOR(x, y, z)	`0x6ed9eba1`	`0x5c4dd124`

Tour j	$S_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	7	11	13	12
2	11	13	14	7	14	9	13	15	6	8	13	6	12	5	7	5

Entrée :

$$A^L_{-3} = h_0 \quad A^L_{-2} = h_1 \quad A^L_{-1} = h_2 \quad A^L_{0} = h_3$$
$$A^R_{-3} = h_0 \quad A^R_{-2} = h_1 \quad A^R_{-1} = h_2 \quad A^R_{0} = h_3$$

Sortie :

$$h'_0 = A^L_{47} + A^R_{48} + A_{-2} \quad h'_1 = A^L_{48} + A^R_{45} + A_{-1}$$
$$h'_2 = A^L_{45} + A^R_{46} + A_{0} \quad h'_3 = A^L_{46} + A^R_{47} + A_{-3}$$

A.4 RIPEMD-128 [DBP96]

Paramètres par branche :

$n = 128$ $m = 16$ $w = 32$ $r = 4$ $t = 4$ $u = 16$ $s = 64$

Expansion de message :

$$\begin{cases} W^L_{j\times 16+k} = M_{\pi^L_j(k)} \\ W^R_{j\times 16+k} = M_{\pi^R_j(k)} \end{cases}$$

$\pi^L_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 1$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
Tour $j = 2$	3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
Tour $j = 3$	1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2

$\pi^R_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
Tour $j = 1$	6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
Tour $j = 2$	15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
Tour $j = 3$	8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14

Valeurs d'initialisation :

$$A^L_{-3} = A^R_{-3} = \texttt{0x67452301} \quad A^L_{-2} = A^R_{-2} = \texttt{0xefcdab89}$$
$$A^L_{-1} = A^R_{-1} = \texttt{0x98badcfe} \quad A^L_0 = A^R_0 = \texttt{0x10325476}$$

Transformation d'étape :

$$\begin{aligned} A^L_{i+1} &= (A^L_{i-3} + \Phi^L_j(A^L_i, A^L_{i-1}, A^L_{i-2}) + W^L_i + K^L_j)^{\lll s^L_i}, \\ A^R_{i+1} &= (A^R_{i-3} + \Phi^R_j(A^R_i, A^R_{i-1}, A^R_{i-2}) + W^R_i + K^R_j)^{\lll s^R_i}. \end{aligned}$$

Tour j	$\Phi^L_j(x,y,z)$	$\Phi^R_j(x,y,z)$	K^L_j	K^R_j
0	XOR(x,y,z)	IF(z,x,y)	`0x00000000`	`0x50a28be6`
1	IF(x,y,z)	ONX(x,y,z)	`0x5a827999`	`0x5c4dd124`
2	ONX(x,y,z)	IF(x,y,z)	`0x6ed9eba1`	`0x6d703ef3`
3	IF(z,x,y)	XOR(x,y,z)	`0x8f1bbcdc`	`0x00000000`

Tour j	$S^L_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
2	11	13	6	7	14	9	13	15	14	8	13	6	5	12	7	5
3	11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12

Tour j	$S^R_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	8	9	9	11	13	15	15	5	7	7	8	11	14	14	12	6
1	9	13	15	7	12	8	9	11	7	7	12	7	6	15	13	11
2	9	7	15	11	8	6	6	14	12	13	5	14	13	13	7	5
3	15	5	8	11	14	14	6	14	6	9	12	9	12	5	15	8

Entrée :

$$A^L_{-3} = h_0 \qquad A^L_{-2} = h_1 \qquad A^L_{-1} = h_2 \qquad A^L_0 = h_3$$
$$A^R_{-3} = h_0 \qquad A^R_{-2} = h_1 \qquad A^R_{-1} = h_2 \qquad A^R_0 = h_3$$

Sortie :

$$h'_0 = A^L_{63} + A^R_{64} + A_{-2} \qquad h'_1 = A^L_{64} + A^R_{61} + A_{-1}$$
$$h'_2 = A^L_{61} + A^R_{62} + A_0 \qquad h'_3 = A^L_{62} + A^R_{63} + A_{-3}$$

A.5 RIPEMD-160 [DBP96]

Paramètres par branche :

$n = 160$ $m = 16$ $w = 32$ $r = 5$ $t = 5$ $u = 16$ $s = 80$

Expansion de message :

$$\begin{cases} W^L_{j\times 16+k} = M_{\pi^L_j(k)} \\ W^R_{j\times 16+k} = M_{\pi^R_j(k)} \end{cases}$$

$\pi^L_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 1$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
Tour $j = 2$	3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
Tour $j = 3$	1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2
Tour $j = 4$	4	0	5	9	7	12	2	10	14	1	3	8	11	6	15	13

$\pi^R_j(k)$	k-ième étape dans le tour															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tour $j = 0$	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
Tour $j = 1$	6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
Tour $j = 2$	15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
Tour $j = 3$	8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14
Tour $j = 4$	12	15	10	4	1	5	8	7	6	2	13	14	0	3	9	11

Valeurs d'initialisation :

$A^L_{-4} = A^R_{-4} =$`0xc059d148` $A^L_{-3} = A^R_{-3} =$`0x7c30f4b8`
$A^L_{-2} = A^R_{-2} =$`0x1d840c95` $A^L_{-1} = A^R_{-1} =$`0x98badcfe` $A^L_0 = A^R_0 =$`0xefcdab89`

Transformation d'étape :

$$\begin{aligned} A^L_{i+1} &= ((A^L_{i-4})^{\lll 10} + \Phi^L_j(A^L_i, A^L_{i-1}, (A^L_{i-2})^{\lll 10}) + W^L_i + K^L_j)^{\lll s^L_i} + (A^L_{i-3})^{\lll 10}, \\ A^R_{i+1} &= ((A^R_{i-4})^{\lll 10} + \Phi^R_j(A^R_i, A^R_{i-1}, (A^R_{i-2})^{\lll 10}) + W^R_i + K^R_j)^{\lll s^R_i} + (A^R_{i-3})^{\lll 10}. \end{aligned}$$

Tour j	$\Phi^L_j(x,y,z)$	$\Phi^R_j(x,y,z)$	K^L_j	K^R_j
0	XOR(x,y,z)	ONX(y,z,x)	`0x00000000`	`0x50a28be6`
1	IF(x,y,z)	IF(z,x,y)	`0x5a827999`	`0x5c4dd124`
2	ONX(x,y,z)	ONX(x,y,z)	`0x6ed9eba1`	`0x6d703ef3`
3	IF(z,x,y)	IF(x,y,z)	`0x8f1bbcdc`	`0x7a6d76e9`
4	ONX(y,z,x)	XOR(x,y,z)	`0xa953fd4e`	`0x00000000`

Tour j	$S^L_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
2	11	13	6	7	14	9	13	15	14	8	13	6	5	12	7	5
3	11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12
4	9	15	5	11	6	8	13	12	5	12	13	14	11	8	5	6

Tour j	$S^R_{16\times j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	8	9	9	11	13	15	15	5	7	7	8	11	14	14	12	6
1	9	13	15	7	12	8	9	11	7	7	12	7	6	15	13	11
2	9	7	15	11	8	6	6	14	12	13	5	14	13	13	7	5
3	15	5	8	11	14	14	6	14	6	9	12	9	12	5	15	8
4	8	5	12	9	12	5	14	6	8	13	6	5	15	13	11	11

Entrée :

$A^L_{-4} = (h_0)^{\ggg 10}$ $A^L_{-3} = (h_4)^{\ggg 10}$ $A^L_{-2} = (h_3)^{\ggg 10}$ $A^L_{-1} = h_2$ $A^L_0 = h_1$
$A^R_{-4} = (h_0)^{\ggg 10}$ $A^R_{-3} = (h_4)^{\ggg 10}$ $A^R_{-2} = (h_3)^{\ggg 10}$ $A^R_{-1} = h_2$ $A^R_0 = h_1$

Sortie :

$h'_0 = A^L_{79} + (A^R_{78})^{\lll 10} + A_0$ $h'_1 = (A^L_{78})^{\lll 10} + (A^R_{77})^{\lll 10} + A_{-1}$
$h'_2 = (A^L_{77})^{\lll 10} + (A^R_{76})^{\lll 10} + (A_{-2})^{\lll 10}$ $h'_3 = (A^L_{76})^{\lll 10} + A^R_{80} + (A_{-3})^{\lll 10}$
$h'_4 = A^L_{80} + A^R_{79} + (A_{-4})^{\lll 10}$

A.6 SHA-0 [N-sha0]

Paramètres par branche :

$n = 160$ $m = 16$ $w = 32$ $r = 5$ $t = 4$ $u = 20$ $s = 80$

Expansion de message :

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

Valeurs d'initialisation :

A_{-4} =0x0f4b87c3 A_{-3} =0x40c951d8
A_{-2} =0x62eb73fa A_{-1} =0xefcdab89 A_0 =0x67452301

Transformation d'étape :

$$A_{i+1} \quad = \quad (A_i)^{\lll 5} + \Phi_j(A_{i-1}, (A_{i-2})^{\ggg 2}, (A_{i-3})^{\ggg 2}) + (A_{i-4})^{\ggg 2} + W_i + K_j.$$

Tour j	$\Phi_j(x, y, z)$	K_j
0	IF(x, y, z)	0x5a827999
1	XOR(x, y, z)	0x6ed9eba1
2	MAJ(x, y, z)	0x8f1bbcdc
3	XOR(x, y, z)	0xca62c1d6

Entrée :

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

Sortie :

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

A.7 SHA-1 [N-sha1]

Paramètres par branche :

$n = 160$ $m = 16$ $w = 32$ $r = 5$ $t = 4$ $u = 20$ $s = 80$

Expansion de message :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})^{\lll 1}, & \text{pour } 16 \leqslant i \leqslant 79 \end{cases}$$

Valeurs d'initialisation :

$A_{-4} =$`0x0f4b87c3` $A_{-3} =$`0x40c951d8`
$A_{-2} =$`0x62eb73fa` $A_{-1} =$`0xefcdab89` $A_0 =$`0x67452301`

Transformation d'étape :

$$A_{i+1} \quad = \quad (A_i)^{\lll 5} + \Phi_j(A_{i-1}, (A_{i-2})^{\ggg 2}, (A_{i-3})^{\ggg 2}) + (A_{i-4})^{\ggg 2} + W_i + K_j.$$

Tour j	$\Phi_j(x, y, z)$	K_j
0	IF(x, y, z)	`0x5a827999`
1	XOR(x, y, z)	`0x6ed9eba1`
2	MAJ(x, y, z)	`0x8f1bbcdc`
3	XOR(x, y, z)	`0xca62c1d6`

Entrée :

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

Sortie :

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

A.8 SHA-256 [N-sha2, N-sha2b]

Paramètres par branche :

$n = 256$ $m = 16$ $w = 32$ $r = 8$ $t = 1$ $u = 64$ $s = 64$

Expansion de message :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{pour } 16 \leqslant i \leqslant 63 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_0(x) = (x^{\ggg 7}) \oplus (x^{\ggg 18}) \oplus (x^{\gg 3}) \\ \sigma_1(x) = (x^{\ggg 17}) \oplus (x^{\ggg 19}) \oplus (x^{\gg 10}) \end{cases}$$

Valeurs d'initialisation :

$$\begin{array}{rr} A_{-3} = \texttt{0xa54ff53a} & A_{-2} = \texttt{0x3c6ef372} \\ A_{-1} = \texttt{0xbb67ae85} & A_0 = \texttt{0x6a09e667} \\ B_{-3} = \texttt{0x5be0cd19} & B_{-2} = \texttt{0x1f83d9ab} \\ B_{-1} = \texttt{0x62eb73fa} & B_0 = \texttt{0x510e527f} \end{array}$$

Transformation d'étape :

$$\begin{cases} B_{i+1} = T + A_{i-3} \\ A_{i+1} = T + \Sigma_0(A_i) + \text{MAJ}(A_i, A_{i-1}, A_{i-2}) \end{cases}$$

avec $T = B_{i-3} + \Sigma_1(B_i) + \text{IF}(B_i, B_{i-1}, B_{i-2}) + W_i + K_i$, et

$$\begin{cases} \Sigma_0(x) = (x^{\ggg 2}) \oplus (x^{\ggg 13}) \oplus (x^{\gg 22}) \\ \Sigma_1(x) = (x^{\ggg 6}) \oplus (x^{\ggg 11}) \oplus (x^{\gg 25}) \end{cases}$$

Valeurs K_i avec $i = (\text{ligne} \times 4) + \text{colonne}$			
0x428a2f98	0x71374491	0xb5c0fbcf	0xe9b5dba5
0x3956c25b	0x59f111f1	0x923f82a4	0xab1c5ed5
0xd807aa98	0x12835b01	0x243185be	0x550c7dc3
0x72be5d74	0x80deb1fe	0x9bdc06a7	0xc19bf174
0xe49b69c1	0xefbe4786	0x0fc19dc6	0x240ca1cc
0x2de92c6f	0x4a7484aa	0x5cb0a9dc	0x76f988da
0x983e5152	0xa831c66d	0xb00327c8	0xbf597fc7
0xc6e00bf3	0xd5a79147	0x06ca6351	0x14292967
0x27b70a85	0x2e1b2138	0x4d2c6dfc	0x53380d13
0x650a7354	0x766a0abb	0x81c2c92e	0x92722c85
0xa2bfe8a1	0xa81a664b	0xc24b8b70	0xc76c51a3
0xd192e819	0xd6990624	0xf40e3585	0x106aa070
0x19a4c116	0x1e376c08	0x2748774c	0x34b0bcb5
0x391c0cb3	0x4ed8aa4a	0x5b9cca4f	0x682e6ff3
0x748f82ee	0x78a5636f	0x84c87814	0x8cc70208
0x90befffa	0xa4506ceb	0xbef9a3f7	0xc67178f2

Entrée :

$$\begin{array}{llll} A_{-3} = h_3 & A_{-2} = h_2 & A_{-1} = h_1 & A_0 = h_0 \\ B_{-3} = h_7 & B_{-2} = h_6 & B_{-1} = h_5 & B_0 = h_4 \end{array}$$

Sortie :

$$\begin{array}{llll} h'_0 = A_{64} + A_0 & h'_1 = A_{63} + A_{-1} & h'_2 = A_{62} + A_{-2} & h'_3 = A_{61} + A_{-3} \\ h'_4 = B_{64} + B_0 & h'_5 = B_{63} + B_{-1} & h'_6 = B_{62} + B_{-2} & h'_7 = B_{61} + B_{-3} \end{array}$$

A.9 SHA-512 [N-sha2]

Paramètres par branche :

$n = 512$ $m = 16$ $w = 64$ $r = 8$ $t = 1$ $u = 80$ $s = 80$

Expansion de message :

$$W_i = \begin{cases} M_i, & \text{pour } 0 \leqslant i \leqslant 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & \text{pour } 16 \leqslant i \leqslant 79 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_0(x) = (x^{\ggg 1}) \oplus (x^{\ggg 8}) \oplus (x^{\gg 7}) \\ \sigma_1(x) = (x^{\ggg 19}) \oplus (x^{\ggg 61}) \oplus (x^{\gg 6}) \end{cases}$$

Valeurs d'initialisation :

$A_{-3} =$0xa54ff53a5f1d36f1 $A_{-2} =$0x3c6ef372fe94f82b
$A_{-1} =$0xbb67ae8584caa73b $A_0 =$0x6a09e667f3bcc908
$B_{-3} =$0x5be0cd19137e2179 $B_{-2} =$0x1f83d9abfb41bd6b
$B_{-1} =$0x9b05688c2b3e6c1f $B_0 =$0x510e527fade682d1

Transformation d'étape :

$$\begin{cases} B_{i+1} = T + A_{i-3} \\ A_{i+1} = T + \Sigma_0(A_i) + \text{MAJ}(A_i, A_{i-1}, A_{i-2}) \end{cases}$$

avec $T = B_{i-3} + \Sigma_1(B_i) + \text{IF}(B_i, B_{i-1}, B_{i-2}) + W_i + K_i$, et

$$\begin{cases} \Sigma_0(x) = (x^{\ggg 28}) \oplus (x^{\ggg 34}) \oplus (x^{\gg 39}) \\ \Sigma_1(x) = (x^{\ggg 14}) \oplus (x^{\ggg 18}) \oplus (x^{\gg 41}) \end{cases}$$

Valeurs K_i avec $i = (\text{ligne} \times 4) + \text{colonne}$			
0x428a2f98d728ae22	0x7137449123ef65cd	0xb5c0fbcfec4d3b2f	0xe9b5dba58189dbbc
0x3956c25bf348b538	0x59f111f1b605d019	0x923f82a4af194f9b	0xab1c5ed5da6d8118
0xd807aa98a3030242	0x12835b0145706fbe	0x243185be4ee4b28c	0x550c7dc3d5ffb4e2
0x72be5d74f27b896f	0x80deb1fe3b1696b1	0x9bdc06a725c71235	0xc19bf174cf692694
0xe49b69c19ef14ad2	0xefbe4786384f25e3	0x0fc19dc68b8cd5b5	0x240ca1cc77ac9c65
0x2de92c6f592b0275	0x4a7484aa6ea6e483	0x5cb0a9dcbd41fbd4	0x76f988da831153b5
0x983e5152ee66dfab	0xa831c66d2db43210	0xb00327c898fb213f	0xbf597fc7beef0ee4
0xc6e00bf33da88fc2	0xd5a79147930aa725	0x06ca6351e003826f	0x142929670a0e6e70
0x27b70a8546d22ffc	0x2e1b21385c26c926	0x4d2c6dfc5ac42aed	0x53380d139d95b3df
0x650a73548baf63de	0x766a0abb3c77b2a8	0x81c2c92e47edaee6	0x92722c851482353b
0xa2bfe8a14cf10364	0xa81a664bbc423001	0xc24b8b70d0f89791	0xc76c51a30654be30
0xd192e819d6ef5218	0xd69906245565a910	0xf40e35855771202a	0x106aa07032bbd1b8
0x19a4c116b8d2d0c8	0x1e376c085141ab53	0x2748774cdf8eeb99	0x34b0bcb5e19b48a8
0x391c0cb3c5c95a63	0x4ed8aa4ae3418acb	0x5b9cca4f7763e373	0x682e6ff3d6b2b8a3
0x748f82ee5defb2fc	0x78a5636f43172f60	0x84c87814a1f0ab72	0x8cc702081a6439ec
0x90befffa23631e28	0xa4506cebde82bde9	0xbef9a3f7b2c67915	0xc67178f2e372532b
0xca273eceea26619c	0xd186b8c721c0c207	0xeada7dd6cde0eb1e	0xf57d4f7fee6ed178
0x06f067aa72176fba	0x0a637dc5a2c898a6	0x113f9804bef90dae	0x1b710b35131c471b
0x28db77f523047d84	0x32caab7b40c72493	0x3c9ebe0a15c9bebc	0x431d67c49c100d4c
0x4cc5d4becb3e42b6	0x597f299cfc657e2a	0x5fcb6fab3ad6faec	0x6c44198c4a475817

Entrée :

$$A_{-3} = h_3 \qquad A_{-2} = h_2 \qquad A_{-1} = h_1 \qquad A_0 = h_0$$
$$B_{-3} = h_7 \qquad B_{-2} = h_6 \qquad B_{-1} = h_5 \qquad B_0 = h_4$$

Sortie :

$$h'_0 = A_{80} + A_0 \qquad h'_1 = A_{79} + A_{-1} \qquad h'_2 = A_{78} + A_{-2} \qquad h'_3 = A_{77} + A_{-3}$$
$$h'_4 = B_{76} + B_0 \qquad h'_5 = B_{75} + B_{-1} \qquad h'_6 = B_{74} + B_{-2} \qquad h'_7 = B_{73} + B_{-3}$$

ANNEXE B

Conditions totales concernant les collisions locales pour SHA

étape	type	propagation	addition	probabilité
i	introduction		$A_{i+1}^{j} = a$ et $W_i^j = a$	$1/2$
$i+1$	correction		$W_{i+1}^{j+5} = \overline{a}$	1
$i+2$ IF	corr. situation 1 (a,-,-)	$A_{i-1}^{j+2} \neq A_i^{j+2}$	$A_{i-1}^{j+2} = W_{i+2}^{j} \oplus \overline{a}$	$1/4$ ($1/2$ si $j = 31$)
	corr. situation 3 (a,p_1,-)	$A_{i-1}^{j+2} = \overline{a} \oplus p_1$		$1/2$
	corr. situation 5 (a,-,p_2)	$A_i^{j+2} = a \oplus p_2$		$1/2$
	corr. situation 7 (a,p_1,p_2)	$p_1 = p_2$	$W_{i+2}^{j} = p_2 \oplus \overline{a}$	1
$i+2$ MAJ	corr. situation 1 (a,-,-)	$A_{i-1}^{j+2} \neq A_i^{j+2}$	$W_{i+2}^{j} = \overline{a}$	$1/2$
	corr. situation 3 (a,p_1,-)	$p_1 = \overline{a}$		1
	corr. situation 5 (a,-,p_2)	$p_2 = \overline{a}$		1
	corr. situation 7 (a,p_1,p_2)		$MAJ(a, p_1, p_2) \neq W_{i+2}^{j}$	1
$i+2$ XOR	corr. situation 1 (a,-,-)		$A_{i-1}^{j+2} \oplus A_i^{j+2} = W_{i+2}^{j} \oplus \overline{a}$	$1/2$ (1 si $j = 31$)
	corr. situation 3 (a,p_1,-)			1
	corr. situation 5 (a,-,p_2)			1
	corr. situation 7 (a,p_1,p_2)		$W_{i+2}^{j} = \overline{a} \oplus p_1 \oplus p_2$	1
$i+3$ IF	corr. situation 2 (-,a,-)	$A_{i+2}^{j-2} = 1$	$W_{i+3}^{j-2} = \overline{a}$	$1/2$
	corr. situation 3 (p_1,a,-)	$A_i^{j} = \overline{a} \oplus p_1$		$1/2$
	corr. situation 6 (-,a,p_2)			0
	corr. situation 7 (p_1,a,p_2)	$p_2 = a$	$W_{i+3}^{j-2} = \overline{a}$	1
$i+3$ MAJ	corr. situation 2 (-,a,-)	$A_{i+2}^{j-2} \neq A_i^{j}$	$W_{i+3}^{j-2} = \overline{a}$	$1/2$
	corr. situation 3 (p_1,a,-)	$p_1 = \overline{a}$		1
	corr. situation 6 (-,a,p_2)	$p_2 = \overline{a}$		1
	corr. situation 7 (p_1,a,p_2)		$MAJ(p_1, a, p_2) \neq W_{i+3}^{j-2}$	1
$i+3$ XOR	corr. situation 2 (-,a,-)		$A_{i+2}^{j-2} \oplus A_i^{j} = W_{i+3}^{j-2} \oplus \overline{a}$	$1/2$ (1 si $j = 1$)
	corr. situation 3 (p_1,a,-)			1
	corr. situation 6 (-,a,p_2)			1
	corr. situation 7 (p_1,a,p_2)		$W_{i+3}^{j-2} = \overline{a} \oplus p_1 \oplus p_2$	1

TAB. B.1 – PARTIE 1 - Conditions à vérifier pour la réussite d'une collision locale pour SHA-0 ou SHA-1, avec une perturbation introduite à l'étape i et à la position j. Le signe de la perturbation est donnée par la contraite $W_i^j = a$. La première colonne désigne l'étape considerée et la deuxième colonne précise le type d'action appliqué et la situation le cas échéant. La troisième colonne (respectivement la quatrième) donne les conditions pour que la propagation (respectivement l'addition) des différences binaires signées se comporte de façon linéaire. Enfin, la dernière colonne donne la probabilité de réussite (entre parenthèses est indiquée la probabilité lorsque la position considerée est égale à 31).

étape	type	propagation	addition	probabilité
i	introduction		$A_{i+1}^j = a$ et $W_i^j = a$	$1/2$
$i+4$ IF	corr. situation 4 (-,-,a)	$A_{i+3}^{j-2} = 0$	$W_{i+4}^{j-2} = \overline{a}$	$1/2$
	corr. situation 5 (p_1,-,a)	$A_{i+2}^j = a \oplus p_1$		$1/2$
	corr. situation 6 (-,p_2,a)			0
	corr. situation 7 (p_1,p_2,a)	$p_2 = a$	$W_{i+4}^{j-2} = \overline{a}$	1
$i+4$ MAJ	corr. situation 4 (-,-,a)	$A_{i+3}^{j-2} \neq A_{i+2}^j$	$W_{i+4}^{j-2} = \overline{a}$	$1/2$
	corr. situation 5 (p_1,-,a)	$p_1 = \overline{a}$		1
	corr. situation 6 (-,p_2,a)	$p_2 = \overline{a}$		1
	corr. situation 7 (p_1,p_2,a)		$MAJ(p_1, p_2, a) \neq W_{i+4}^{j-2}$	1
$i+4$ XOR	corr. situation 4 (-,-,a)		$A_{i+3}^{j-2} \oplus A_{i+2}^j = W_{i+4}^{j-2} \oplus \overline{a}$	$1/2$ (1 si $j = 1$)
	corr. situation 5 (p_1,-,a)			1
	corr. situation 6 (-,p_2,a)			1
	corr. situation 7 (p_1,p_2,a)		$W_{i+4}^{j-2} = \overline{a} \oplus p_1 \oplus p_2$	1
$i+5$	correction		$W_{i+5}^{j-2} = \overline{a}$	1

TAB. B.2 – PARTIE 2 - Conditions à vérifier pour la réussite d'une collision locale pour SHA-0 ou SHA-1, avec une perturbation introduite à l'étape i et à la position j. Le signe de la perturbation est donnée par la contraite $W_i^j = a$. La première colonne désigne l'étape considerée et la deuxième colonne précise le type d'action appliqué et la situation le cas échéant. La troisième colonne (respectivement la quatrième) donne les conditions pour que la propagation (respectivement l'addition) des différences binaires signées se comporte de façon linéaire. Enfin, la dernière colonne donne la probabilité de réussite (entre parenthèses est indiquée la probabilité lorsque la position considerée est égale à 31).

Résumé

Les fonctions de hachage se situent parmi les primitives les plus utilisées en cryptographie, par exemple pour l'authentification ou l'intégrité des messages. Contrairement à tous les autres outils en cryptographie, ces fonctions n'utilisent aucun secret, mais doivent tout de même satisfaire des notions de sécurité, telles que la résistance à la recherche de collisions ou de préimages. Depuis de nombreuses années, l'algorithme d'extension de domaine de Merkle et Damgård, couplé avec une fonction de compression dédiée de la famille MD ou SHA, forme la base type d'une fonction de hachage. Cependant, de récentes avancées en cryptanalyse ont fortement augmenté l'attention que portent les chercheurs à ce domaine. En réponse à ces vulnérabilités, un appel à soumissions organisé par le NIST a récemment été lancé.

Bien que nous nous soyons aussi attachés à la partie conception durant ces années de recherche, dans ce mémoire nous nous intéressons principalement à la cryptanalyse des fonctions de hachage. Plus précisément, nous nous concentrons sur la recherche de collisions pour la fonction de compression interne. Premièrement, nous étudions et étendons les attaques récentes sur les fonctions de hachage de la famille SHA, fonctions standardisées et de loin les plus utilisées en pratique. Les attaques considérées étant très complexes, nous avons concentré nos efforts pour expliciter tous les détails nécessaires à la compréhension totale du lecteur. Nous présentons les deux meilleures attaques pratiques connues à ce jour contre SHA-0 et SHA-1. Ensuite, nous décrivons la première attaque calculant des collisions pour la famille de fonctions de hachage GRINDAHL, un nouveau candidat reposant sur des principes de conception assez novateurs. Enfin, nous étudions la fonction de hachage FORK-256 et montrons que ce schéma ne peut être considéré comme cryptographiquement sûr.

Mots-clés: cryptographie symétrique, fonctions de hachage, cryptanalyse, SHA, GRINDAHL, FORK-256.

Abstract

Hash functions are one of the most useful primitives in cryptography, for example in authentication or message integrity solutions. In contrast to other tools from cryptography, these functions manipulate no secret. Nonetheless, they must fulfill some security properties such as collision resistance or preimage resistance. For several years, the typical building blocks of these functions have been the domain extension algorithm proposed by Merkle and Damgård with a dedicated compression function from the MD or SHA family. However, recent cryptanalysis works raised the community attention upon this domain. In response to these vulnerabilities, a call for submissions has just been organised by the NIST.

Even if we also sat ourselves in the design side during our research time, we will mostly concentrate on hash functions cryptanalysis in this thesis. More precisely, we will analyse the collision search problem for the internal compression function. Firstly, we study and extend recent attacks on the standardized hash functions from the SHA family, by far the most utilized candidate in practice. Since the attacks considered here are quite complex, we concentrated our efforts in describing all the details needed for a good understanding of the reader. We present the best known practical attacks against SHA-0 and SHA-1. Then, we describe the first algorithm that finds collisions for the GRINDAHL family of hash functions, a new candidate based on innovative design concepts. Finally, we analyse the hash function FORK-256 and show that it cannot be considered as cryptographically strong.

Keywords: Symmetric cryptography, Hash Functions, Cryptanalysis, SHA, GRINDAHL, FORK-256.