Analyse de primitives symétriques

Pierre Karpman

To cite this version:

Pierre Karpman. Analyse de primitives symétriques. Cryptographie et sécurité [cs.CR]. Université Paris Saclay (COmUE); Nanyang Technological University (Singapour), 2016. Français. NNT : 2016SACLX095 . tel-01495634

HAL Id: tel-01495634
https://pastel.hal.science/tel-01495634

Submitted on 25 Mar 2017

NNT : 2016SACLX095

Thèse de doctorat de l'Université Paris-Saclay préparée à l'École Polytechnique et à la Nanyang Technological University

École doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité : Informatique

par

M. Pierre Karpman

Analyse de primitives symétriques

Thèse présentée et soutenue à Palaiseau, le 18 novembre 2016.

Composition du Jury :

M. Louis Goubin	Professeur des universités Université de Versailles	(Président du jury)
Mme. Anne Canteaut	Directrice de recherche Inria de Paris	(Rapporteur)
M. Antoine Joux	Chaire partenariale de l'UPMC Université de Paris 6	(Rapporteur)
M. Benoît Gérard	Ingénieur de recherche Direction générale de l'armement	(Examinateur)
M. Hongjun Wu	Associate professor Nanyang Technological University	(Examinateur)
M. Daniel Augot	Directeur de recherche Inria Saclay Île-de-France	(Directeur de thèse)
M. Pierre-Alain Fouque	Professeur des universités Université de Rennes 1	(Directeur de thèse)
M. Thomas Peyrin	Associate professor Nanyang Technological University	(Directeur de thèse)

Remerciements

Le temps est finalement venu de conclure la rédaction de ce manuscrit, et il semble naturel que les derniers paragraphes à être écrits le soient pour remercier les différentes personnes qui ont contribué à cette thèse, d'une façon ou d'une autre.

En premier lieu, je tiens à remercier mes trois encadrants, Daniel, Pierre-Alain et Thomas. La gestion d'un encadrement triple entre Palaiseau, Rennes et Singapour n'a pas toujours été facile, mais la complexité de cet arrangement a largement été récompensée par la richesse des années de thèse qui en ont résulté. Je suis notamment très heureux d'avoir pu profiter de trois enseignements différents de la cryptographie, qui m'ont sans doute poussé à explorer bien plus de ses aspects que ce à quoi je m'attendais initialement.

Je tiens aussi à remercier tous les membres de mon jury : Anne et Antoine pour avoir accepté de rapporter cette thèse et pour les nombreux commentaires qu'ils ont fourni sur le manuscrit ; Louis pour avoir présidé la soutenance ; Benoît et Hongjun pour avoir accepté d'en être examinateurs ; et encore une fois mes encadrants Daniel, Pierre-Alain et Thomas. Je remercie également toutes les personnes qui sont venues assister à ces quelques 45 minutes de présentation, et particulièrement celles qui pour cela se sont déplacées depuis des lieux parfois lointains.

Le contenu scientifique de cette thèse n'aurait pas été le même sans les différentes collaborations que j'ai eu l'occasion d'avoir. Je remercie spécialement tous mes co-auteurs : Daniel Augot, Patrick Derbez, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Jian Guo, Paul Kirchner, Brice Minaud, Ivica Nikolić, Thomas Peyrin, Marc Stevens, Lei Wang et Shuang Wu, mais aussi toutes les personnes bien plus nombreuses avec qui j'ai pu discuter de cryptographie.

La grande qualité de mes différents environnements de travail pour ces trois années doit aussi beaucoup à mes collègues de bureau, que je remercie vivement : à Rennes, Jean-Christophe, Pierre, Brice, Raphaël et Florent ; à Singapour tous les membres du grand bureau et particulièrement Ivica, Jérémy, Jian, Wang Lei, Wei Lei, Siang Meng, Shuang et Yu ; et tous les membres de l'équipe GRACE qui m'ont accueilli chaleureusement lors de mes visites à Palaiseau. Même s'ils n'ont pas tous partagé un bureau avec moi, je remercie aussi tous les membres des équipes Celtique et EMSEC de Rennes qui ont largement contribué à agrémenter les pauses café et les déjeuners.

Pour s'éloigner du côté scientifique et professionnel, qui bien qu'important ne fait pas tout, je tiens à remercier ma famille dans son ensemble, tous ses membres n'étant d'ailleurs pas étrangers à l'origine de mon intérêt pour l'informatique et la cryptographie.

Je remercie également mes amis de Rennes : Antoine, Simon, Doriane, Christopher, Paul et Pierre, les trois premiers ayant aussi eu la lourde tâche de partager un appartement avec moi ; de Paris : Claire, Gildas et Raphaëlle ; et enfin Raphaël à Toulouse.

Pour finir, je tiens à remercier la direction générale de l'armement pour avoir financé tous ces travaux, et la fonction de hachage ΓOCT 34.11-2012 pour avoir indirectement financé la machine à café de l'équipe à Singapour, outil indispensable à toute réflexion prolongée.

0x0515

Sommaire

Chapitre 1

Introduction

1.1 Algorithmes de chiffrement par bloc

Un *algorithme de chiffrement par bloc*, ou chiffrement (par bloc), ou chiffre (par bloc), est une famille d'applications injectives d'ensembles de départ et d'arrivée finis. On ne s'intéressera dans ce manuscrit qu'à des familles d'applications d'un ensemble de chaînes binaires de taille fixe vers lui-même, c'est à dire à des applications de type $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ telles que $\mathcal{E}(k,\cdot)$ est une permutation pour tout $k \in \{0,1\}^\kappa$. On appelle κ la *taille de clef* et n la *taille de bloc* de $\mathcal{E}$. Les tailles habituelles sont $\kappa \in \{64, 80, 128, 192, 256\}$ et $n \in \{64, 128, 256\}$, bien que les clefs de 64 ou 80 bits ne soient plus considérées de nos jours comme apportant une sécurité suffisante.

On requiert aussi que $\mathcal{E}$ et son inverse $\mathcal{E}^{-1}$ soient calculables efficacement, bien qu'en fonction des applications il puisse être suffisant qu'un seul des deux le soit.

L'emploi le plus courant qui d'un chiffrement par bloc est d'assurer la confidentialité de communications. Deux entités A et B partageant une clef k pour le même chiffrement peuvent communiquer par l'intermédiaire de messages chiffrés $c := \mathcal{E}(k,p)$, $c' := \mathcal{E}(k,p')$, etc. Le second argument p de $\mathcal{E}$ est généralement appelé *texte clair*, et le résultat c est généralement appelé *texte chiffré*. Nous ne nous intéresserons pas ici à la question de savoir comment A et B obtiennent le secret partagé k, les techniques utilisées à cette fin étant nettement différentes de celles employées pour les chiffrements par bloc.

Si $\mathcal{E}$ est tel que la permutation $\mathcal{E}(k,\cdot)$ est difficile à inverser quand k est inconnu, A et B pourraient s'attendre à ce qu'un canal de communication sûr consiste à injecter leurs messages vers les chaînes $m_0||m_1||\ldots||m_\ell$ de tailles multiples de n et à envoyer les chiffrés $\mathcal{E}(k,m_0)||\mathcal{E}(k,m_1)||\ldots||\mathcal{E}(k,m_\ell)$. Ce schéma comporte cependant deux problèmes de taille, indépendamment de la sécurité du chiffrement choisi : en premier lieu, le schéma est déterministe, c'est à dire que chiffrer le même texte clair plusieurs fois donne toujours le même chiffré pour résultat. Un adversaire passif espionnant la conversation sur le canal entre A et B peut donc détecter quand deux messages identiques ont été envoyés. En second lieu, la communication n'est pas authentifiée. Un adversaire actif présent sur le canal peut supprimer et modifier certains des blocs d'un message, ajouter des blocs d'un précédent message, ou encore ajouter des blocs générés aléatoirement. Tout ceci peut être fait sans que A et B ne détectent qu'une entité extérieure agit sur le canal avec des intentions hostiles.

Des problèmes tels que ceux ci-dessus sont résolus en concevant des *modes d'opération* sûrs, mais nous n'étudions pas ceux-ci dans ce manuscrit.

1.1.1 Sécurité des chiffrements par bloc

Le but de cette section est d'expliquer brièvement en quoi consiste un bon chiffrement par bloc, d'un point de vue pratique. Nous commençons toutefois par d'abord définir une idéalisation de cette primitive, sous la forme du modèle du *chiffrement idéal.*

Définition 1.1 (Chiffrement par bloc idéal)**.** Un *chiffrement par bloc idéal* $\mathcal{E}$ est une application $\{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ telle que toutes les permutations $\mathcal{E}(k,\cdot)$ sont tirées aléatoirement et uniformément parmi les permutations de $\{0,1\}^n$.

Cette notion correspond intuitivement à ce qu'on peut atteindre de mieux étant donnée la définition d'un chiffrement par bloc. Pour de petites valeurs de n, par exemple 20, on peut implémenter un chiffrement idéal en utilisant un algorithme de génération de permutation, par exemple celui attribué de façon variable à Fisher, Yates, Knuth, etc. [FY48]. Un tel algorithme permet de tirer uniformément une permutation pour chaque clef, mais déterminer la permutation associée à une clef donnée a une complexité de $\mathcal{O}(2^n)$ en temps et en mémoire. Il n'est donc pas envisageable de suivre cette approche pour les tailles classiques de bloc $n \geq 64$ utilisées en cryptographie. Même pour de petites valeurs de n, l'algorithme de Fisher *et al.* requiert une quantité considérable d'aléa paramétrisé par une clef, ce qui peut constituer un obstacle à son utilisation. En pratique, les chiffrements utilisés ne sont donc que des « approximations » de chiffrements idéaux. Une façon utile, bien qu'essentiellement théorique, de quantifier la sécurité d'un chiffrement par bloc est alors précisément de mesurer à quel point celui-ci est éloigné d'un chiffrement idéal. Informellement, ceci est fait en bornant supérieurement l'*avantage* sur une réponse aléatoire qu'a un adversaire de distinguer s'il interagit avec une permutation tirée aléatoirement ou avec une instance du chiffrement paramétré avec une clef inconnue. Cette proposition peut être précisée sous la forme suivante, similaire à celle qui peut être trouvée par exemple dans [BKR00] :

Définition 1.2 (Permutations pseudo-aléatoires (PRP))**.** Soit $\mathcal{E}$ un chiffrement par bloc de taille de clef κ et de taille de bloc n. On note Π_{2^n} l'ensemble des permutations sur les chaînes binaires de longueur n ; $x \overset{\$}{\leftarrow} \mathcal{S}$ l'action de tirer x aléatoirement et uniformément parmi les éléments de l'ensemble $\mathcal{S}$; $\mathcal{A}^f$ un algorithme ayant accès à un oracle f et qui retourne un unique bit. On définit alors l' *avantage PRP* de $\mathcal{A}$ pour $\mathcal{E}$, écrit $\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A})$ comme :

$$\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A}) = |\Pr[\mathcal{A}^f = 1 \mid f \overset{\$}{\leftarrow} \Pi_{2^n}] - \Pr[\mathcal{A}^f = 1 \mid f := \mathcal{E}(k,\cdot), k \overset{\$}{\leftarrow} \{0,1\}^\kappa]|.$$

La *sécurité PRP* de $\mathcal{E}$ pour une *complexité en donnée* q et une *complexité en temps* t est :

$$\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(q,t) \coloneqq \max_{\mathcal{A} \in \mathrm{Alg}^{f \backslash q, \mathcal{E} \backslash t}} \{\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A})\}.$$

Ici, $\mathrm{Alg}^{f \backslash q, \mathcal{E} \backslash t}$ est l'ensemble des algorithmes $\mathcal{A}$ qui ont accès à un oracle f auquel ils font au plus q accès, et qui tournent en temps inférieur à $\mathcal{O}(t)$, l'unité de temps étant le temps nécessaire pour calculer $\mathcal{E}$ une fois.

La notion de PRP est utile pour par exemple prouver qu'une construction utilisant un chiffrement par bloc n'est pas significativement moins sûre que ce dernier. Un tel résultat est typiquement obtenu en définissant une fonction d'avantage pour la construction de haut niveau similaire à celle utilisée dans la sécurité PRP, et en montrant que cet avantage n'est pas plus qu'une fonction « raisonnable » de la sécurité PRP. Par exemple, dans le cas de [BKR00], la construction de haut niveau considérée est CBC-MAC.

Toutefois, la définition 1.2 n'est pas constructive, dans le sens où elle ne donne pas de procédure efficace permettant de calculer la sécurité PRP d'un chiffrement par bloc en général. Un des objectifs majeurs de la cryptographie symétrique consiste à analyser explicitement des instances de chiffrements par blocs dans le but d'établir leur sécurité concrète contre certaines attaques. S'inspirant de la terminologie de la définition 1.2, ceci revient à trouver des algorithmes pour lesquels q, t et l'avantage PRP sont connus. Une telle attaque sur un chiffrement $\mathcal{E}$ permet alors de borner inférieurement sa sécurité PRP en un certain point (q,t). Cependant, en réalité, déterminer complètement le niveau de sécurité apporté par un chiffrement est plus complexe que ce que la définition 1.2 pourrait faire croire. Des caractéristiques importantes des attaques qui ne sont pas visibles dans cette définition sont la complexité en mémoire ; la quantité de calculs qui ont lieu en ligne ou hors ligne ; savoir si les attaques s'appliquent aussi bien à toutes les clefs ou à seulement certaines d'entre elles ; savoir si elles permettent de retrouver k quand f est une instance de $\mathcal{E}$, ou un algorithme équivalent à $\mathcal{E}(k,\cdot)$, etc. Nous consacrons le reste de cette section à introduire quelques éléments caractéristiques d'attaques de chiffrements par bloc.

1.1.2 Distingueurs et attaques

Beaucoup d'attaques sur les chiffrements par blocs exploitent des *distingueurs*, qui sont des algorithmes utilisant des ressources raisonnables qui ont un avantage non négligeable d'après la définition 1.2. Il n'existe pas de réponse évidente à ce que « raisonnable » et « non négligeable » signifient dans le contexte d'attaques réelles, notamment parce que les tailles de clefs et de blocs sont fixes. Certains chiffrements ou distingueurs potentiels peuvent parfois être paramétrés pour aider à préciser ces notions, mais ce n'est pas toujours le cas. Cependant, les algorithmes utilisés dans des attaques sont souvent suffisamment performants pour lever tout ambiguïté sur leur nature de distingueur. Par exemple, un algorithme permettant de distinguer $\mathcal{E}$ de clef et bloc de taille 128 avec $q = 2$, $t = 2^{20}$ et probabilité 0.9 utilise objectivement peu de ressources par rapport à la taille de $\mathcal{E}$ et a une probabilité de succès élevée.

1.1.2.1 Classes de distingueurs

Nous décrivons brièvement deux types de distingueurs qui exploitent des comportements non idéaux de différentes natures.

Nous commençons par présenter les *distingueurs différentiels*, qui appartiennent à la classe plus générale des distingueurs dits *statistiques*. Le principe d'un distingueur statistique est de définir un événement dont la distribution de probabilité pour la cible à distinguer est différente de celle obtenue pour une permutation tirée aléatoirement dans Π_{2^n}. Appliquer le distingueur correspond alors à collecter un certain nombre d'échantillons obtenus grâce à un oracle et à déterminer quelle est la distribution suivant laquelle ils ont le plus probablement été tirés. Un distingueur différentiel applique ce principe en considérant un certain type d'événement statistique. Une autre classe majeure de distingueurs statistiques est celle des *distingueurs linéaires*.

Soit $\mathcal{E}$ un chiffrement par bloc, une *différentielle* pour $\mathcal{E}$ est une paire $(\Delta \neq 0, \delta)$ de différences d'entrée et de sortie associées à une certaine loi de groupe +. Dans la grande majorité des cas, + est l'addition sur $\mathbf{F}_2^n$, c'est à dire le OU exclusif bit à bit (XOR) ; on utilisera alors souvent la notation alternative $\oplus$ pour cette loi. Parfois, + est l'addition dans $\mathbf{Z}/2^n\mathbf{Z}$, et d'autres fois les différences peuvent être considérées suivant plusieurs lois à la fois. Une *paire différentielle* pour la différentielle (Δ, δ) et pour une certaine clef k est une paire ordonnée de textes clairs et des chiffrés correspondant $((p,c),(p',c'))$ avec p, $c := \mathcal{E}(k,p)$, p', $c' := \mathcal{E}(k,p')$, telle que $p - p' = \Delta$ et $c - c' = \delta$. Quand la différence est sur $\mathbf{F}_2^n$, la soustraction coïncide avec l'addition et devient donc commutative, et la paire n'a alors plus besoin d'être ordonnée. Nous nous plaçons dans un tel cas pour le reste de cette description.

On appelle *probabilité différentielle* d'une différentielle pour une permutation $\mathcal{P}$ la probabilité d'obtenir une paire différentielle pour $\mathcal{P}$: $\mathrm{DP}^{\mathcal{P}}(\Delta, \delta) := \Pr_{p \in \{0,1\}^n}[\mathcal{P}(k,p) \oplus \mathcal{P}(k, p \oplus \Delta) = \delta]$. La caractéristique la plus importante d'une différentielle pour un chiffrement par bloc est sa *probabilité différentielle espérée*, qui est simplement la moyenne sur k de ses probabilités différentielles pour $\mathcal{E}(k,\cdot)$: $\mathrm{EDP}^{\mathcal{E}}(\Delta, \delta) := 2^{-\kappa} \sum_{k \in \{0,1\}^\kappa} \mathrm{DP}^{\mathcal{E}(k,\cdot)}(\Delta, \delta)$. Une hypothèse courante est que pour la plupart des clefs et des différentielles, la probabilité DP à clef fixée est proche de la moyenne EDP. La DP d'une différentielle aléatoire pour une permutation aléatoire peut être approchée par une loi de Poisson : le nombre approché de paires différentielles est $\sim \mathrm{Poi}(2^{-1})$, de moyenne et variance 2^{-1} (voir [DR07], se basant sur un résultat antérieur [O'C95]). Puisqu'il y a 2^{n-1} paires possibles, la DP espérée est de 2^{-n} ; on notera cependant que la DP prend forcément sa valeur parmi les multiples de 2^{-n+1}. Pour qu'un distingueur sur $\mathcal{E}$ soit utile, il faut donc en quelque sorte que son EDP soit différente de 2^{-n}. Si elle en est suffisamment éloignée, par exemple égale à $2^{-3n/4}$, on fait généralement l'hypothèse simplificatrice que toutes les DPs sont égales à leur espérance, ou plutôt la valeur possible la plus proche. Dans ce cas, utiliser un distingueur consiste à accumuler environ $1/\mathrm{EDP}^{\mathcal{E}}(\Delta, \delta)$ textes clairs vérifiant la différence d'entrée et à compter combien d'entre eux vérifient la différence de sortie. Le distingueur fait l'hypothèse qu'il interagit avec $\mathcal{E}$ si et seulement si cette valeur est au moins un.

Tous les distingueurs ne sont pas statistiques ; une autre approche possible est d'exploiter une représentation *algébrique* des chiffrements par bloc. Il est toujours possible de redéfinir un chiffrement $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ comme un ensemble ordonné de fonctions $\mathcal{F}_i : \{0,1\}^{\kappa+n} \to \{0,1\}$ projetant $\mathcal{E}$ sur son $i^{\text{ème}}$ bit de sortie : $\mathcal{E} \equiv (\mathcal{F}_0, \ldots, \mathcal{F}_{n-1})$. Les $\mathcal{F}_i$s peuvent être vues comme des fonctions booléennes $\mathbf{F}_2^{\kappa+n} \to \mathbf{F}_2$ qui sont elles-mêmes en bijection avec les éléments de $\mathbf{F}_2[x_0, x_1, \ldots x_{\kappa+n-1}]/\langle x_i^2 - x_i \rangle_{i<\kappa+n}$, c'est à dire des polynômes en $\kappa + n$ variables sur $\mathbf{F}_2$. Le polynôme correspondant à une fonction booléenne est appelé sa *forme algébrique normale* (ANF) ; l'ANF de $\mathcal{E}$ est l'ensemble ordonné des ANFs de ses projections.

Une caractéristique importante d'une ANF est son degré, qui peut être utilisé pour définir des distingueurs simples mais efficaces. Le degré de l'ANF d'une permutation sur n bits est au plus $n-1$, et il est attendu d'une permutation aléatoire qu'elle soit de degré maximal. Si un chiffrement par bloc a un degré $d < n - 1$, il peut être distingué en étant dérivé en suffisamment de points. Ceci nécessite simplement d'évaluer l'oracle fourni au distingueur sur 2^{d+1} valeurs bien choisies, essentiellement un cube de dimension d, et de toutes les additionner. Si le résultat vaut zéro sur chaque bit, l'oracle est probablement de degré moins que d et est donc supposé être $\mathcal{E}$; si ce n'est pas le cas, il est nécessairement de degré d ou plus et est donc supposé être une permutation aléatoire.

1.1.2.2 Étendre un distingueur pour retrouver une clef

Pour définir la sécurité PRP, la notion de distingueur était suffisante. Cependant, dans le cas d'attaques concrètes, l'objectif final est dans l'idéal de retrouver la clef inconnue utilisée par l'oracle. Le contexte d'une attaque est aussi souvent différent de celui d'un jeu de sécurité PRP, car l'attaquant connaît généralement déjà le chiffrement $\mathcal{E}$ avec lequel il interagit, et sait également que celui-ci n'est pas une permutation aléatoire ; il peut donc paraître finalement inutile d'appliquer un distingueur. Malgré ces objections apparentes, les distingueurs sont utiles dans bien des cas, et sont souvent à la base d'attaques retrouvant la clef.

Nous expliquons maintenant brièvement l'idée derrière les conversions de distingueurs en attaques plus complètes. Pour ceci, nous devons faire certaines hypothèses sur la structure de $\mathcal{E}$, très souvent valables en pratique.

Un *chiffrement par bloc itératif* est un chiffrement $\mathcal{E}$ pouvant être décrit comme la composition multiple d'une *fonction de tour* $\mathcal{R}$, éventuellement aussi composée avec une fonction d'initialisation et une fonction de finalisation que nous ignorons ici : $\mathcal{E} \equiv \mathcal{R} \circ \cdots \circ \mathcal{R}$. On note r le nombre total d'applications de $\mathcal{R}$ dans $\mathcal{E}$ pour une version « complète » de ce dernier. Une attaque basée sur un distingueur et dont l'objectif est de recouvrir la clef consiste tout d'abord à trouver un distingueur sur une version réduite de $\mathcal{E}$ constituée de la composition de $d < r$ fonctions de tour. L'étape suivante consiste à interroger l'oracle sur des entrées vérifiant les conditions du distingueur, par exemple des textes clairs de différence Δ ; les réponses obtenues étant chiffrées avec une version complète du chiffrement, il n'est pas attendu du distingueur qu'il réussisse avec un avantage non négligeable. Cependant, l'idée principale de l'attaque consiste alors à faire une hypothèse sur la valeur d'une partie de la clef inconnue k de $\mathcal{E}$, qui permet de déchiffrer partiellement les textes chiffrés sur $r - d$ tours. Ainsi, si l'hypothèse était correcte, l'attaquant obtient des chiffrés pour le chiffrement réduit à d tours, pour lequel l'avantage du distingueur est supposé non négligeable. En revanche, si l'hypothèse est fausse, le déchiffrement sur $r - d$ tours est fait avec une clef incorrecte et il peut alors être assimilé à du chiffrement. L'adversaire obtient donc des chiffrés équivalents à ceux d'un chiffrement sur $r + (r - d)$ tours, et le distingueur aura probablement à nouveau un avantage négligeable. Finalement, on peut donc voir que cette approche donne une méthode pour vérifier une hypothèse sur une partie a priori inconnue de la clef.

Plusieurs remarques peuvent-être faites sur cette procédure. Tout d'abord, le coût associé aux hypothèses répétées sur la valeur d'une partie de la clef augmente la complexité de l'attaque par rapport à celle du distingueur employé. On ne peut donc utiliser de cette façon que des distingueurs de complexité suffisamment faible. Il n'est aussi possible d'utiliser que des distingueurs permettant à l'attaquant de faire une hypothèse sur une *partie* de la clef avant d'être exécutés, ce point constituant le cœur de l'attaque. Cette condition est souvent vérifiée en pratique, par exemple parce que le distingueur est « local » à une certaine partie de l'état du chiffrement. Pour utiliser un tel distingueur, il peut alors suffire de deviner la partie de la clef utilisée pour chiffrer la partie de l'état intervenant dans le distingueur. Enfin, la partie de la clef qui n'est pas retrouvée grâce au distingueur peut être déterminée de plusieurs façons ; par exemple, un autre distingueur impliquant une hypothèse sur une autre partie de la clef peut être utilisé, ou alors elle peut simplement être énumérée exhaustivement.

1.1.2.3 Modèles d'attaques

Jusqu'à présent, nous avons considéré la sécurité des chiffrements dans un cas simple où l'attaquant a accès à un unique oracle secret. Ce scénario peut être généralisé de plusieurs façons, par exemple en donnant accès à plus d'un oracle. Une généralisation de ce type qui est relativement courante prend la forme du modèle d'attaques à *clefs liées* (ou *corrélées*), où l'attaquant peut appeler $\mathcal{E}(k,\cdot)$, $\mathcal{E}(\phi(k),\cdot)$, avec $\phi(\cdot)$ une ou plusieurs applications sur l'espace des clefs. Une observation importante est que ϕ ne peut pas être arbitraire, car certaines applications sont suffisamment puissantes pour permettre d'attaquer n'importe quel chiffrement ; il ne peut alors pas y avoir de définition sensée de la sécurité de $\mathcal{E}$ dans un tel cas.

Il est donc utile de garder à l'esprit qu'une attaque doit toujours être spécifiée dans un modèle clair. Bien que certains modèles puissent être considérés comme trop puissants et donc moins utiles que d'autres, ce qui est une critique parfois formulée à l'égard des attaques à clefs liées en général, cette question est toujours secondaire par rapport à celle de savoir si un modèle permet de mener des attaques triviales dans n'importe quelles circonstances.

1.1.3 Utilisation des chiffrements par bloc

Pour sécuriser un canal de communication, il n'est pas suffisant de disposer d'un bon chiffrement par bloc. Celui-ci doit aussi être utilisé suivant un *mode d'opération* adéquat, dont le but est de définir comment chiffrer des messages potentiellement longs de plus d'un bloc. Nous ne décrivons pas de mode dans cette section, mais précisons certains critères qu'un bon mode doit vérifier.

Une condition essentielle pour un mode est qu'il ne doit pas être déterministe. En particulier, cela veut dire que chiffrer deux fois le même texte clair avec la même clef doit donner des chiffrés différents. Cette condition peut être formalisée par la notion d'*indistinguabilité* dans un scénario d'*attaque à clair choisi* (IND-CPA), ainsi que ses variantes proches. Ceci est approximativement défini par le processus suivant : un adversaire est doté d'un accès à un oracle de chiffrement pour un certain système, puis il prépare deux messages m_0 et m_1 qu'il envoie à un second oracle. Ce dernier choisit aléatoirement un des deux messages et retourne son chiffré. Finalement, l'adversaire peut à nouveau émettre des requêtes à son premier oracle et doit ensuite deviner quel message a été chiffré. Le cryptosystème est IND-CPA si aucun adversaire aux ressources bornées de façon appropriées ne gagne ce jeu avec un avantage non négligeable. Il est notamment clair qu'aucun système déterministe ne peut être sûr suivant cette définition.

Un autre critère important pour un cryptosystème est qu'il doit permettre aux entités communicantes de s'authentifier. Ceci peut être assuré directement par le mode d'opération d'un chiffrement par bloc, qu'on appelle alors mode de *chiffrement authentifié*. Alternativement, ceci peut être fait en combinant de façon appropriée un mode assurant le seul chiffrement avec un *code d'authentification* (MAC). La tendance actuelle est de favoriser la première approche, qui mène généralement à des systèmes plus efficaces.

1.2 Fonctions de hachage

Une fonction de hachage est une application d'un ensemble quelconque, non nécessairement fini, vers un ensemble fini de petite taille. Comme précédemment, nous nous restreindrons ici au cas binaire ; ainsi, on considérera qu'une fonction de hachage est une application des chaînes binaires de taille arbitraire, appelés *messages* vers des chaînes de taille fixe, appelés *empreintes*, ou *hachés* : $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n$ pour un entier n. La plupart des fonctions de hachage ne suivent pas strictement cette définition et ne sont définies que sur un domaine fini, de taille par exemple égale à 2^{64}. Ceci n'a a priori pas d'impact en pratique car les arguments potentiels d'une fonction sont bien plus petits que ces limites supérieures. La taille de sortie typique d'une fonction de hachage est de quelques centaines de bits, et généralement un multiple de trente-deux ; la plupart des fonctions existantes ont des sorties de taille $n \in \{128, 160, 224, 256, 384, 512\}$, bien qu'une sortie de taille inférieure à 224 bit soit maintenant considérée comme trop courte.

Il est plus compliqué de définir la sécurité des fonctions de hachage *cryptographiques* que celle des chiffrements par bloc de la section précédente. En effet, contrairement à ces derniers, les fonctions de hachage ne sont pas paramétrées et en particulier aucun secret n'intervient dans le calcul de l'empreinte d'un message. Il n'est donc pas possible d'exprimer la résistance d'une fonction via une définition similaire à la définition 1.2. La façon habituelle et relativement informelle de procéder (voir par exemple [MvOV96, Chapitre 9]) consiste alors à définir des messages vérifiant certaines propriétés particulières et à exiger d'une fonction sûre qu'elle ne permette pas à un attaquant de trouver « efficacement » de tels messages. Ces messages particuliers sont définis comme suit.

Définition 1.3 (Préimage)**.** Une préimage de t par la fonction $\mathcal{H}$ est un message m tel que $\mathcal{H}(m) = t$.

Définition 1.4 (Seconde préimage)**.** Une seconde préimage de $t := \mathcal{H}(m)$ par la fonction $\mathcal{H}$ est un message $m' \neq m$ tel que $\mathcal{H}(m') = \mathcal{H}(m) = t$.

Définition 1.5 (Collision)**.** Une collision pour la fonction $\mathcal{H}$ est une paire de messages $(m, m' \neq m)$ telle que $\mathcal{H}(m) = \mathcal{H}(m')$.

Une fonction $\mathcal{H}$ *résistante aux attaques en (seconde) préimages* est une fonction pour laquelle une (seconde) préimage d'une cible t choisie aléatoirement ne peut pas être calculée plus efficacement que pour une fonction dont les sorties sont aléatoires uniformément et indépendantes. Notamment, si la taille de sortie de $\mathcal{H}$ est n, ceci veut dire qu'une (seconde) préimage ne peut pas être trouvée plus efficacement qu'en calculant $\mathcal{H}(x)$ pour environ 2^n entrées x distinctes. Il est important de noter que des préimages peuvent facilement être précalculées pour certaines valeurs, puisque le calcul de $\mathcal{H}(x) = y$ permet d'apprendre que x est une préimage de y. C'est pourquoi on considère uniquement qu'un algorithme attaque la résistance en préimage d'une fonction s'il s'exécute efficacement sur des cibles qui ne sont pas contrôlées par l'attaquant.

Une fonction résistante aux collisions est une fonction pour laquelle on ne peut pas trouver de collisions plus facilement que par un processus générique s'appliquant à n'importe quelle fonction. Autrement dit, une collision ne peut pas être trouvée plus efficacement qu'en calculant $\mathcal{H}(x)$ pour environ $2^{n/2}$ entrées x distinctes.

Algorithmes « dégénérés ». On peut remarquer que pour n'importe quelle fonction $\mathcal{H}$, il existe des algorithmes s'exécutant avec des besoins en temps et en mémoire négligeables et retournant toujours une unique collision (m, m'). Celle-ci peut par exemple avoir été obtenue par un calcul préalable dont le coût n'intervient pas visiblement dans celui de l'algorithme, ou avoir été donnée par un oracle. Comme contrairement aux préimages les collisions ne sont pas associées à des cibles, il n'est pas possible d'exclure de tels algorithmes dégénérés en exigeant qu'ils produisent des collisions associées à des cibles particulières. Cependant, l'existence de ces algorithmes est tout de même ignorée en pratique, et on considère toujours que trouver une collision pour une fonction aux sorties aléatoires uniformément et indépendantes nécessite en moyenne $2^{n/2}$ appels à la fonction.

1.2.1 Applications des fonctions de hachage

Les notions de résistance aux préimages et aux collisions d'une fonction de hachage ne sont pas arbitraires ; au contraire, elles sont imposées par les utilisations concrètes qui sont faites de ces fonctions en cryptographie. Bien que la résistance simultanée à toutes les attaques ne soit pas nécessaire dans toutes les applications, on attend généralement d'une fonction de hachage qu'elle puisse être utilisée dans un grand nombre d'applications différentes, et il est donc désirable qu'elle résiste à tous les types d'attaque.

En guise d'illustration, nous décrivons ci-dessous quelques utilisations possibles des fonctions de hachage.

Paradigme « hacher et signer ». Il est courant d'utiliser des fonctions de hachage au sein de schémas de signature électronique. Ceci est généralement fait pour deux raisons : tout d'abord, les fonctions de hachage sont beaucoup plus rapides à calculer que les algorithmes au cœur des schémas de signature. Il est donc plus efficace de d'abord calculer une empreinte de petite taille d'un message à signer et de ne réellement signer que cette dernière. Ensuite, certains schémas de signature se définissent également plus facilement quand les messages sont de taille fixe ; cette condition est beaucoup plus facile à remplir par des empreintes de messages plutôt que par les messages eux-mêmes.

Les signatures produites par ce type de schémas sont de la forme $(m, \mathcal{S}(k, \mathcal{H}(m)))$, avec k la clef utilisée pour signer avec $\mathcal{S}$. Il est évident que pour qu'un tel schéma soit sûr, la fonction $\mathcal{H}$ doit au moins être résistante en seconde préimage. Dans le cas contraire, un adversaire pourrait intercepter un message m signé par A, remplacer m par m' de même empreinte, et prétendre que A a signé m'. Pour des raisons similaires, on peut voir que la résistance aux collisions est également nécessaire.

Hachage de mots de passe. Le hachage de mots de passe est une autre application courante des fonctions de hachage. On suppose qu'une entité souhaite autoriser des utilisateurs à s'authentifier grâce à un mot de passe. En conséquence, elle doit mémoriser le mot de passe correspondant à chaque utilisateur. Comme il est évident que stocker les mots de passe eux-mêmes mènerait à de graves problèmes de sécurité, une idée consiste à stocker à la place leurs images à travers une fonction de hachage $\mathcal{H}$. Ainsi, même si un adversaire devait trouver la liste des utilisateurs et des mots de passe hachés associés, il ne serait pas capable de trouver les mots de passe eux-mêmes ou des entrées équivalentes, pour peu que $\mathcal{H}$ résiste aux attaques en préimage. Dans cet exemple, résister aux attaques en seconde préimage ou en collision n'est alors pas nécessaire. Cependant, il est bon de noter que le schéma esquissé ci-dessus est en fait trop simpliste pour apporter une réelle sécurité, indépendamment de la fonction de hachage utilisée. Nous ne rentrerons toutefois pas dans les détails de la construction de bons schémas.

Signature à base de fonctions de hachage. Une utilisation moins courante des fonctions de hachage est de les employer pour directement définir un schéma de signature (voir par exemple [Mer87]). Nous ne décrivons pas un tel schéma ici, mais il est intéressant de noter certaines de leurs caractéristiques. Le principe utilisé est que l'entité souhaitant effectuer des signatures publie en avance un certain nombre d'empreintes, tout en gardant secrètes les entrées de la fonction de hachage les produisant. Signer un message consiste alors à révéler certaines de ces entrées ; ainsi, être capable de calculer des préimages pour la fonction permet d'attaquer le schéma, mais les collisions ne sont pas une menace. Enfin, une propriété inhabituelle de ces constructions est qu'elles peuvent être basées sur une fonction de hachage qui a seulement besoin d'être définie sur des entrées de petite taille.

Codes d'authentification. Le dernier usage possible d'une fonction de hachage que nous évoquons ici est la construction de codes d'authentification (MACs), qui peuvent d'une certaine façon être vus comme des variantes à clef des fonctions de hachage. Un MAC $\mathcal{T}$ prend comme entrée une clef k et un message m et retourne un tag $t \coloneqq \mathcal{T}(k, m)$. Un adversaire ne connaissant pas la clef ne doit pas être capable de trouver efficacement une paire (m, t) valide pour $\mathcal{T}(k, \cdot)$, qu'il ait le choix du message m pour lequel produire t ou non. En fonction de ce dernier point, la propriété de sécurité associée est appelée *contrefaçon existentielle* ou *universelle*. Bien sûr, il est aussi nécessaire qu'aucune collision parmi les tags n'ait lieu, que ce soit sur des messages fournis par un adversaire ou non.

Les fonctions de hachage semblent être de bons candidats pour construire des MACs, et en effet des constructions génériques telles que HMAC [BCK96] sont populaires. Cependant, la sécurité exacte fournie par ces constructions n'est pas toujours facile à déterminer, et on dispose généralement d'alternatives plus rapides, comme les MACs dits polynomiaux (voir par exemple [BHK+99]).

1.2.2 Fonctions de hachage de type Merkle-Damgård

Une des premières méthodes à avoir été développée pour concevoir des fonctions de hachage est la construction dite de Merkle-Damgård, due indépendamment à Merkle et Damgård [Mer89, Dam89]. L'idée de cette construction consiste à calculer des empreintes de messages de taille arbitraire en définissant une fonction de hachage comme l'itération d'une *fonction de compression* d'ensemble de départ et d'arrivée de taille fixe. On dit alors que la construction est un *extendeur de domaine*. Cette approche facilite la conception de fonctions de hachage, mais ne garantit pas a priori que le résultat sera résistant aux attaques. La contribution principale de Merkle et Damgård sur ce point là a été de proposer une construction permettant de partiellement réduire la sécurité des fonctions de hachage à celles de leurs fonctions de compression. En effet, ils montrent que pour les attaques en collision, une attaque sur une fonction de hachage utilisant leur construction peut être utilisée pour donner une attaque sur la fonction de compression sous-jacente. Par contraposée, tant qu'on ne connaît pas d'attaque en collision sur la fonction de compression, on peut être convaincu qu'il n'y en a pas sur la fonction de hachage. D'une certaine façon, tout ceci est proche de l'objectif de construire un mode d'opération sûr à partir d'un chiffrement par bloc sûr lui aussi.

La construction Merkle-Damgård est définie à partir d'une fonction de compression $\mathrm{H} : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$. Celle-ci prend deux entrées : une *valeur de chaînage* c et un *bloc de message* m, et produit une autre valeur de chaînage comme sortie. La fonction de hachage de type Merkle-Damgård $\mathcal{H}$ associée à H est construite en étendant le domaine de la seconde à $\{0,1\}^*$ (ou plutôt $\{0,1\}^N$ pour un grand entier N, la plupart du temps). Ceci est fait en spécifiant une *valeur initiale* IV pour la première valeur de chaînage c_0, qui est une constante pour la fonction de hachage, et en définissant l'image d'un message m par $\mathcal{H}$ par la procédure suivante :

1. Le message m est étendu à une taille multiple de la *taille de bloc* b. Plusieurs méthodes peuvent être employées à ces fins, la condition la plus importante étant qu'elles ne produisent pas de collisions triviales. De plus, la plupart du temps, la taille du message non étendu m est incluse dans le message étendu. Ce procédé s'appelle « renforcement Merkle-Damgård » et est essentiel à la sécurité de beaucoup d'instantiations courantes de la construction. Un problème potentiel pouvant apparaître en l'absence d'un tel renforcement est qu'il serait possible d'utiliser des points fixes pour la fonction de compression pour construire des collisions ; de tels points fixes sont faciles à trouver pour certaines fonctions de compression populaires, comme celles de type *Davies-Meyer* décrites dans la section 1.2.5.

2. Le message étendu est itérativement traité par la fonction de compression : soit $\mathsf{m}_0||\mathsf{m}_1||\dots||\mathsf{m}_r$ un message de $r+1$ blocs, on définit $\mathsf{c}_{i+1} \coloneqq \mathrm{H}(\mathsf{c}_i, \mathsf{m}_i)$. L'empreinte $\mathcal{H}(m)$ est égale à la dernière valeur de chaînage c_{r+1}.

Nous donnons une illustration de cette procédure dans la figure 1.1.

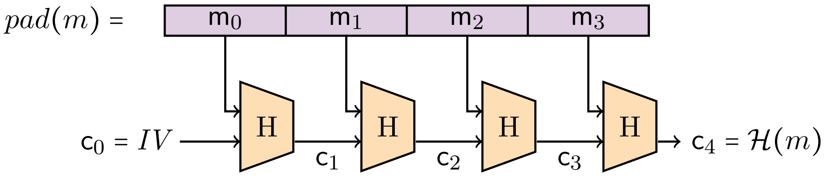


FIGURE 1.1 – Une fonction Merkle-Damgård traitant un message sur quatre blocs. Figure adaptée de [Jea].

Nous concluons cette présentation en esquissant les preuves de réduction de sécurité de la construction Merkle-Damgård pour les attaques en collision et en première préimage. Il n'existe pas de telle preuve pour les secondes préimages, car il existe des attaques génériques indépendamment de la sécurité de la fonction de compression, que nous discuterons brièvement dans la section 1.2.3, et la sécurité exactement atteinte n'est pas connue. Il est aussi important de remarquer qu'une attaque en collision sur une fonction de compression H n'implique pas de façon générique une attaque en collision sur une fonction de hachage de type Merkle-Damgård $\mathcal{H}$ l'utilisant. Cependant, une telle attaque viole la réduction de sécurité, et invalide toute garantie formelle qu'on pourrait avoir sur la résistance en collisions de $\mathcal{H}$. Nous discuterons ce point un peu plus en détails dans la section 1.2.3.

Proposition 1.1. *Une collision sur une fonction de hachage Merkle-Damgård $\mathcal{H}$ implique une collision sur sa fonction de compression* H.

Démonstration. On suppose ici que le renforcement Merkle-Damgård est utilisé, que la taille d'un message est ajoutée à la fin du message étendu qu'il définit, et que cette taille tient sur un unique bloc.

On suppose qu'on a m, $m' \neq m$ t.q. $\mathcal{H}(m) = \mathcal{H}(m')$.

Cas 1 : m et m' ne font pas la même taille. Les derniers blocs de message m_r $\mathsf{m}'_{r'}$ incluent tous les deux les tailles de leurs messages étendus respectifs, qui sont différentes. Ainsi, $(\mathsf{c}_r, \mathsf{m}_r)$ et $(\mathsf{c}'_{r'}, \mathsf{m}'_{r'})$ sont distincts et collisionnent à travers H.

Cas 2 : m et m' sont de la même longueur. On suppose sans perte de généralité que les deux messages étendus font $r + 1$ blocs. On appelle i le numéro de bloc le plus grand tel que $\mathsf{m}_i \neq \mathsf{m}'_i$. Si $i = r$, alors $(\mathsf{c}_r, \mathsf{m}_r)$ et $(\mathsf{c}'_r, \mathsf{m}'_r)$ sont distincts et collisionnent à travers H. Si $i < r$, ou bien $\mathsf{c}_{i+1} = \mathsf{c}'_{i+1}$ (et donc $(\mathsf{c}_i, \mathsf{m}_i)$ et $(\mathsf{c}'_i, \mathsf{m}'_i)$ forment une collision valide pour H), ou bien il existe une suite non vide de paires $(\mathsf{c}_j, \mathsf{m}_j)$ and $(\mathsf{c}'_j, \mathsf{m}'_j)$, $j = i + 1 \ldots r$ telles que $\mathsf{m}_j = \mathsf{m}'_j$ pour tout j. Puisque $\mathsf{c}_{r+1} = \mathsf{c}'_{r+1}$, au moins un des éléments de la suite collisionne à travers H. Les deux entrées du premier de ceux-ci étant différentes, la collision qu'elles définissent est non triviale. □

Proposition 1.2. *Une préimage sur une fonction Merkle-Damgård $\mathcal{H}$ implique une préimage sur sa fonction de compression* H.

Démonstration. Soit m un message étendu sur $r + 1$ blocs t.q. $\mathcal{H}(m) = t$, avec t la cible de préimage, alors $\mathrm{H}(\mathsf{c}_r, \mathsf{m}_r) = t$, et $(\mathsf{c}_r, \mathsf{m}_r)$ est une préimage valide pour H. □

1.2.3 Préciser la sécurité des fonctions de hachage

Les notions de sécurité de résistance aux collisions et aux préimages peuvent être raffinées et complétées avec des notions supplémentaires. Celles-ci ne sont pas toujours pertinentes pour les emplois concrets qui sont faits des fonctions de hachage, mais elles sont néanmoins utiles pour évaluer la sécurité d'une fonction de manière plus fine. Ce raffinement peut se faire de deux façons : ou bien en caractérisant le comportement non idéal d'une méthode de construction de fonctions de hachage, comme par exemple la construction Merkle-Damgård ; ou bien en caractérisant les faiblesses d'une partie d'une fonction de hachage, par exemple une fonction de compression. Afin de donner des définitions plus explicites de ces notions supplémentaires, il est utile de tout d'abord définir une vision plus forte et idéalisée des fonctions de hachage :

Définition 1.6 (Oracle aléatoire)**.** Un *oracle aléatoire* sur n bits $\mathcal{R}$ est une application $\{0,1\}^* \rightarrow \{0,1\}^n$ telle que pour toute entrée x, l'image $\mathcal{R}(x)$ est tirée aléatoirement et uniformément dans $\{0,1\}^n$.

Un oracle aléatoire est nécessairement résistant aux attaques que nous avons définies jusqu'à présent, puisqu'il est clair que seuls des algorithmes génériques peuvent être utilisés sur un tel objet. De cette façon, un oracle aléatoire capture entièrement les applications qui peuvent être réalisées avec des fonctions de hachages : si une construction de haut niveau n'est pas sûre quand elle est réalisée avec un oracle aléatoire, elle ne le sera pas non plus si ce dernier est remplacé par une fonction de hachage concrète. En revanche, même si une construction est sûre dans un tel modèle, dit de l'oracle aléatoire, elle ne le sera pas nécessairement une fois réalisée en pratique.

Les fonctions de type Merkle-Damgård présentent certains comportements non idéaux dans le sens où elles permettent à certains calculs d'être plus efficaces que pour un oracle aléatoire.

Un bon exemple de tels calculs correspond au concept de *multicollision*, qui consiste à chercher $r > 2$ messages $m_0, \ldots, m_{r-1}$ dont les images par $\mathcal{H}$ sont toutes égales. La complexité générique de ce problème est $\mathcal{O}(2^{n \times (r-1)/r})$ appels à $\mathcal{H}$ pour une fonction sur n bits, mais Joux a montré comment trouver des (2^r)-multicollisions en temps $\mathcal{O}(r \times 2^{n/2})$ pour les fonctions de type Merkle-Damgård [Jou04]. L'idée de base utilisée dans cette attaque exploite le fait que des collisions pour une fonction Merkle-Damgård $\mathcal{H}$ peuvent être chaînées ensemble pour mener à un nombre exponentiellement grand de messages distincts ayant tous la même empreinte. Ceci peut se voir avec l'exemple suivant : supposons qu'un attaquant ait trouvé deux messages distincts de même longueur m_0 et m'_0 formant une collision ; ceux-ci peuvent être trouvés génériquement pour un coût de $2^{n/2}$. Il peut ensuite chercher une seconde collision, cette fois pour la fonction $\widetilde{\mathcal{H}}$ obtenue en remplaçant la valeur de chaînage initiale c_0 par $\mathcal{H}(m_0) = \mathcal{H}(m'_0)$, ce qui peut encore être fait pour un coût $2^{n/2}$, donnant m_1 et m'_1. La propriété de chaînage de la construction Merkle-Damgård implique alors que les quatre messages $m_0||m_1$, $m'_0||m_1$, $m_0||m'_1$, $m'_0||m'_1$ ont tous la même empreinte. Il est facile de voir que cette procédure se généralise à des messages plus longs, ce qui donne la complexité mentionnée égale à $\mathcal{O}(r \times 2^{n/2})$. On peut observer que la faiblesse de la construction qui est exploitée ici est le fait qu'une collision sur les empreintes implique une collision sur l'état interne de la fonction de hachage. Nous verrons brièvement dans la section 1.2.4 qu'augmenter la taille de l'état d'une fonction est effectivement un moyen de se protéger contre de telles attaques.

Un autre bon exemple d'une faiblesse de Merkle-Damgård, qui cette fois viole directement une des propriétés de sécurité définie au début de cette section, consiste en les attaques génériques en seconde préimages pour des messages longs. Dean [Dea99], puis indépendamment Kelsey et Schneier [KS05] ont montré comment il est possible d'exploiter la structure Merkle-Damgård et des collisions internes pour calculer la seconde préimage d'un long message plus efficacement qu'avec un algorithme générique. La complexité de l'attaque de Kelsey et Schneier pour trouver une seconde préimage pour un message de 2^k *blocs* est $\approx \mathcal{O}(2^{n-k+1})$ appels à $\mathcal{H}$. Cela veut dire que les fonctions Merkle-Damgård sont en fait intrinsèquement attaquées si on interprète strictement ce que nous avons énoncé comme objectifs de sécurité. Cependant, bien que significatives, de telles attaques restent coûteuses, particulièrement pour les tailles habituelles de messages. En conséquence, elles ne sont pas généralement considérées comme posant une menace sur l'utilisation de fonctions Merkle-Damgård en pratique, et de telles fonctions comme SHA-2 sont toujours utilisées [NIS15a].

Nous avons déjà mentionné dans la section 1.2.3 qu'une attaque sur la fonction de compression d'une fonction Merkle-Damgård fait que celle-ci ne peut plus être supposée « sûre ». Il paraît donc naturel d'également analyser la sécurité des fonctions de compressions en tant que telles. Une attaque dans ce cas démontrerait alors un comportement non idéal du deuxième type mentionné ci dessus, puisqu'elle ne ciblerait pas la fonction de hachage elle-même mais un de ses éléments constitutifs. Il existe une façon naturelle de généraliser les propriétés de sécurité définies plus haut à ce contexte, ce qui nous mène aux notions d'attaques à initialisation (semi) libre :

Définition 1.7 (Préimage à initialisation libre)**.** Une *préimage à initialisation libre* pour une fonction de hachage Merkle-Damgård $\mathcal{H}$ est une paire (i, m) d'IV et de message telle que $\mathcal{H}_{\mathsf{i}}(m) = t$, avec $\mathcal{H}_{\mathsf{i}}(\cdot)$ dénotant la fonction $\mathcal{H}$ dont l'IV original a été remplacé par i.

Définition 1.8 (Collision à initialisation semi-libre)**.** Une *collision à initialisation semi-libre* pour une fonction Merkle-Damgård $\mathcal{H}$ est une paire $((\mathsf{i}, m), (\mathsf{i}, m'))$ de deux paires d'IV et de messages telle que $\mathcal{H}_{\mathsf{i}}(m) = \mathcal{H}_{\mathsf{i}}(m')$.

Définition 1.9 (Collision à initialisation libre)**.** Une *collision à initialisation libre* pour une fonction Merkle-Damgård $\mathcal{H}$ est une paire $((\mathsf{i}, m), (\mathsf{i}', m'))$ de deux paires d'IV et de messages telle que $\mathcal{H}_{\mathsf{i}}(m) = \mathcal{H}_{\mathsf{i}'}(m')$.

On peut remarquer que si deux messages d'une collision à initialisation libre ont une longueur d'un bloc, la définition ci-dessus devient équivalente à celle d'une collision sur la fonction de compression H utilisée pour construire $\mathcal{H}$. Il y a peu de différences entre les deux notions en général, l'intérêt principal des attaques à initialisation libre comparées aux attaques sur la fonction de hachage étant qu'elles peuvent profiter de la liberté supplémentaire offerte par la valeur de chaînage de la fonction de compression.

On attend d'une bonne fonction qu'elle résiste autant aux attaques à initialisation libres qu'à leurs variantes plus classiques. Par exemple, une collision à initialisation libre ne doit pas pouvoir être trouvée avec moins d'environ $2^{n/2}$ requêtes à la fonction de compression H.

Enfin, une notion supplémentaire quelque peu mal définie s'ajoutant à celles discutées jusqu'à présent est le concept de distingueur pour fonctions de hachage, qui cherche à capturer les comportements non idéaux qui ne le sont pas par les notions définies ci-dessus. Nous ne donnerons pas de définitions ici, car celles-ci sont difficiles à formaliser dans le cas de fonctions de hachage, à cause de l'absence de clefs. À la place, nous mentionnons brièvement un exemple de distingueur pour la fonction de compression de SHA-1.

Prenant un peu d'avance, nous verrons dans la partie III que l'IV et les blocs de message de la fonction de compression de SHA-1 sont faits de cinq et seize mots de 32 bits respectivement. En 2003, Saarinen a montré que des *paires glissées* peuvent être trouvées pour cette fonction de compression pour un coût équivalent à 2^{32} appels [Saa03]. De telles paires consistent en deux IVs $\mathcal{A}_{0,\ldots,4}$, $\mathcal{A}'_{0,\ldots,4}$ et messages $m_{0,\ldots,15}$, $m'_{0,\ldots,15}$ avec $\mathcal{A}'_i = \mathcal{A}_{i-1}$ et $m'_i = m_{i-1}$, tels que la paire des sorties de la fonction appelée sur ces entrées possède aussi cette propriété. Bien qu'il ne soit pas attendu d'une fonction aléatoire qu'elle possède une telle propriété, il ne semble pas dans ce cas qu'on puisse exploiter celle-ci pour monter une attaque sur une des notions de sécurité définies ci-dessus.

1.2.4 Constructions de fonctions de hachage modernes

Nous avons mentionné deux faiblesses génériques des fonctions Merkle-Damgård. La présence de celles-ci fait que les fonctions de hachage modernes sont généralement basées sur des constructions alternatives offrant une meilleure sécurité. Nous présentons brièvement deux d'entre elles : la variante *tuyau-large* de Merkle-Damgård (ou *coupe-MD*), et les constructions *éponge*.

Merkle-Damgård « tuyau-large ». La construction tuyau-large a été introduite en 2005 par Lucks [Luc05] et Coron *et al.*, sous le nom coupe-MD [CDMP05]. Elle est conceptuellement simple, et consiste à utiliser la construction Merkle-Damgård avec une fonction de compression de taille de sortie plus grande que celle de la valeur de chaînage. Si on écrit $\lfloor\cdot\rfloor_n$ une fonction de troncation arbitraire de $m > n$ vers n bits, on peut définir une construction tuyau-large de n bits basée sur une fonction de compression $\mathrm{H} : \{0,1\}^m \times \{0,1\}^b \to \{0,1\}^m$ comme $\lfloor\mathcal{H}(\cdot)\rfloor_n$, avec $\mathcal{H}$ une fonction Merkle-Damgård classique construite depuis H. Quelques variations sont possibles, par exemple en considérant d'autres applications de m vers n que des troncations.

On peut facilement voir qu'une collision pour une telle fonction de hachage n'implique plus une collision sur son état interne. En choisissant m suffisamment grand, par exemple $m = 2n$, on peut obtenir une résistance générique aux multicollisions. En fait, Coron *et al.* ont montré que cette construction est un *extendeur de domaine sûr pour un oracle aléatoire* ; c'est à dire que si la fonction de compression est un oracle aléatoire avec des entrées de taille fixe, l'utiliser avec une construction tuyau-large donne une fonction ε-indifférentiable d'un oracle aléatoire (au sens de Maurer *et al.* [MRH04]), avec $\varepsilon \approx 2^{-t}q^2$, $t = m - n$ le nombre de bits tronqués et q le nombre de requêtes faites à H. Ceci est un résultat très utile, puisqu'il dit qu'au contraire d'une construction Merkle-Damgård classique, aucun comportement non idéal n'est introduit par la construction d'extension de domaine. De telles fonctions de hachage sont donc supposées se comporter comme les oracles aléatoires qu'on attend qu'elles réalisent, à condition que leurs fonctions de compression soient « idéales » et qu'elles ne soient pas appelées un nombre de fois trop important par rapport à la taille de leurs paramètres.

Construction éponge. La construction éponge a été introduite en 2007 par Bertoni *et al.* [BDPA07]. Elle est quelque peu différente de la construction Merkle-Damgård, notamment parce qu'elle n'utilise pas une fonction de compression comme primitive, mais une fonction d'ensemble de départ et d'arrivée identiques. Habituellement, cette fonction est en plus bijective.

La construction en elle-même est simple. Supposons qu'on souhaite construire une fonction sur n bits basée sur une permutation P sur b bits. On définit le *taux* r et la *capacité* c comme deux entiers tels que que $b = r + c$. Hacher le message m consiste à l'étendre à une taille multiple de r et à le traiter itérativement en deux phases. La phase d'*absorption* calcule une valeur pour l'état interne $\mathsf{i} := \mathrm{P}(\mathrm{P}(\ldots\mathrm{P}(\mathsf{m}_0||0^c) \oplus \mathsf{m}_1||0^c)\ldots)$. La phase de *compression* produit ensuite une sortie sur n bits définie par $\mathcal{H}(m) := \lfloor\mathsf{i}\rfloor_r||\lfloor\mathrm{P}(\mathsf{i})\rfloor_r||\ldots||\lfloor\mathrm{P}^{n\div r}(\mathsf{i})\rfloor_{n \bmod r}$.

Une caractéristique notable des constructions éponges est que la taille de sortie d'une instance est naturellement décorrélée de la taille de sa fonction constitutrice. Ainsi, cette construction permet facilement de construire des fonctions de hachage à taille de sortie variable. Une même permutation ou fonction peut aussi être utilisée dans différentes réalisations offrant un compromis entre la vitesse (un taux plus élevé donne des fonctions plus rapides) et la sécurité (une capacité plus élevée donne une fonction plus sûre).

Bertoni *et al.* ont aussi montré en 2008 que de la même façon que Merkle-Damgård tuyau-large, la construction éponge réalisée avec une permutation ou une fonction aléatoire est ε-indifférentiable d'un oracle aléatoire de sortie de même taille, avec $\varepsilon \approx 2^{-c}q^2$ [BDPA08]. Pour atteindre les propriétés de sécurité classiques, il est alors optimal de prendre $c = 2n$.

Un des meilleurs exemples de fonction éponge est Keccak [BDPA11], qui est devenu le nouveau standard SHA-3 en 2015 [NIS15b].

1.2.5 La famille MD-SHA

Nous présentons maintenant la famille de fonctions de hachage MD-SHA, à la fois pour son importance historique et parce que la fonction SHA-1 étudiée plus tard dans ce manuscrit est un de ses membres.

Les origines de cette famille datent de la conception de la fonction MD4, introduite par Rivest [Riv90] en 1990. Une attaque sur une version réduite a rapidement été trouvée par den Boer et Bosselaers [dB91], et Rivest a proposé MD5 comme version améliorée. Bosselaers a lui-même proposé RIPEMD en 1992 comme amélioration alternative de MD4 [BP95, Chapitre 3], et la NSA fit de même l'année suivante en introduisant la première génération des algorithmes SHS/SHA [NIS93]. Dans les deux derniers cas, les fonctions ont été modifiées peu de temps après, en 1996 et 1995 respectivement [DBP96, NIS95]. D'autres algorithmes tels que SHA-2 [NIS15a], introduits 2002, ont également été influencés par MD4.

Il existe quelques variations parmi les membres de la famille ; notablement RIPEMD utilise une structure parallèle pour sa fonction de compression. Nous énumérons spécifiquement ci-dessous les caractéristiques partagées par MD4, MD5 et SHA, qui sont aussi communes aux autres fonctions MD-SHA dans une large mesure.

- La construction Merkle-Damgård est utilisée comme extendeur de domaine.

- La fonction de compression est construite à partir d'un chiffrement par bloc ad hoc utilisé en mode *Davies-Meyer* : soit $\mathcal{E}(x, y)$ le chiffrement du texte clair y avec la clef x par $\mathcal{E}$, alors la fonction de compression est définie par $\mathsf{c}_{i+1} := \mathcal{E}(\mathsf{m}_{i+1}, \mathsf{c}_i) + \mathsf{c}_i$.

- Le chiffrement par bloc utilisé dans la fonction de compression est un réseau de Feistel déséquilibré qui utilise des additions modulaires, des XORs, des rotations sur les bits, et des fonctions booléennes bit à bit comme éléments constitutifs. Son expansion de clef est linéaire et très efficace à calculer.

La compétition SHA-3 organisée par le NIST entre 2007 et 2012 a motivé la conception d'un grand nombre de nouvelles fonctions de hachage basées sur des principes variés. La famille MD-SHA a influencé un certain nombre de candidats, et cette influence continue d'être exercée encore aujourd'hui.

Chapitre 2

Présentation de mes travaux

Les travaux réalisés pendant cette thèse appartiennent tous au domaine de la cryptographie symétrique ; on peut néanmoins distinguer certains sous-thèmes en fonction des objectifs et des objets d'étude.

Une importante partie de cette thèse et de ce manuscrit est dédiée à des attaques sur la fonction de hachage SHA-1. Notre objectif principal concernant cette fonction a été d'obtenir une attaque explicite, dans le sens où l'attaque peut être exécutée jusqu'à son terme, sur une version *complète* de SHA-1, ceci contrairement aux attaques explicites précédentes qui ont toujours ciblé des versions réduites ; le type d'attaque retenu a été celui des attaques en collision, qui sont à ce jour les plus efficaces sur SHA-1. Pour atteindre notre objectif, néanmoins, nous nous sommes placés dans un modèle à initialisation libre, légèrement plus favorable à l'attaquant.

Nous avons aussi développé une seconde attaque sur SHA-1 visant cette fois à calculer efficacement des préimages. Ces résultats restent cependant théoriques, dans le sens où le coût de l'attaque est trop important pour mener à un calcul explicite comme dans le premier cas. De plus, aucune attaque, même théorique, n'est connue pour la fonction complète, et notre objectif a précisément été d'augmenter le nombre de tours de la fonction pour lequel une attaque peut être menée.

D'autres travaux menés pendant cette thèse ont consisté en des attaques sur des algorithmes différents de SHA-1. Le premier de ces autres résultats a porté sur la famille de primitive ASASA, qui est faite de plusieurs instances à clef secrète et à clef publique. Nous avons développé une idée générale d'attaque exploitant certaines propriétés algébriques de la structure utilisée dans ASASA, ce qui nous a permis d'attaquer toutes les instances proposées, à l'exception d'une, qui avait également été attaquée avant nos travaux en utilisant d'autres méthodes.

Un second algorithme que nous avons attaqué en plus de SHA-1 est le schéma de chiffrement authentifié Prøst-OTR, pour lequel nous avons développé une attaque à clefs liées extrêmement efficace. L'idée utilisée dans l'attaque est très simple, et permet génériquement d'améliorer en un certain sens les attaques à clefs liées existantes sur les constructions de type Even-Mansour.

Enfin, une autre partie de cette thèse a été consacrée à la conception d'algorithmes de chiffrements et de certains de leurs éléments constitutifs. Une première contribution a porté sur la conception de matrices de diffusion de grande taille, que nous avons définies à partir de codes géométriques. Nous nous sommes aussi intéressés aux algorithmes de chiffrement dits légers, qui sont conçus spécifiquement pour être utilisés dans des environnements contraints, comme par exemple un petit processeur. Nous avons développé l'algorithme Fly, qui répond bien à ce contexte. Enfin, nous avons conçu PuppyCipher et CoureurDesBois, deux algorithmes pouvant être implémentés de façon incompressible, et qui de cette façon répondent à un certain modèle de chiffrement en boîte blanche.

Bien que certains des travaux mentionnés ci-dessus ne soient pas inclus dans ce manuscrit, toutes ces contributions sont cependant décrites avec un peu plus de détail dans les trois sections ci-dessous. Nous excluons toutefois de ces descriptions deux articles écrits avant le début de cette thèse.

2.1 Deux cryptanalyses

2.1.1 Attaques sur ASASA [MDFK15]

Nous décrivons ici des travaux réalisés avec Brice Minaud, Pierre-Alain Fouque et Patrick Derbez, publiés à ASIACRYPT 2015.

À ASIACRYPT 2014, Biryukov *et al.* ont proposé la structure ASASA, dans le but de réaliser un certain nombre de primitives à clef secrète et à clef publique [BBK14]. La structure commune à ces primitives consiste à alterner un petit nombre de fois des applications linéaires, ou plus généralement affines, A avec des applications non-linéaires S, une structure complète pouvant être notée comme $A''\circ S'\circ A'\circ S\circ A$. De façon relativement inhabituelle, le secret paramétrant ces structures n'est pas injecté séparément, mais ce sont les applications A, S, etc. elles-mêmes qui sont tenues secrètes. En pratique, celles-ci peuvent être générées depuis un générateur pseudo-aléatoire cryptographique paramétré par la « vraie » clef). Par exemple, les instances ASASA de chiffrement à clef secrète proposées par Biryukov *et al.* sont constituées de trois matrices inversibles tirées aléatoirement dans $\mathcal{M}_{128}(\mathbf{F}_2)$, alternant avec deux applications parallèles de seize boîtes S toutes générées aléatoirement et indépendamment les unes des autres. D'un point de vue pratique, de telles constructions peuvent être intéressantes, car elles nécessitent peu de tours par rapport à une structure plus classique. D'un point de vue plus théorique, elles peuvent aussi être vues comme des généralisations de structures plus habituelles telles que les réseaux de substitution-permutation (SPNs) ; une étude de la structure ASASA peut ainsi par exemple fournir une borne inférieure sur le nombre de tours de n'importe quel SPN, en dessous de laquelle celui-ci pourrait être attaqué génériquement.

Dans nos travaux, nous avons montré que les constructions basées sur ASASA présentent des faiblesses qui permettent de retrouver efficacement les différentes applications A, S, etc., à équivalence prête. Ceci est le cas pour l'instance « chiffrement par bloc » esquissée ci-dessus, mais aussi pour une instance définissant un schéma de cryptographie multivariée à clef publique, ainsi que des variantes à taille réduite de l'instance chiffrement par bloc utilisées pour définir des algorithmes en boîte blanche. Il est à noter qu'un quatrième type d'instance de type clef publique, que nous n'attaquons pas, avait aussi été proposé par Biryukov *et al.*. Ces instances ont cependant aussi été attaquées par Gilbert *et al.* à CRYPTO 2015 [GPT15].

La base de nos attaques consiste à exploiter des faiblesses du degré algébrique en sortie des composants de la structure ASASA. Nous détaillons brièvement cette idée dans le cas de l'instance chiffrement par bloc. On considère les formes algébriques normales (ANFs) associées aux bits de sortie après la seconde application S', juste avant l'application de la dernière couche A'' ; celles-ci sont de degré au plus 49, car les boîtes des couches S sont sur huit bits et inversibles. Si on multiplie deux de ces ANFs issues de boîtes différentes de S', le degré du résultat sera très probablement supérieur à 49 ; ce n'est par contre pas le cas si les ANFs sont issues de la même boîte, le degré restant alors toujours borné supérieurement par 49. À la suite de cette observation, il est possible de définir un système d'équations dont la résolution fournit la couche A'', à équivalence prête ; essentiellement, les coefficients de A'' sont les inconnues, et les contraintes consistent à forcer les produits de deux sorties des mêmes boîtes S à sommer à zéro sur un certain nombre de cubes. Une fois le système résolu, il ne reste plus qu'à par exemple appliquer la fin de l'attaque sur les constructions SASAS de Biryukov et Shamir [BS01]. La complexité totale de notre attaque est dominée par le calcul des cubes, et équivalente à environ 2^{63} appels à l'algorithme de chiffrement.

2.1.2 Attaques à clefs liées sur Prøst-OTR [Kar15]

Nous décrivons ici un article publié à ISC 2015.

En 1991, à ASIACRYPT, Even et Mansour ont proposé une construction générique de chiffrement par bloc qui porte désormais leur nom [EM91]. Partant d'une permutation « aléatoire » fixe et publiquement connue $\mathcal{P}$, on définit le chiffrement $\mathcal{E}((k_1, k_0), p)$ du texte clair p comme $\mathcal{P}(p \oplus k_0) \oplus k_1$. Soit n la taille de bloc de ce chiffrement et en supposant qu'on accède à $\mathcal{P}$ en boîte noire, on peut montrer que la probabilité de retrouver les clefs d'une instance inconnue de $\mathcal{E}$, ou même seulement de la distinguer d'une permutation aléatoire, est inférieure à $\mathcal{O}(qt \cdot 2^{-n})$, avec q le nombre d'accès à l'oracle de chiffrement et t le nombre d'accès à la permutation. De plus, on connaît une attaque atteignant cette borne.

Si on considère un modèle d'attaque *à clefs liées*, la construction Even-Mansour est cependant beaucoup plus faible. En effet, supposant un accès à deux oracles de chiffrement O et O', l'un avec la clef (k_1, k_0) et l'autre avec la clef $(k_1, k_0 \oplus \delta)$ avec δ une constante non nulle connue de l'attaquant, il est clair que celui-ci peut facilement distinguer $\mathcal{E}$ d'une construction aléatoire en vérifiant que $O(p) = O'(p \oplus \delta)$. Nous avons alors fait l'observation élémentaire que ce distingueur peut être rendu conditionnel à la valeur des bits de la clef en utilisant des clefs liées par l'addition modulaire plutôt que le XOR. Nous présentons cette idée dans le cas simple mais répandu où $k := k_0 = k_1$. Soit $O(\cdot) := \mathcal{E}(k, \cdot)$, $O_i(\cdot) := \mathcal{E}(k \boxplus \Delta_i, \cdot)$ avec $\Delta_i = 2^i$, représenté par une chaîne binaire de n

bits valant tous zéro sauf celui en position i ; on peut alors apprendre la valeur du $i^{\text{ème}}$ bit de k en comparant $x := O(p)$ et $y := O_i(p \oplus \Delta_i)$: si celui-ci vaut zéro, alors la valeur en entrée de la permutation $\mathcal{P}$ pour O_i est la même que pour O, et on a $y \boxminus x = \Delta_i$; s'il vaut un, les entrées de $\mathcal{P}$ sont différentes pour les deux oracles, et la différence de leurs sorties est très probablement différente de Δ_i ; les deux cas peuvent donc se distinguer efficacement, pour le coût d'une requête à chaque oracle. Un attaquant ayant accès à O et O_i, $i \in \{0, \dots, n-2\}$, peut donc retrouver l'ensemble de la clef k avec un nombre de requête linéaire en sa taille.

Nous avons appliqué l'attaque à clef liée décrite ci-dessus à l'algorithme de chiffrement authentifié Prøst-OTR [KLL⁺14], qui est une instance du mode de chiffrement OTR avec un chiffrement de type Even-Mansour. L'utilisation d'un padding sur la moitié du bloc fait qu'une application directe de notre idée ne permet de retrouver que la moitié de la clef ; toutefois, nous montrons comment exploiter la propagation de retenue dans l'addition modulaire ainsi que la position du padding pour également retrouver la seconde moitié pour le même coût que la première.

2.2 Conception de primitives

2.2.1 Construction de matrices de diffusion issues de codes géométriques [AFK14]

Nous décrivons ici des travaux effectués avec Daniel Augot et Pierre-Alain Fouque, publiés à SAC 2014.

Une structure classique utilisée dans la conception de chiffrements par bloc est celle des réseaux de substitution-permutation (SPNs), qui alternent applications linéaires et non-linéaires, ces dernières étant généralement définies comme l'application parallèle de boîtes S de petites tailles. Les applications linéaires utilisées peuvent diversement consister en des permutations binaires, des matrices sur un corps fini (généralement $\mathbf{F}_2$, $\mathbf{F}_{2^4}$ ou $\mathbf{F}_{2^8}$), ou une composition des deux. Bien sûr, toute application linéaire peut être représentée par une matrice sur une structure bien choisie. Ce n'est cependant pas toujours sa représentation la plus naturelle.

Dans ces travaux, nous nous sommes intéressés à la conception de matrices denses de grande dimension pouvant être utilisées pour assurer toute la diffusion dans un SPN, c'est à dire qu'elles agissent sur l'ensemble du bloc. Concrètement, les matrices que nous avons étudiées ont une dimension de 16 et permettent de définir des algorithmes avec des blocs de 64 bits, quand définies sur $\mathbf{F}_{2^4}$, ou 128 bits, quand définies sur $\mathbf{F}_{2^8}$. Utiliser une matrice de grande dimension permet d'assurer une diffusion très rapide, à condition qu'elle possède un *branch number* (BN) élevé ; cette notion correspond à la distance minimale d'un code généré par la matrice. Le lien entre matrices de diffusion et codes correcteurs fait qu'il est naturel de construire les premières à partir des seconds. En particulier, il est courant de sélectionner des codes *MDS*, atteignant la borne de Singleton. Malheureusement, on ne connaît pas de tels codes au delà d'une certaine dimension fonction de la taille du corps, et en particulier aucun code MDS de dimension 16 sur $\mathbf{F}_{2^4}$ n'est connu.

Nos travaux ont donc particulièrement consisté à explorer l'utilisation de codes géométriques pour définir des matrices de diffusion de grande dimension sur de petits corps, et de BN en pratique seulement légèrement inférieur à celui d'une hypothétique matrice MDS de même taille. Un code géométrique est dans notre cas un code construit à partir d'une courbe algébrique, qui permet de définir à la fois un ensemble de points, ou plus précisément de places, et des espaces de fonctions associés à ses diviseurs ; on peut alors définir des codes comme l'évaluation d'un espace de fonctions donné sur un sous-ensemble des points. En choisissant correctement la courbe et l'espace de fonctions, on peut par exemple construire une matrice de dimension 16 sur $\mathbf{F}_{2^4}$ de BN 15, soit seulement deux de moins que la borne de Singleton. Cependant, implémenter efficacement la multiplication par une telle matrice n'est pas immédiat, et nous proposons aussi un algorithme vectoriel rapide qui répond à ce besoin.

2.2.2 L'algorithme de chiffrement léger Fly [KG16]

Nous décrivons ici des travaux préliminaires réalisés avec Benjamin Grégoire.

Nous nous intéressons toujours à la structure SPN, mais cette fois dans le cas d'un algorithme ciblant des plateformes dont les ressources sont limitées, ce qui place des contraintes sur les éléments constitutifs de la fonction de tour. Notamment, les grandes matrices denses étudiées précédemment sont trop coûteuses dans ce contexte, et il est par exemple plus approprié d'utiliser une couche linéaire constituée d'une simple permutation de bits. Une conséquence d'un tel choix est cependant qu'une partie de la diffusion doit maintenant être assurée par les boîtes S elles-mêmes ; on peut ainsi définir une notion de *branch number* pour les boîtes S similaire à celle utilisée pour les applications linéaires.

Pour obtenir une fonction de tour de SPN efficace sur de petites architectures, nous avons construit une boîte S sur huit bits avec un BN de trois qui s'implémente efficacement « en tranche », et qui peut être couplée avec une permutation elle aussi facile d'implémentation. La structure obtenue possède une régularité qui rend l'analyse de certaines propriétés de sécurité relativement aisée. Une caractéristique intéressante de notre fonction de tour est aussi qu'elle n'est pas trop coûteuse à protéger contre un certain nombre d'attaques physiques, celles-ci représentant une menace particulièrement pertinente dans le contexte de la cryptographie embarquée.

2.2.3 PuppyCipher et CoureurDesBois : deux primitives prouvablement incompressibles [FKKM16]

Nous décrivons ici des travaux communs avec Pierre-Alain Fouque, Paul Kirchner et Brice Minaud, publiés à ASIACRYPT 2016.

L'idée de la cryptographie en boîte blanche a été introduite par Chow *et al.* à SAC 2002 [CEJvO02], et correspond à l'objectif de fournir via un « compilateur boîte blanche » une implémentation complète d'un algorithme symétrique, par exemple AES, déjà paramétré par une clef, tout en rendant celle-ci difficile à extraire. La réalisation d'un tel objectif est complexe, et l'essentiel des propositions jusqu'à ce jour ont été cassées, voir par ex. [Gil16] pour une revue. À SAC 2013, Delerablée *et al.* ont proposé plusieurs modèles plus ou moins exigeants permettant de formaliser les objectifs de sécurité d'implémentations boîte blanche [DLPR13]. Le modèle le plus atteignable correspond à une notion d'*incompressibilité* : face à une implémentation d'une certaine taille, un adversaire doit être incapable de fournir une implémentation significativement plus petite qui est fonctionellement équivalente avec très forte probabilité. Cet objectif est relativement facile à atteindre, mais ne satisfait pas nécessairement tous les emplois qu'on pourrait souhaiter faire de la cryptographie boîte blanche, qui restent au demeurant vagues. Certaines propositions récentes d'algorithmes en boîte blanche ciblent (des variantes proches de) ce modèle : les instantiations boîte blanche de la construction ASASA de Biryukov *et al.* [BBK14], largement attaquées [DDKL15, MDFK15], et la famille SPACE de Bogdanov et Isobe [BI15].

Nos travaux ont consisté à raffiner le modèle d'incompressibilité de Delerablée *et al.* en définissant des notions d'incompressibilité faible et forte, la nuance correspondant aux hypothèses faites sur les stratégies des adversaires, puis de proposer deux familles de constructions prouvablement incompressibles par rapport à ces modèles. La première, PuppyCipher, est une famille de chiffrements par bloc de tailles d'implémentation variables, prouvable dans le modèle faible. La seconde, CoureurDesBois, est une famille, également de taille variable, de fonctions pouvant être utilisées comme générateurs de clef, prouvable dans le modèle faible et dans le modèle fort. La base de nos algorithmes est semblable à celles des précédentes propositions de Biryukov *et al.* et Bogdanov *et al.* : elle consiste à utiliser des tables de taille importante remplies de données pseudo-aléatoire et à forcer leur accès en des points a priori imprévisibles pour être capable de chiffrer un message aléatoire. L'incompressibilité vient du fait qu'un adversaire « oubliant » une partie significative des tables se trouverait forcé d'en deviner à nouveau une certaine partie non négligeable pour correctement implémenter l'algorithme sur la plupart des entrées. Nos preuves consistent à montrer que ceci est effectivement le cas pour nos structures. Enfin, notons qu'en plus d'être dotées de preuves d'incompressibilité, nos constructions sont relativement efficaces, et notamment nettement plus performantes que les propositions de Bogdanov et Isobe.

2.3 Nouvelles attaques sur la fonction de hachage SHA-1

La fonction de hachage SHA-1 a été développée par la NSA en 1995 [NIS95], sur la base d'un algorithme datant de 1993 [NIS93], et a été jusqu'à 2002 l'unique standard du NIST pour le hachage cryptographique. En tant que telle, elle fut largement déployée et consiste donc une cible de choix pour les cryptanalystes.

2.3.1 Collisions à initialisations libres pour SHA-1 [KPS15, SKP16]

Nous décrivons conjointement deux articles écrits avec Marc Stevens et Thomas Peyrin, publiés à CRYPTO 2015 [KPS15] et EUROCRYPT 2016 [SKP16].

Les premières attaques en collision sur une version non-réduite de SHA-1 ont été présentées par Wang *et al.* à CRYPTO 2005 [WYY05a]. Bien que, comme souvent en cryptanalyse, de complexité trop élevée pour être effectivement exécutées, ces attaques ont eu un impact important sur l'étude des fonctions de hachage durant la décennie qui a suivie. Elles ont notamment poussé à l'organisation d'une compétition internationale pour le choix d'un nouveau standard, qui a vu Keccak devenir SHA-3 en 2015 [NIS15b].

Dans ces travaux, nous avons souhaité obtenir la première attaque explicite sur une version non-réduite de SHA-1. La complexité attendue d'une attaque complète à la manière de Wang étant d'environ 2^{61} appels à la fonction de compression de SHA-1, suivant les travaux de Stevens d'EUROCRYPT 2013 [Ste13], nous avons plutôt choisi comme objectif le calcul de collisions *à initialisation libre*, qui offrent une liberté supplémentaire à l'attaquant et permettent en théorie de baisser le coût de celles-ci.

Du point de vue de la cryptanalyse, nos travaux ont donc consisté à adapter les attaques de Wang *et al.* à un modèle à initialisation libre, ce qui dans notre cas revient essentiellement à décaler l'initialisation de l'état interne SHA-1 de quelques tours tout en s'assurant que les conditions nécessaires à une collision soient préservées. Les attaques de ce type étant aussi relativement lourdes à effectuer, nous avons aussi amélioré certaines phases de la génération et de l'implémentation de l'attaque. Une autre part importante de ces travaux a consisté à implémenter efficacement l'ensemble de l'attaque, notamment en utilisant des cartes graphiques pour le calcul de la partie la plus coûteuse. Ceci a nécessité un certain soin lors du développement de l'attaque, entre autres choses pour que sa structure soit adaptée au modèle de calcul imposé par les cartes graphiques.

Nos résultats ont consisté en une collision à initialisation libre pour 76 tours sur 80 de SHA-1 avec un coût d'environ $2^{50.25}$ appels à la fonction de compression sur cartes graphiques, ainsi qu'en une attaque similaire sur la fonction complète pour un coût d'environ $2^{57.5}$. Dans le dernier cas, ceci correspond à environ dix jours de calcul sur soixante-quatre cartes graphiques récentes, ce qui est raisonnablement à la portée d'un adversaire moyen.

2.3.2 Attaques en préimages jusqu'à 62 tours de SHA-1 [EFK15]

Nous concluons en décrivant des travaux menés avec Thomas Espitau et Pierre-Alain Fouque, publiés à CRYPTO 2015.

Si la dernière décennie a permis de développer des attaques en collisions puissantes sur SHA-1, bien que seulement à la limite supérieure de la faisabilité, nous ne connaissons toujours pas d'attaque en préimage même complètement théorique sur la fonction complète. En 2012 Knellwolf et Khovratovich ont présenté à CRYPTO une vision différentielle des attaques en préimages utilisant les techniques de rencontre par le milieu [KK12]. Ceci leur a permis à l'époque d'atteindre le plus grand nombre de tours attaqués pour SHA-1, à 57 sur 80. Dans nos travaux, nous avons étendu cette approche différentielle par l'utilisation de différentielles d'ordre supérieur. Ceci permet de trouver de meilleures partitions pour une attaque par le milieu, dans le sens où elles couvrent plus de tours, au prix d'une phase de rencontre plus complexe, rendant la technique surtout utile pour des attaques de coût élevé. De cette façon, nous avons pu augmenter le nombre de tours attaqués de 5, atteignant 62 tours avec une attaque de très grosse complexité estimée à $2^{159.3}$. Nous avons aussi pu baisser la complexité des attaques à nombre de tours équivalents par rapport à Knellwolf et Khovratovich, attaquant 58 tours pour un coût de $2^{157.9}$ contre 57 tours à $2^{158.8}$. Enfin, nous avons appliqué les mêmes techniques avec un succès modéré sur les fonctions de la famille BLAKE, améliorant quelque peu le nombre de tours attaqués en préimage, avec cependant un gain seulement marginal sur la recherche exhaustive.

A Mes publications

A.1 Article de journal

[MDFK] Key-Recovery Attacks on ASASA. B. Minaud, P. Derbez, P.-A. Fouque, P. Karpman. (En soumission, invité au J. Cryptology)

A.2 Articles de conférences

[FKKM16] Efficient and Provable White-Box Primitives. P.-A. Fouque, P. Karpman, P. Kirchner, B. Minaud. (ASIACRYPT 2016)

[SKP16] Freestart Collision for Full SHA-1. M. Stevens, P. Karpman, T. Peyrin. (EUROCRYPT 2016)

[MDFK15] Key-Recovery Attacks on ASASA. B. Minaud, P. Derbez, P.-A. Fouque, P. Karpman. (ASIACRYPT 2015)

[Kar15] From Distinguishers to Key Recovery : Improved Related-Key Attacks on Even-Mansour. P. Karpman. (ISC 2015)

[KPS15] Practical Free-Start Collision Attacks on 76-step SHA-1. P. Karpman, T. Peyrin, M. Stevens. (CRYPTO 2015)

[EFK15] Higher-Order Differential Meet-in-the-middle Preimage Attacks on SHA-1 and BLAKE. T. Espitau, P.-A. Fouque, P. Karpman. (CRYPTO 2015)

[AFK14] Diffusion Matrice from Algebraic-Geometry Codes with Efficient SIMD Implementation. D. Augot, P.-A. Fouque, P. Karpman. (SAC 2014)

[GKN$^+$14] Analysis of BLAKE2. J. Guo, P. Karpman, I. Nikolić, L. Wang, S. Wu. (CT-RSA 2014)

[FK13] Security Amplification against Meet-in-the-Middle Attacks Using Whitening. P.-A. Fouque, P. Karpman. (IMACC 2013)

A.3 Prépublication

[KG16] The Littlun S-box and the Fly block cipher. P. Karpman, B. Grégoire.

Part One

Preliminaries

Overview

The first part of this thesis serves as an introduction to the other two and to some of the main topics of symmetric-key cryptography. The first chapter introduces block ciphers while the second introduces hash functions. These two objects are the main focus of our work.

Contents

3 Block ciphers basics

4 Hash functions basics

Chapter 3

Block ciphers basics

1 Block ciphers

A *block cipher* is a family of injective mappings over finite domains and co-domains, indexed by a finite set of *keys*. In this manuscript, we will only consider ciphers with domains and co-domains of identical sizes, with all arguments taken to be binary strings of a fixed length. Hence, a block cipher is a mapping $\mathcal{E} : \{0,1\}^{\kappa} \times \{0,1\}^{n} \rightarrow \{0,1\}^{n}$ such that for all $k \in \{0,1\}^{\kappa}$, $\mathcal{E}(k,\cdot)$ is a permutation. We call κ the *key size* and n the *block size* of $\mathcal{E}$. Typical parameter sizes are $\kappa \in \{64, 80, 128, 192, 256\}$ and $n \in \{64, 128, 256\}$, though 64 and 80-bit keys are now considered to be too short to provide adequate security. We usually require $\mathcal{E}$ and its inverse $\mathcal{E}^{-1}$ to be efficiently computable, though depending on the intended application, it may be enough for only one of these to be efficient.

The most immediate purpose of block ciphers is to provide confidentiality of communications. Assuming that two parties A and B have been able to share a secret k, they can then use k as the key input to the same block cipher to send encrypted messages $c := \mathcal{E}(k,p)$, $c' := \mathcal{E}(k,p')$, etc. The non-key input to $\mathcal{E}$ is generally called the *plaintext*, and the output of $\mathcal{E}$ is called the *ciphertext*.

If $\mathcal{E}$ is such that the permutations $\mathcal{E}(k,\cdot)$ are hard to invert when k is unknown, A and B may suppose that a secure channel of communication between them consists in injecting their messages to strings $m_0||m_1||\ldots||m_\ell$ of sizes multiple of n and sending encrypted messages $\mathcal{E}(k,m_0)||\mathcal{E}(k,m_1)||\ldots||\mathcal{E}(k,m_\ell)$. There are two major problems with this scheme, however, regardless of the security of the block cipher: first, the scheme is not *randomised*, *i.e.* encrypting the same plaintext twice always results in the same ciphertext. An eavesdropper on the channel between A and B, *i.e.* a "passive adversary", can thus detect when identical message blocks have been sent; second, the communication is not authenticated. An active adversary on the channel may delete or modify some of the blocks of a message, append to a message some blocks from a previous message, or add randomly generated blocks. All of this can be done without A and B noticing that someone is maliciously tampering with the channel.

Problems such as the ones above are solved by designing secure *modes of operation.*

We do not study this topic in this thesis, but we mention some elements related to modes in Section 3.

2 Security of block ciphers

We keep this section relatively informal. Our goal is to be able to specify what it means for $\mathcal{E}$ to be a good block cipher from a practical point of view. We start nonetheless by defining the useful notion of *ideal block cipher*.

Definition 3.1 (Ideal block cipher)**.** An *ideal block cipher* $\mathcal{E}$ is a mapping $\{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ s.t. all the permutations $\mathcal{E}(k,\cdot)$ are drawn independently and uniformly at random among the permutations of $\{0,1\}^n$.

This definition intuitively corresponds to the best we can achieve from the definition of a block cipher. For small values of n, *e.g.* 20, one can implement ideal block ciphers by using an appropriate shuffling algorithm, such as the one variously attributed to Fisher, Yates, Knuth, etc. [FY48]. As this algorithm requires $\mathcal{O}(2^n)$ setup time and memory per key, it is impractical for cryptographically common block sizes $n \geq 64$. Even for small values of n, the shuffling requires a considerable amount of key-dependent randomness, which may be something hard to provide. All of this leads to the fact that we are forced most of the time to use “approximations” of ideal block ciphers. A useful, mostly theoretical, way of quantifying the security of a specific block cipher is then to measure “how far” it is from being ideal. Informally, this is done by upper-bounding the *advantage* over a random answer that any adversary with some bounded resources has of distinguishing whether he is given black-box access to a randomly-drawn permutation or to an instance of the block cipher with a randomly chosen key. This statement can be made more precise in the form of the following definition, similar to the one that can be found *e.g.* in [BKR00]:

Definition 3.2 (Pseudo-random permutations (PRP))**.** We consider a block cipher $\mathcal{E}$ of key size κ and block size n. We write Π_{2^n} for the set of permutations on binary strings of length n; $x \xleftarrow{\$} \mathcal{S}$ the action of drawing x uniformly at random among elements of the set $\mathcal{S}$; $\mathcal{A}^f$ an algorithm with oracle, black-box access to the function f and which outputs a single bit. Then we define the *PRP advantage* of $\mathcal{A}$ over $\mathcal{E}$, written $\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A})$ as:

$$\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A}) = |\Pr[\mathcal{A}^f = 1 \mid f \xleftarrow{\$} \Pi_{2^n}] - \Pr[\mathcal{A}^f = 1 \mid f := \mathcal{E}(k,\cdot), k \xleftarrow{\$} \{0,1\}^\kappa]|.$$

The *PRP security* of $\mathcal{E}$ w.r.t. the *data complexity* q and *time complexity* t is:

$$\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(q,t) := \max_{\mathcal{A} \in \mathrm{Alg}^{f \backslash q, \mathcal{E} \backslash t}} \{\mathbf{Adv}_{\mathcal{E}}^{\mathrm{PRP}}(\mathcal{A})\}.$$

Here, $\mathrm{Alg}^{f \backslash q, \mathcal{E} \backslash t}$ is the set of all algorithms $\mathcal{A}$ with oracle access to f that perform at most q oracle accesses and which run in time $\mathcal{O}(t)$, with the time unit being the time necessary to compute $\mathcal{E}$ once.

Definition 3.2 is quite useful in some contexts, for instance to prove that a construction using a block cipher is not significantly less secure than the latter. This is typically done by defining an advantage function similar to PRP security for the higher-level construction and by showing that it is not more than a reasonable function of the PRP security of the block cipher. For instance, the high-level construction studied in [BKR00] is CBC-MAC.

However, this definition is not constructive, in the sense that it does not provide any efficient way of computing the PRP security of a block cipher in general. A major topic in symmetric cryptography is to analyse explicit instances of block ciphers in order to assess their concrete security against attacks. In the language of Definition 3.2, this consists in finding algorithms for which q, t and the PRP advantage is known. Any such attack on a block cipher $\mathcal{E}$ allows to lower-bound its PRP-security at a given point (q,t). In reality, cryptanalysis results on block ciphers are seldom as easily expressed as what Definition 3.2 may lead us to believe; practically important characteristics of an attack are also its memory complexity, distinguishing between its online and offline time complexity, whether it applies equally well to all keys or if it is only successful for some "weak" subset thereof, whether it also recovers k when f was instantiated from $\mathcal{E}$, or an algorithm equivalent to $\mathcal{E}(k,\cdot)$, etc. We devote the remainder of this section to sketching some typical elements of attacks on block ciphers.

2.1 Distinguishers and attacks

Many concrete attacks on block ciphers use *distinguishers* as their basis. These can be defined as algorithms using reasonable resources which have a non-negligible advantage according to Definition 3.2. There is no easy answer as to what "reasonable" and "non-negligible" should mean in the context of actual cryptanalysis, as the key and block size of a specific cipher are fixed values. While some ciphers or potential distinguishers may be parameterised in a way that helps to make the notion meaningful, this does not have to be the case. Sometimes, one is easily convinced by the performance of an algorithm so that there is consensus that it can be called a distinguisher, *e.g.* distinguishing $\mathcal{E}$ of key and block size 128 with $q = 2$, $t = 2^{20}$, probability ≈ 1, while some other times the picture is much less clear, *e.g.* $q = t = 2^{120}$ and probability ≈ 1. We ignore this issue altogether and assume that all the attacks of this chapter are consensual.

2.1.1 Classes of distinguishers for block ciphers

We now briefly describe two types of distinguishers, which exploit "non-ideal" behaviours of different nature.

We start with *differential distinguishers*, which are part of the broader class of *statistical* distinguishers. The basic idea of the latter is to define events which have different probability distributions for the target, *i.e.* the block cipher $\mathcal{E}$ than for a random permutation drawn from Π_{2^n}. Running the distinguisher then consists in collecting a certain number of samples obtained through an oracle and deciding from which distribution those are the most likely to have been drawn. A differential distinguisher instantiates this idea by considering a certain type of statistical events. Another major class of statistical distinguishers is the one of *linear distinguishers*.

Consider a block cipher $\mathcal{E}$; a *differential* for $\mathcal{E}$ is a pair $(\Delta \neq 0, \delta)$ of input and output *differences*, according to some group law $+$. In the huge majority of cases, $+$ is the addition in $\mathbf{F}_2^n$, *i.e.* the bitwise exclusive OR (XOR); in this case we usually use the alternative notation $\oplus$. Sometimes, $+$ is taken to be the addition in $\mathbf{Z}/2^n\mathbf{Z}$, and some other times differences according to the two laws may be jointly used. A *differential pair* for the differential (Δ, δ) and the key k is an ordered pair $((p, c), (p', c'))$ of plaintexts and their corresponding ciphertexts p, $c := \mathcal{E}(k, p)$, p', $c' := \mathcal{E}(k, p')$ such that $p - p' = \Delta$, $c - c' = \delta$. When differences are over $\mathbf{F}_2^n$, subtraction coincides with addition and the pair can be unordered. We consider this to be the case in the remainder of this description.

We call *differential probability* of a differential w.r.t. a permutation $\mathcal{P}$ the probability of obtaining a differential pair for $\mathcal{P}$: $\mathrm{DP}^{\mathcal{P}}(\Delta, \delta) := \Pr_{p \in \{0,1\}^n}[\mathcal{P}(k, p) \oplus \mathcal{P}(k, p \oplus \Delta) = \delta]$. The most important characteristic of a differential for a block cipher is its *expected differential probability*, which is simply its differential probability for $\mathcal{E}(k, \cdot)$ averaged over k: $\mathrm{EDP}^{\mathcal{E}}(\Delta, \delta) := 2^{-\kappa} \sum_{k \in \{0,1\}^\kappa} \mathrm{DP}^{\mathcal{E}(k,\cdot)}(\Delta, \delta)$. A common assumption is that for most keys and differentials, the fixed-key DP is close to the average EDP. The DP of a random differential w.r.t. a random permutation can be approximated by a Poisson distribution: the approximate number of differential pairs is $\sim \mathrm{Poi}(2^{-1})$, of mean and variance 2^{-1}, see [DR07], using an earlier result [O'C95]. As there are 2^{n-1} possible pairs, the expected DP is thus 2^{-n}; note however that the DP is in fact restricted to values multiple of 2^{-n+1}. For a distinguisher on $\mathcal{E}$ to be of any use, we need its EDP not to be equal to 2^{-n}. If it is far enough from that, *e.g.* $2^{-3n/4}$, we usually make the simplifying working hypothesis that all DPs are equal to their expected value, or rather the nearest possible values. In such a case, using the distinguisher consists in collecting $\propto 1/\mathrm{EDP}^{\mathcal{E}}(\Delta, \delta)$ plaintext pairs verifying the input difference and counting how many of them verify the output difference. We decide that we are interacting with $\mathcal{E}$ if and only if this is one or more.

Another kind of distinguishers is based on *algebraic* representations of block ciphers. One can always redefine a block cipher $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ as an ordered set of functions $\mathcal{F}_i : \{0,1\}^{\kappa+n} \to \{0,1\}$ that project $\mathcal{E}$ on its i^{th} output bit: $\mathcal{E} \equiv (\mathcal{F}_0, \ldots, \mathcal{F}_{n-1})$. The $\mathcal{F}_i$s can be understood as Boolean functions $\mathbf{F}_2^{\kappa+n} \to \mathbf{F}_2$ which are themselves in bijection with elements of $\mathbf{F}_2[x_0, x_1, \ldots x_{\kappa+n-1}]/\langle x_i^2 - x_i \rangle_{i<\kappa+n}$, *i.e.* multivariate polynomials in $\kappa + n$ variables over $\mathbf{F}_2$. The polynomial to which a Boolean function is mapped is called its *algebraic normal form* (ANF); the ANF of $\mathcal{E}$ is the ordered set of ANFs of its projections.

An important characteristic of an ANF is its degree, which can be used to define simple yet efficient distinguishers. The degree of the ANF of an n-bit permutation is at most $n - 1$, and it is expected of a random permutation to be of maximal degree. If a block cipher has degree $d < n - 1$, it can be distinguished by differentiating it on enough values. This simply requires to evaluate the oracle on 2^{d+1} properly chosen values — essentially a cube of dimension d— and to sum them together. If the result is all-zero, the oracle is likely to be of degree less than d and is hence assumed to be $\mathcal{E}$; if this is not the case, it is necessarily of degree strictly more than d and hence assumed to be a random permutation.

2.1.2 Extending distinguishers to key-recovery attacks

In the definition of PRP security, we were content with the notion of distinguisher. In actual attacks on block ciphers, however, the end objective would ideally be to recover the unknown key used by the oracle. The context of a concrete attack is also different from a PRP security game as one usually knows that he is interacting with a specific cipher $\mathcal{E}$ and not a random permutation, and there is seemingly no point in running a distinguisher at all. Despite these observations, distinguishers are in fact useful in many cases, and are often at the basis of key-recovery attacks, although it should be noted that not all such attacks are based on distinguishers. We briefly explain the basic idea of this conversion; to do this, we need to assume that $\mathcal{E}$ possesses a certain structure, which is in fact very common.

An *iterative block cipher* is a cipher $\mathcal{E}$ that can be described as the multiple composition of a *round function* $\mathcal{R}$, possibly with additional composition of an initialisation or finalisation function that we ignore here: $\mathcal{E} \equiv \mathcal{R} \circ \cdots \circ \mathcal{R}$. Let us assume that a "full" application of $\mathcal{E}$ is made of r rounds. A distinguisher-based key-recovery attack first consists in finding a distinguisher on a *reduced-round* version of $\mathcal{E}$ made of the composition of $d < r$ round functions. The next step simply consists in querying the oracle on inputs verifying the distinguisher condition, for instance plaintexts with difference Δ, in a differential case; as one obtains ciphertexts encrypted with the full block cipher, one is not expected to be successful when running the distinguisher on these values. The main idea comes from the third step, where one guesses values for part of the unknown key k of $\mathcal{E}$ which allow him to partially decrypt the ciphertexts by $r - d$ rounds. Then, if the guess was correct, he obtains ciphertexts for the cipher reduced to d rounds, on which the distinguisher is expected to be successful. On the other hand, if the guess was incorrect, one may assume that decrypting with invalid keys amounts to encrypting with random keys. Hence, the inputs to the distinguishers may be seen as ciphertexts encrypted with a cipher using $r + (r - d) = 2r - d$ rounds, and the distinguisher should fail. The overall approach thus gives a method to verify a guess for part of the unknown key.

The procedure as described above calls for several comments. First, the cost of guessing part of the key obviously adds to the complexity of the distinguisher, so the overall complexity of the attack is higher than the latter. Thus, only distinguishers of low-enough complexity may be converted to key-recovery attack. Second, one may only use distinguishers that allow to verify a guess on *part* of the key. For instance, one can use distinguishers that are "local", in the sense that they only base their answers on the value of part of the state of the cipher. For such distinguishers, it is enough to guess the part of the key necessary to decrypt the relevant part of the state. Finally, the part of the key that was not recovered thanks to the distinguisher can be obtained by different means. For instance, another distinguisher may be used which recovers another part of the key, or it can simply be guessed exhaustively.

2.1.3 Attack models

So far we have discussed how to express the security of block ciphers and how to attack them in a rather simple case when one is given access to a single “secret” oracle. This setting may be generalised in some ways, for instance by providing more than one oracle. One such common generalisation is to attack a cipher in the *related-key model*, where one is given oracle access to $\mathcal{E}(k,\cdot)$, $\mathcal{E}(\phi(k),\cdot)$, with $\phi(\cdot)$ one or more mappings on the key space. A crucial observation in this case is that ϕ cannot be arbitrary, as some mappings may be so powerful that they allow to attack every cipher; speaking of the security of $\mathcal{E}$ in such a model is then meaningless. We will mention this matter again in Chapter 5.

The potential problems arising from ill-defined related-key models are a useful reminder that attacks should be specified in a well-founded way. While some models are more powerful than others and may then be considered less relevant to the actual security of block ciphers, a main concern about a new model should be whether it trivially allows to attack any algorithm.

3 Using block ciphers

We mentioned in the beginning of this chapter that block ciphers do not provide adequate security if they are used directly and not as part of a wider construction. One calls *mode of operation* such a construction that results in a functional cryptosystem. We do not describe modes in this section, but reiterate from the introduction the essential conditions that they must meet.

A foremost requirement is that a mode be randomised, in the sense that encrypting the same message with the same key twice should not result in the same ciphertext. This can be enforced through the notion of *indistinguishability* in a *chosen-plaintext attack* scenario (IND-CPA) and its close relatives. Roughly, this is defined thanks to the following process: an adversary is given a black-box access to the encryption procedure of a certain cryptosystem, then prepares two messages m_0 and m_1 and sends them to an oracle. This oracle randomly selects one of the two messages and returns its associated ciphertext. Finally, the adversary is again given access to the cryptosystem and then tries to guess which message was encrypted. The cryptosystem is IND-CPA if no adversary with appropriately bounded resources is successful in his guess with a non-marginal advantage. It is clear in particular that a deterministic cryptosystem cannot be secure according to this definition.

We also already mentioned that a cryptosystem should provide authentication of the communicating parties. This is either done directly by the mode of operation, which is then called an *authenticated encryption* mode, or AE, or by combining an encryption-only mode with a *message authentication code* (MAC) in an appropriate way. The current trend is to favour the former approach, as it tends to lead to more efficient schemes.

Chapter **4**

Hash functions basics

1 Hash functions

A hash function is a mapping from an arbitrary, non-necessarily finite set to a finite set of small size. As in the previous section, we restrict our presentation to the binary case; hence, we define hash functions as mappings from *messages*: bit strings of arbitrary length, to *digests* or *hashes*: strings of a fixed predetermined length: $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n$ for some integer n. Many hash functions do not strictly adhere to this definition, as they upper-bound the length of their inputs by a large integer such as 2^{64}. This currently has no impact in practice, as all potential inputs to a hash function are much shorter than these upper limits; it is of course debatable whether this will always remain the case. The typical output length of hash functions is of a few hundred bits, often a multiple of thirty-two; current and past hash functions have $n \in \{128, 160, 224, 256, 384, 512\}$.

A *cryptographic* hash function, or simply hash function is a function $\mathcal{H}$ that verifies a certain number of security properties which express the difficulty of computing inputs to $\mathcal{H}$ that verify some conditions. There are three "classical" security properties that must be met, at least to some extent by any secure hash function. We only give here rather informal definitions similar to the ones found *e.g.* in [MvOV96, Chapter 9]. The specific kinds of messages involved in the security properties are the following:

Definition 4.1 (Preimage)**.** A preimage of t for the function $\mathcal{H}$ is a message m such that $\mathcal{H}(m) = t$.

Definition 4.2 (Second preimage)**.** A second preimage of $t := \mathcal{H}(m)$ for the function $\mathcal{H}$ is a message $m' \neq m$ such that $\mathcal{H}(m') = \mathcal{H}(m) = t$.

Definition 4.3 (Collision)**.** A collision for the function $\mathcal{H}$ is a pair of messages $(m, m' \neq m)$ such that $\mathcal{H}(m) = \mathcal{H}(m')$.

A function $\mathcal{H}$ is *(second) preimage-resistant* if there is no way of finding a preimage for a random target t that is more efficient than for a function whose outputs are drawn uniformly at random and independently of each other, *i.e.* computing $\mathcal{H}(x)$ for about 2^n

different inputs x. It should be noted that anyone can easily compute $\mathcal{H}(x) = y$ for some x and learn a preimage of y. The notion of preimage resistance is thus only meaningful if associated with random targets.

Similarly, a function $\mathcal{H}$ is collision-resistant if one cannot find collisions for $\mathcal{H}$ more efficiently than for a random function, *i.e.* computing $\mathcal{H}(x)$ for about $2^{n/2}$ distinct inputs x.

Degenerate algorithms. One may notice that for any function $\mathcal{H}$, there are algorithms whose time and space complexity are negligible and that return a single constant collision (m, m'). As collisions, unlike preimages, are not associated with targets, these algorithms cannot be easily excluded by demanding that they produce collisions related to a specific input. However, such degenerate algorithms are ignored in practice and the cost of finding a collision for a random function is always assumed to be in the neighbourhood of $2^{n/2}$ calls to the function.

The complexities associated with these security notions correspond to the ones of the best known generic algorithms that can find messages with the desired properties with high probability for any function. An attack on a specific hash function is an algorithm that can achieve one of the above tasks significantly more efficiently than the best generic algorithm. By definition, this violates the associated security property. Above, we define the complexity of these generic algorithms in terms of calls to $\mathcal{H}$. In practice, it may be necessary to adjust this to a finer granularity; for instance, for the Merkle-Damgård hash functions that we describe in Section 2, it makes more sense to evaluate the complexity in terms of calls to what will be defined as a *compression function*.

1.1 Applications of hash functions

The nature of the properties used to define security for cryptographic hash functions is based on requirements from the various cases where they may be used in concrete cryptosystems or protocols. Depending on the situation, resistance w.r.t. all properties from above may not be necessary; for instance, it may be the case that collision resistance is not required, or that, say, only second preimage resistance is relevant. However, a hash function is usually expected to be used in a certain number of settings, and it is thus understandingly expected to be secure against all attacks.

We only briefly sketch some possible uses of hash functions here, as an illustration.

Hash and sign signatures. Hash and sign signatures are one of the main settings where hash functions may be employed; the objective is, given a digital signature algorithm $\mathcal{S}$, to efficiently and securely sign *long* messages. Directly doing so using $\mathcal{S}$ is usually not an option, because signature algorithms are rather slow, especially compared to hash functions which are typically at least 500 ~ 1000 times faster, and may not behave well on long messages. A useful alternative is instead to first compute the digest of the message m that needs to be signed and to sign the digest instead of the whole message. One then gives an output of the form $(m, \mathcal{S}(k, \mathcal{H}(m)))$, with k a key for $\mathcal{S}$.

It is obvious that for such a scheme to work, the function $\mathcal{H}$ needs to be at least resistant to second preimages. If this were not the case, an adversary could intercept a message m signed by A, replace m by m' such that the two messages collide through $\mathcal{H}$, and claim that A signed m'. Collision resistance is also important for similar reasons.

Password hashing. Password hashing is another common setting where hash functions may be useful. In this case, we assume that a certain entity wishes to allow users to authenticate themselves through passwords. Consequently, it must remember the valid password of each user. As there would be obvious security issues with the entity storing the passwords themselves, an idea is to instead store their images through a hash function $\mathcal{H}$. Thus, if an adversary finds the database of users and their associated hashed passwords, he would be unable to find the passwords or some equivalent input, provided that $\mathcal{H}$ is preimage-resistant.

In this setting, second preimage and collision resistance are not strictly needed. However, it should be noted that even when built from a secure hash function, the scheme just described above has severe issues, whose details are beyond the scope of this introduction.

Hash-based signatures. A less common use of hash functions is to utilise them to directly define signature schemes, see *e.g.* [Mer87]. We will not describe such schemes in detail here, but it is interesting to mention a few of their specificities. The main idea of the schemes is that the signing party makes some digests public while keeping their inputs secret; signing a message then consists in selectively revealing some of these inputs. Thus, being able to compute preimages for the hash function breaks the scheme, but collisions are not a threat. A distinguishing feature of hash-based signatures is that the hash function may only need to be used on inputs of short, possibly fixed size.

Message authentication codes. The last possible usage of hash functions that we describe here is the building of *message authentication codes*, or MACs, which can somehow be seen as the keyed variants of hash functions. A MAC $\mathcal{T}$ takes as input a key k and a message m and outputs a tag $t \coloneqq \mathcal{T}(k, m)$. If an adversary does not know the key, it should be hard for him to find a valid (message, tag) pair for $\mathcal{T}(k, \cdot)$, with the message either being of his choosing or imposed by a challenger. These notions correspond to *existential* (for the former) and *universal* (for the latter) *forgery*. Of course, it is also highly necessary that no tag collisions occur, whether or not these happen on specific messages requested by an adversary.

Hash functions seem to be good candidates to build MACs, and indeed generic hash function based constructions such as HMAC [BCK96] are popular. However, the exact security of these constructions is not always easy to establish, and there are usually faster alternatives such as MACs based on universal/polynomial hash functions (see *e.g.* [BHK+99]).

2 Merkle-Damgård hash functions

One of the first frameworks for hash function design to have been developed is the so-called Merkle-Damgård construction, that was independently developed by Merkle and Damgård in 1989 [Mer89, Dam89]. The idea of this construction is to make the arbitrary-length inputs to a hash function manageable by defining the latter as the iteration of a *compression function* with a small fixed-size (co-)domain; we say that the construction is a *domain extender* for a compression function. This approach makes the overall design much easier, but without *a priori* ensuring that the resulting function will be secure. The main contribution of Merkle and Damgård in that respect is to give a construction such that the security of the function can be partially reduced to the one of the compression function: they show that for collision attacks, an attack on the hash function can be exploited to build an attack on the compression function. Taking the contrapositive, as long as there are no collision attacks on the compression function, we can be confident that the hash function is secure against this kind of attacks. This is in fact quite similar to building symmetric cryptosystems by combining a secure block cipher and a secure mode of operation.

The construction works as following. A compression function $\mathrm{H} : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$ takes two inputs: a *chaining value* c and a *message block* m, and produces another chaining value as output. The hash function $\mathcal{H}$ associated with H is built by extending the domain of the latter to $\{0,1\}^*$, or rather $\{0,1\}^N$ for a large N, most of the time. This is done by specifying an *initial value* IV for the first chaining value c_0, which is a constant for the hash function, and by defining the image of a message m through $\mathcal{H}$ by the following process:

1. m is padded to a size multiple of the message *block size* b. Various padding rules may be employed, but it is obviously important that they should not introduce trivial collisions. Also, most of the time, the size, usually in bits, of the non-padded message m is included in the padding in one way or the other. This is usually called Merkle-Damgård-strengthening, and it is essential to make many of the common instantiations of the framework secure. Indeed, one issue that could arise in the absence of such strengthening is that fixed-points for the compression function could be used to produce collisions; such fixed-points are easy to build for some popular compression function constructions, including the *Davies-Meyer* construction of the MD-SHA family described in Section 5.

2. The padded message $\mathsf{m}_0||\mathsf{m}_1||\ldots||\mathsf{m}_r$is then iteratively fed to the compression function by having $\mathsf{c}_{i+1} := \mathrm{H}(\mathsf{c}_i, \mathsf{m}_i)$. The digest $\mathcal{H}(m)$ is equal to the last chaining value c_{r+1}.

We give an illustration of this process in Figure 4.1.

We conclude this presentation by sketching the reduction proofs of the construction for collision and first preimage attacks. There can be no such proof for second preimages, as there exists some generic attacks on Merkle-Damgård functions independently of the security of the compression function, and the actual security reached by Merkle-Damgård

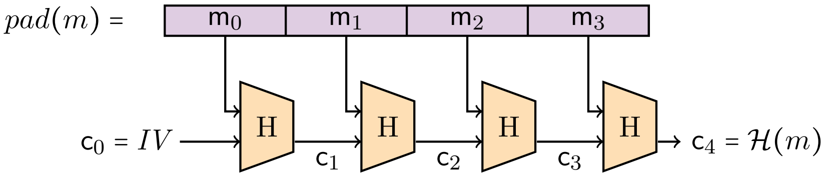


Figure 4.1 – A Merkle-Damgård hash function processing a four-block input. Figure adapted from [Jea].

functions is not clear. We will briefly discuss these in Section 3. It is also important to notice that a collision attack for a compression function H does not necessarily lead to an attack for a Merkle-Damgård function $\mathcal{H}$ based on it. Still, such a collision would violate the security reduction, and thus no formal guarantee could be given anymore on the collision resistance of $\mathcal{H}$. We will discuss such issues in slightly more details in Section 3.

Proposition 4.1. *A collision on a Merkle-Damgård hash function $\mathcal{H}$ implies a collision on its compression function* H*.*

Proof. We assume that Merkle-Damgård strengthening is used, that the message length is appended at the end of the padding, and that it fits in a single message block.

Assume we have m, $m' \neq m$ s.t. $\mathcal{H}(m) = \mathcal{H}(m')$.

Case 1: m and m' have a different length. The last message blocks m_r and $\mathsf{m}'_{r'}$ both include the length of the message, which is different. Thus $(\mathsf{c}_r, \mathsf{m}_r)$ and $(\mathsf{c}'_{r'}, \mathsf{m}'_{r'})$ are distinct and collide through H.

Case 2: m and m' are of the same length. Assume w.l.o.g. that the messages fit on $r + 1$ blocks after padding. Call i the highest block number such that $\mathsf{m}_i \neq \mathsf{m}'_i$. If $i = r$, then $(\mathsf{c}_r, \mathsf{m}_r)$ and $(\mathsf{c}'_r, \mathsf{m}'_r)$ are distinct and collide through H. If $i < r$, either $\mathsf{c}_{i+1} = \mathsf{c}'_{i+1}$, thus $(\mathsf{c}_i, \mathsf{m}_i)$ and $(\mathsf{c}'_i, \mathsf{m}'_i)$ form a valid collision pair for H, or, we have a non-empty sequence of input pairs $(\mathsf{c}_j, \mathsf{m}_j)$ and $(\mathsf{c}'_j, \mathsf{m}'_j)$, $j = i + 1 \ldots r$ such that $\mathsf{m}_j = \mathsf{m}'_j$ for all j. As $\mathsf{c}_{r+1} = \mathsf{c}'_{r+1}$, at least one element of the sequence collides through H. The two input pairs in the first one to do so are different, and thus form a valid collision for H. □

Proposition 4.2. *A preimage on a Merkle-Damgård hash function $\mathcal{H}$ implies a preimage on its compression function* H*.*

Proof. Let m be a message fitting on $r + 1$ blocks after padding s.t. $\mathcal{H}(m) = t$, with t the preimage target. Then $\mathrm{H}(\mathsf{c}_r, \mathsf{m}_r) = k$, and $(\mathsf{c}_r, \mathsf{m}_r)$ is thus a valid preimage input for H. □

3 Refining the security of hash functions

The security notions of collision and preimage resistance can be refined and completed with additional ones. These are not always directly relevant to actual uses of hash functions, but they may nonetheless be useful to evaluate the security of a function in a finer way. We may roughly distinguish between two kinds of additional properties by what they characterize: non-ideal behaviours of certain frameworks for hash function constructions, and non-ideal behaviours of specific instances of hash functions or of their building blocks. To allow for more explicit definitions of these additional properties, it is useful to define a stronger, idealised view of hash functions:

Definition 4.4 (Random oracle). An n-bit *random oracle* $\mathcal{R}$ is a mapping $\{0,1\}^* \rightarrow \{0,1\}^n$ such that for every input x, its image $\mathcal{R}(x)$ is drawn uniformly at random over $\{0,1\}^n$.

According to the way we defined attacks, a random oracle is not vulnerable to them, as only generic algorithms may be used against it. It thus completely captures what the usefulness of hash functions: if a high-level construction is not secure when it is idealised as using a random oracle, that is when analysed in the *random oracle model*, one can never hope to make it so when instantiating the oracle with a concrete hash function. However, even if a construction is secure in such a model, it is not necessarily true that this will still be the case once instantiated.

The Merkle-Damgård functions from Section 2 exhibit some of this *non-ideal behaviours*, in the sense that some properties may be computed more efficiently for Merkle-Damgård functions than for random oracles.

A good example of such a property is the concept of *multicollision*, where one is required to find $r > 2$ messages $m_0, \ldots, m_{r-1}$ whose images through $\mathcal{H}$ are all equal. The generic complexity of this problem on a random oracle is $\mathcal{O}(2^{n \times (r-1)/r})$ calls to $\mathcal{H}$ for an n-bit function, but Joux showed how to find (2^r)-multicollisions in time $\mathcal{O}(r \times 2^{n/2})$ for Merkle-Damgård hash functions [Jou04]. The basic idea used in this attack is that collisions for a Merkle-Damgård function $\mathcal{H}$ can be chained together to lead to an exponentially growing number of distinct messages hashing to the same value. We can easily see this with a small example: assume that an attacker found two distinct colliding messages m_0 and m'_0 of the same length; this can be done generically at a cost of $2^{n/2}$. One then looks for a second collision for the function $\widetilde{\mathcal{H}}$ obtained by replacing the initial chaining value c_0 by $\mathcal{H}(m_0) = \mathcal{H}(m'_0)$; this can again be obtained for a cost of $2^{n/2}$, resulting in m_1 and m'_1. Then, thanks to the chaining property of the Merkle-Damgård construction, we have found four colliding messages $m_0||m_1$, $m'_0||m_1$, $m_0||m'_1$, $m'_0||m'_1$. It is easy to see how this generalises to longer messages, leading to the quoted complexity of $\mathcal{O}(r \times 2^{n/2})$. We can observe that one of the weaknesses of the construction that is exploited here is that a hash collision is also a collision for the *internal state* of the hash function. We will briefly see in Section 4 that increasing the size of the state of a function is indeed a way to make it resistant to such attacks.

Another good example for Merkle-Damgård, which this time directly violates the security property from Definition 4.2, consists in second preimage attacks for long messages. Dean [Dea99], and later Kelsey and Schneier [KS05] showed how one can again exploit the structure of the function and internal collisions to compute a second preimage of a long message more efficiently than with a generic algorithm. The complexity of Kelsey and Schneier's attack to find a second preimage for a message of 2^k *blocks* is $\approx \mathcal{O}(2^{n-k+1})$ calls to $\mathcal{H}$. This means that Merkle-Damgård hash functions are actually inherently insecure, if we adhere strictly to what we stated as security objectives. However, although significant, these attacks remain expensive, especially for messages of usual sizes. As such, they are not usually considered as threatening the practical use of Merkle-Damgård functions, and indeed functions following this framework such as SHA-2 [NIS15a] are still widely used.

We already mentioned in Section 2 that an attack on a compression function makes a security reduction in a Merkle-Damgård construction impossible. Thus it seems natural to also analyse the security of the compression functions themselves. An attack would in this case demonstrate a non-ideal behaviour of the second kind we mentioned above, not targeting a hash function in itself but one of its building blocks. There is a natural way to generalise the security properties expressed in Definition 4.1 ~ Definition 4.3 to this context, leading to the notion of *(semi-)freestart* attacks:

Definition 4.5 (Freestart preimage). A *freestart preimage* for a Merkle-Damgård hash function $\mathcal{H}$ is a pair (i, m) of an IV and a message such that $\mathcal{H}_{\mathsf{i}}(m) = t$, with $\mathcal{H}_{\mathsf{i}}(\cdot)$ denoting the hash function $\mathcal{H}$ with its original IV replaced by i.

Definition 4.6 (Semi-freestart collision). A *semi-freestart collision* for a Merkle-Damgård hash function $\mathcal{H}$ is a pair $((\mathsf{i}, m), (\mathsf{i}, m'))$ of two IV and message pairs such that $\mathcal{H}_{\mathsf{i}}(m) = \mathcal{H}_{\mathsf{i}}(m')$.

Definition 4.7 (Freestart collision). A *freestart collision* for a Merkle-Damgård hash function $\mathcal{H}$ is a pair $((\mathsf{i}, m), (\mathsf{i}', m'))$ of two IV and message pairs such that $\mathcal{H}_{\mathsf{i}}(m) = \mathcal{H}_{\mathsf{i}'}(m')$.

It can be noted that if the two messages of, say, a freestart collision pair are one-block long, the definition above becomes equivalent to the one of a collision on the compression function H used to build $\mathcal{H}$. There is little difference between the two in general, the feature of freestart attacks precisely being that compared to hash function attacks, they may exploit the additional available input offered by the chaining value of the compression function.

Lastly, a somehow loosely defined addition to the security notions presented so far is the concept of distinguishers, which denote non-ideal behaviours of a hash function that are not otherwise captured by the previous definitions. We will not give a precise definition here, as these are made tricky by the unkeyed nature of hash functions. Instead, we just briefly mention an example of such a distinguisher for the compression function of SHA-1.

Jumping ahead, we will see in Part III that the IV and the message block of SHA-1's compression function are made of five and sixteen 32-bit words respectively. In 2003, Saarinen showed that *slid pairs* could be found for this compression function for a cost equivalent to 2^{32} function calls [Saa03]. Such pairs are made of two *IVs* $\mathcal{A}_{0,\ldots,4}$, $\mathcal{A}'_{0,\ldots,4}$ and messages $m_{0,\ldots,15}$, $m'_{0,\ldots,15}$ with $\mathcal{A}'_i = \mathcal{A}_{i-1}$ and $m'_i = m_{i-1}$, such that the pair of outputs of the function called on these inputs also has this property. Although it is not expected of a random function to exhibit such a property, there is no clear way to use it to mount an attack against the main and secondary security properties defined above.

4 Modern hash function frameworks

We have mentioned some generic weaknesses of the Merkle-Damgård framework in Section 3. As a consequence of these, modern hash function designs are usually based on alternative, more secure constructions. We briefly review two of them: the *wide-pipe* variation of Merkle-Damgård, or *chop-MD*, and the *sponge* construction.

Wide-pipe Merkle-Damgård. The wide-pipe construction was introduced in 2005 by Lucks [Luc05] and Coron *et al.* under the name chop-MD [CDMP05]. It is conceptually simple, and consists in using the Merkle-Damgård construction with a compression function of output size larger than the one of the chaining value. If we write $\lfloor\cdot\rfloor_n$ an arbitrary truncation function from $m > n$ to n bits, we may define the n-bit chop-MD construction based on a compression function $\mathrm{H} : \{0,1\}^m \times \{0,1\}^b \to \{0,1\}^m$ as $\lfloor\mathcal{H}(\cdot)\rfloor_n$, with $\mathcal{H}$ a standard Merkle-Damgård function built from H. Some variations are possible, for instance by considering other mappings from m to n bits instead of just a truncation.

One can easily see that a hash collision for such a function does not anymore imply a collision for its internal state. By choosing m to be sufficiently large, for instance taking $m = 2n$, one can achieve generic resistance to, say, multicollisions. In fact, Coron *et al.* proved that this construction is a secure *domain extender for random oracles*, in the sense that if the compression function is a fixed-size random oracle, using it in a chop-MD mode yields a function that is ε-indifferentiable from a random oracle, in the sense of Maurer *et al.* [MRH04], with $\varepsilon \approx 2^{-t}q^2$, $t = m - n$ being the number of truncated bits and q the number of queries to H. In the light of Section 3, this is a very useful result, as it says that unlike plain Merkle-Damgård, no non-ideal behaviour is introduced by the domain-extending construction. Such hash functions are thus expected to behave closely to the random oracles they may be expected to instantiate, as long as their compression functions are "ideal" and that they are not queried too much w.r.t. to the size of their parameters.

Sponge construction. The sponge construction was introduced in 2007 by Bertoni *et al.* [BDPA07]. It is quite distinct from the Merkle-Damgård framework, notably because it does not use a compression function as building block, but a function of equally-sized domain and co-domain. Usually, this function is even taken to be bijective.

The construction in itself is simple. Assume we want to build an n-bit function based on a b-bit permutation P. We define the *rate* r and the *capacity* c as two integers such that $b = r + c$. Then, hashing the message m consists in padding it to a length multiple of r and to process it iteratively in two phases. The *absorbing* phase computes an internal state value $\mathsf{i} := \mathrm{P}(\mathrm{P}(\ldots\mathrm{P}(\mathsf{m}_0||0^c) \oplus \mathsf{m}_1||0^c)\ldots)$. The squeezing phase then produces the n-bit output as $\mathcal{H}(m) := \lfloor\mathsf{i}\rfloor_r||\lfloor\mathrm{P}(\mathsf{i})\rfloor_r||\ldots||\lfloor\mathrm{P}^{n\div r}(\mathsf{i})\rfloor_{n \bmod r}$.

A distinguishing feature of the sponge construction is that the output length of an instance is entirely decorrelated from the size of its building block. Thus, it allows to swimmingly build variable-length hash functions. A given permutation can also be used in different instantiations offering a tradeoff between speed (a larger rate giving faster functions) and security (a larger capacity giving more secure functions).

Bertoni *et al.* also showed in 2008 that similarly as chop-MD, the sponge construction instantiated with a random function or permutation is ε-indifferentiable from a random oracle of the same output size, with $\varepsilon \approx 2^{-c}q^2$ [BDPA08]. To achieve the classical security requirements of a hash function, it is thus optimal to take $c = 2n$.

One of the best examples of a sponge function is Keccak [BDPA11], which became the SHA-3 standard in 2015 [NIS15b].

5 The MD-SHA family

We now briefly present the "MD-SHA" hash function family, both because it is of somewhat historical importance and because, the function studied later in this manuscript, SHA-1, is one of its members.

The family originated in 1990 with the MD4 function, introduced by Rivest [Riv90]. An attack on a reduced version was quickly found by den Boer and Bosselaers [dB91], and Rivest proposed MD5 as a strengthened version of MD4 [Riv92]. Bosselaers proposed RIPEMD in 1992 as another attempt to strengthen MD4 [BP95, Chapter 3], and the NSA did the same the following year by introducing the first generation of the SHS/SHA algorithms [NIS93]. Both algorithms were quickly modified in 1996 and 1995 respectively [DBP96, NIS95]. Some later algorithms such as SHA-2 [NIS15a], introduced in 2002, also trace their roots back to MD4.

Their are some variations inside members of the family; notably, RIPEMD uses a parallel structure for its compression function. We specifically list below features that are shared by MD4, MD5 and SHA, but they are also true for other MD-SHA functions to a large extent.

- The Merkle-Damgård construction is used as a domain extender.
- The compression function is built from an *ad-hoc* block cipher used in a *Davies-Meyer* mode: let $\mathcal{E}(x, y)$ be the encryption of the plaintext y with the key x by the cipher $\mathcal{E}$, then the compression function is defined by $\mathsf{c}_{i+1} := \mathcal{E}(\mathsf{m}_{i+1}, \mathsf{c}_i) + \mathsf{c}_i$.
- The block cipher inside the compression function is an unbalanced Feistel network that uses modular additions, XORs, bit rotations, and bitwise Boolean functions as constitutive elements. Its key schedule is linear, and very fast to compute.

With the advent of the NIST SHA-3 competition, which ran from 2007 to 2012, much more diversity was introduced to hash function design. However, the MD-SHA family still had an influence on many competition candidates, and it remains influential as of today.

Part Two

New attacks and elements of design for block ciphers

Overview

A widespread design type for block ciphers is the *SPN* structure. Initially, this stood for *substitution-permutation network*, with *permutation* referring to bit permutations; in its current usage, this term generally denotes arbitrary $\mathbf{F}_2$-linear mappings.

The rationale behind the SPN structure is that one may obtain a good block cipher round function by alternating a substitution mapping made of the parallel application of *S-boxes* (short for *substitution boxes*) and a linear mapping. The role of both mappings are complementary, and each is also usually selected to interact well with the other. While the S-boxes are expected to break any exploitable linear and algebraic structure at the local level, the permutation should spread the effects of the S-boxes at the global level, in particular to ensure that no local change in the state of the cipher remains so through the entire cipher execution.

An advantage of SPNs over some other design structures is that they lend themselves relatively well to analysis, in particular with respect to statistical attacks; this is exemplified by the *wide-trail strategy* used in the design of the AES. Another notable point about SPNs is that they offer many possible tradeoffs in their instantiations. One can choose between large and small S-boxes, large S-boxes being comparatively stronger than small ones but also more expensive; best-performing S-boxes or cheaper average ones; bit permutations, particularly efficient in hardware, or non-permutation matrices; optimally diffusing (MDS) matrices or not; state-wide matrices or not, etc. The design space of SPNs is thus quite large, and concrete instantiations of the framework may be quite distinct one from another.

In the second part of this thesis, we study two mappings intended to be used as part of an SPN round function, the second of them being also instantiated in a complete block cipher design. On the cryptanalysis side, we start by presenting a simple attack of the Prøst-OTR authenticated-encryption scheme, using a related-key model.

In Chapter 5, we show how the simple related-key distinguisher on Even-Mansour constructions can be converted into a key-recovery attack by considering related-key introduced with a modular addition rather than an XOR. Although this observation is quite elementary, it finds a practical application in a very efficient related-key attack on PRØST-OTR, which is an instantiation of the OTR authenticated-encryption mode of operation with the PRØST permutation.

In Chapter 6, we present large linear mappings with very good diffusion and efficient software implementations, with respect to their sizes. They are derived from linear codes over a small field, typically $\mathbf{F}_{2^4}$, with a high dimension, typically 16, and a high minimum distance. This results in diffusion matrices with equally high dimension and a large branch number. Because no MDS code is known to exist for the selected parameters, the matrices were derived from more flexible *algebraic geometry* codes. We present two simple yet efficient algorithms for the software implementation of matrix-vector multiplication in this context, and derive conditions on the generator matrices of the codes to yield efficient encoders. We then specify an appropriate code and use its automorphisms as well as random sampling to find good such matrices.

In Chapter 7, we present the construction and implementation of an 8-bit S-box with a differential and linear branch number of 3. We show an application by designing FLY, a simple block cipher based on bitsliced evaluations of the S-box and bit rotations which is designed to be efficient on 8-bit microcontrollers in particular. The round function of FLY achieves good performance on its target platform, in terms of number of instructions per round and overall code size, while providing good cryptographic properties. The S-box also has an efficient implementation with SIMD instructions, a low implementation cost in hardware and it can be masked efficiently thanks to its sparing use of non-linear gates and to the fact that it has a natural expression in terms of a single 4-bit S-box.

Contents

5 Improved related-key attacks on Even-Mansour

6 Efficient diffusion matrices from algebraic geometry codes

7 The Littlun S-box and the Fly block cipher

Chapter 5

Improved related-key attacks on Even-Mansour

1 Introduction

The Even-Mansour scheme, or simply Even-Mansour, or EM, is arguably the simplest way to construct a block cipher from publicly available components. It defines the encryption $\mathcal{E}((k_1, k_0), p)$ of the plaintext p under the possibly equal keys k_0 and k_1 as $\mathcal{P}(p \oplus k_0) \oplus k_1$, where $\mathcal{P}$ is a public permutation. Even and Mansour proved in 1991 that for a permutation over n-bit values, the probability of recovering the keys is upper-bounded by $\mathcal{O}(qt \cdot 2^{-n})$ when the attacker considers the permutation as a black box, where q is the data complexity (the number of accesses to the encryption oracle) and t is the time complexity (the number of accesses to the permutation) of the attack [EM91]. Although of considerable interest, this bound also shows at the same time that the construction is not ideal, as one gets security only up to $\mathcal{O}(2^{\frac{n}{2}})$ queries, which is less than the $\mathcal{O}(2^n)$ one would expect from an n-bit block cipher. For this reason, much later work investigated the security of variants of the original construction. A simple one is the iterated Even-Mansour scheme (IEM). When using independent keys and independent permutations, its r-round version is defined as $\mathsf{IEM}^r((k_r, k_{r-1}, \ldots, k_0), p) \coloneqq \mathcal{P}_{r-1}(\mathcal{P}_{r-2}(\ldots \mathcal{P}_0(p \oplus k_0) \oplus k_1) \ldots) \oplus k_r$, and it has been established by Chen and Steinberger that this construction is secure up to $\mathcal{O}(2^{\frac{rn}{r+1}})$ queries [CS14]. On the other hand, in a related-key model, the same construction lends itself to trivial distinguishing attacks, and one must consider alternatives if security in this model is necessary. Yet until the recent work of Cogliati and Seurin [CS15] and Farshim and Procter [FP15], no variant of Even-Mansour was proved to be secure in the related-key model. This is not the case anymore and it has now been proven that one can reach a non-trivial level of related-key security for IEM^r starting from $r = 3$ when using keys linearly derived from a single master key instead of using independent keys, or even for the original non-iterated EM scheme if one uses a non-linear key derivation meeting some conditions. While related-key analysis obviously gives much more power to the attacker than the single-key setting, it is a widely accepted model that may provide useful results on primitives studied in a general context.

2 Notation

We use $||$ to denote string concatenation, a^i with i an integer to denote the string made of the concatenation of i copies of the character a, and a^* to denote any string of the set $\{a^i, i \in \mathbf{N}\}$, a^0 denoting the empty string ε. For any string s, we use $s[i]$ to denote its i^{th} element, starting from zero.

We also use Δ_i^n to denote the string $0^{n-i-1}||1||0^{i-1}$. The value of the superscript n will in fact always be clear from the context and therefore omitted.

Finally, we identify strings of length n over the binary alphabet $\{0,1\}$ with elements of the vector-space $\mathbf{F}_2^n$ and with the binary representation of elements of the group $\mathbf{Z}/2^n\mathbf{Z}$. The addition operations on these structures are respectively denoted by $\oplus$, the bitwise exclusive or (XOR) and $+$, the modular addition.

3 Generic related-key key-recovery attacks on IEM

Since the work of Bellare and Kohno [BK03], it is well-known that no block cipher can resist related-key attacks (RKA) when an attacker may request encryptions under related keys using two relation classes. A simple example showing why this cannot be the case is to consider the classes $\phi^{\oplus}(k)$ and $\phi^{+}(k)$ of keys related to k by the XOR and the modular addition of any constant chosen by the attacker respectively. If we have access to related-key encryption oracles $\mathcal{E}(k,\cdot)$, $\mathcal{E}(\phi^{\oplus}(k),\cdot)$ and $\mathcal{E}(\phi^{+}(k),\cdot)$ for the block cipher $\mathcal{E}$ with κ-bit keys, we can easily learn the value of the bit $k[i]$ of k by comparing the results of the queries $\mathcal{E}(k+\Delta_i,p)$ and $\mathcal{E}(k\oplus\Delta_i,p)$. For $i<\kappa-1$, the plaintext p is encrypted under the same key if $k[i]=0$, then resulting in the same ciphertext, and is encrypted under different keys if $k[i]=1$, then resulting in different ciphertexts with overwhelming probability. Doing this test for every bit of k thus allows to recover the whole key with a complexity linear in κ, except its most significant bit as the carry of a modular addition on the MSB never propagates.This latter bit can of course easily be recovered once all the others have been determined.

In the same paper, Bellare and Kohno also show that no such trivial generic attack exists when the attacker is restricted to using only one of the two classes $\phi^{\oplus}$ or ϕ^{+}, and they prove that an ideal cipher is in this case resistant to RKA. Taken together, these results mean in essence that a related-key attack on a block cipher $\mathcal{E}$ using both classes $\phi^{\oplus}(k)$ and $\phi^{+}(k)$ does not say much on $\mathcal{E}$, as nearly all ciphers fall to an attack in the same model. On the other hand, an attack using either of $\phi^{\oplus}$ or ϕ^{+} *is* meaningful, because an ideal cipher is secure in that case. Nonetheless, one should note that when queries with many different related keys are allowed, this security remains lower than in a single-key model, because of the ability to create collisions. One can for instance query $\mathcal{E}(k\oplus\delta_i,p)$ with $2^{k/2}$ different values for δ_i and a constant p and then compute

$\mathcal{E}(\gamma_i, p)$ with $2^{k/2}$ guesses γ_i for k. With high probability, one of the γ_is will be equal to one of the $k \oplus \delta_i$s, which allows to recover k.

3.1 Key-recovery attacks on IEM^r with independent keys

Going back to IEM, we explicit the trivial related-key distinguishers mentioned in the introduction. These distinguishers exist for r-round iterated Even-Mansour schemes with independent keys, for any value of r. As they only use keys related with, say, the $\phi^\oplus$ class, they are therefore meaningful when considering the related-key security of IEM.

From the very definition of IEM^r, it is obvious that the two values $\mathcal{E}((k_{r-1}, k_{r-2}, \dots, k_0), p)$ and $\mathcal{E}((k_{r-1}, k_{r-2}, \dots, k_0 \oplus \delta), p \oplus \delta)$ are equal for any difference δ when $\mathcal{E} =$ IEM^r and that this equality does not hold in general, thence allowing to distinguish IEM^r from an ideal cipher. This is illustrated for the original EM in Figure 5.1.

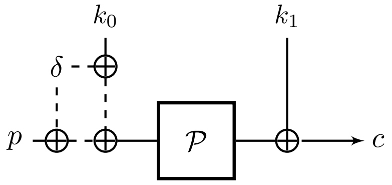


Figure 5.1 – A related-key distinguisher on EM. Dashed lines represent datapaths where a difference is injected and not canceled.

We now show how these distinguishers can be combined with the two-class attack of Bellare and Kohno in order to extend it to a very efficient key-recovery attack. We give a description in the case of one-round Even-Mansour, but it can easily be extended to an arbitrary r. The attack is very simple and works as follows: consider again $\mathcal{E}((k_1, k_0), p) = \mathcal{P}(p \oplus k_0) \oplus k_1$; one can learn the value of the bit $k_0[i]$ by querying $\mathcal{E}((k_1, k_0), p)$ and $\mathcal{E}((k_1, k_0 + \Delta_i), p \oplus \Delta_i)$ and by comparing their values. These differ with overwhelming probability if $k_0[i] = 1$ and are equal otherwise. This is illustrated in Figure 5.2 and Figure 5.3

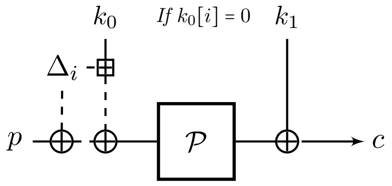


Figure 5.2 – Related-key queries to IEM^1 (*i.e.* EM) with no output difference.

A similar attack works on the variant of the (iterated) EM that uses modular addition instead of XOR for the combination of the key with the plaintext. This variant was first

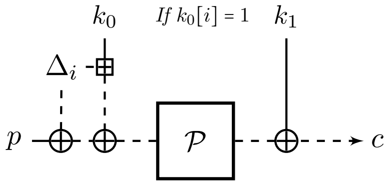


Figure 5.3 – Related-key queries to IEM^1 (*i.e.* EM) with an output difference.

analysed by Dunkelman, Keller and Shamir [DKS12] and offers the same security bounds as the original Even-Mansour. An attack in that case works similarly by querying *e.g.* $\mathcal{E}((k_1, k_0), \Delta_i)$ and $\mathcal{E}((k_1, k_0 \oplus \Delta_i), 0^\kappa)$.

Both attacks use a single difference class for the related keys, either $\phi^\oplus$ or ϕ^+, and they are therefore meaningful as related-key attacks. They simply emulate the attack that uses both classes simultaneously by taking advantage of the fact that the usage of key material is very simple in Even-Mansour. Finally, we can see that in the particular case of a one-round construction, the attack still works without adaptation if one chooses the keys k_1 and k_0 to be equal.

3.2 Extension to IEM^2 with a linear key schedule

Cogliati and Seurin [CS15] showed that it is also possible to very efficiently distinguish IEM^2 with related keys, even when the keys are equal or derived from a master key by a linear key schedule. Similarly as for independent-key IEM, we can adapt the distinguisher and transform it into a key-recovery attack. The idea remains the same: one replaces the $\phi^\oplus$ class of the original distinguisher with ϕ^+, which makes its success conditioned on the value of a few key bits, hence allowing their recovery. We give the description of our modified distinguisher for $\mathcal{E}(k,p) \coloneqq \mathcal{P}(\mathcal{P}(k \oplus p) \oplus k) \oplus k$, where we let the δ_is denote arbitrary differences:

1. Query $y_1 \coloneqq \mathcal{E}(k + \delta_1, x_1)$.
2. Set x_2 to $x_1 \oplus \delta_1 \oplus \delta_2$ and query $y_2 \coloneqq \mathcal{E}(k + \delta_2, x_2)$.
3. Set y_3 to $y_1 \oplus \delta_1 \oplus \delta_3$ and query $x_3 \coloneqq \mathcal{E}^{-1}(k + \delta_3, y_3)$.
4. Set y_4 to $y_2 \oplus \delta_1 \oplus \delta_3$ and query $x_4 \coloneqq \mathcal{E}^{-1}(k + (\delta_1 \oplus \delta_2 \oplus \delta_3), y_4)$.
5. Test if $x_4 = x_3 \oplus \delta_1 \oplus \delta_2$.

If the test is successful, it means that with overwhelming probability the key bits at the positions of the differences δ_1, δ_2, δ_3 are all zero, as in that case $k + \delta_i = k \oplus \delta_i$ and the distinguisher works “as intended”, and as otherwise at least one uncontrolled difference goes through $\mathcal{P}$ or $\mathcal{P}^{-1}$. It is possible to restrict oneself to using differences in only two bits for the δ_is, and as soon as two zero key bits have been found (which happens after an expected four trials for random keys), the rest of the key can be determined one bit at a time.

We conclude this short section by showing why the test of line 5 is successful when $k + \delta_i = k \oplus \delta_i$, but refer to Cogliati and Seurin for a complete description of their distinguisher, including the general case of distinct permutations and keys linearly derived from a master key.

We write $k \oplus \delta_i$ for $k + \delta_i$ to make the simplifications more obvious, as these values are equal by hypothesis. By definition, $y_1 = \mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_1) \oplus k \oplus \delta_1$ and $y_3 = \mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_1) \oplus k \oplus \underline{\delta_1} \oplus \overline{\delta_1} \oplus \delta_3$ which simplifies to $\mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_1) \oplus k \oplus \delta_3$. This yields the following expression for x_3:

$$\begin{aligned}
x_3 &= \mathcal{P}^{-1}(\mathcal{P}^{-1}(\mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_1) \oplus \underline{k \oplus \delta_3} \oplus \overline{k \oplus \delta_3}) \oplus k \oplus \delta_3) \oplus k \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\underline{\mathcal{P}^{-1}(}\overline{\mathcal{P}(}\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_1\overline{)}\underline{)} \oplus k \oplus \delta_3) \oplus k \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus \underline{k} \oplus \delta_1 \oplus \overline{k} \oplus \delta_3) \oplus k \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus \delta_1 \oplus \delta_3) \oplus k \oplus \delta_3
\end{aligned}$$

Similarly, $y_2 = \mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_2) \oplus k \oplus \delta_2$ and $y_4 = \mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_2) \oplus k \oplus \delta_2 \oplus \delta_1 \oplus \delta_3$, which yields the following expression for x_4:

$$\begin{aligned}
x_4 &= \mathcal{P}^{-1}(\mathcal{P}^{-1}(\mathcal{P}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_2) \oplus \underline{k \oplus \delta_2 \oplus \delta_1 \oplus \delta_3} \oplus \overline{k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3}) \\
&\quad \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3) \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\underline{\mathcal{P}^{-1}(}\overline{\mathcal{P}(}\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus k \oplus \delta_2\overline{)}\underline{)} \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3) \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus \underline{k \oplus \delta_2} \oplus \overline{k} \oplus \delta_1 \oplus \overline{\delta_2} \oplus \delta_3) \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3 \\
&= \mathcal{P}^{-1}(\mathcal{P}(x_1 \oplus k \oplus \delta_1) \oplus \delta_1 \oplus \delta_3) \oplus k \oplus \delta_1 \oplus \delta_2 \oplus \delta_3
\end{aligned}$$

From the final expressions of x_3 and x_4, we see that their XOR difference is indeed $\delta_1 \oplus \delta_2$.

4 Application to Prøst-OTR

We now apply the simple generic key-recovery attack of Section 3 to the CAESAR candidate PRØST-OTR, which is an authenticated-encryption scheme member of the PRØST family [KLL+14]. This family is based on the PRØST permutation and defines three schemes instantiating as many modes of operation, namely COPA, OTR and APE. Only the latter can readily be instantiated with a permutation, and both COPA and OTR rely on a keyed primitive. In order to be instantiated with PRØST, the permutation is expanded to a block cipher defined as a one-round Even-Mansour scheme with identical keys $\mathcal{E}(k, p) := \mathcal{P}(p \oplus k) \oplus k$, with the PRØST permutation as $\mathcal{P}$. We will denote this cipher as PRØST/SEM.

Although the attack of Section 3 could readily be applied to PRØST/SEM, this cipher is only meant to be embedded into a specific instantiation of a mode such as OTR, and attacking it out of context is not relevant to its intended use. Hence we must be able to mount an attack on Prøst-COPA or Prøst-OTR as a whole for it to be really significant, which is precisely what we describe now for the latter.

Because our attack solely relies on the Even-Mansour structure of the cipher, we refer the interested reader to the submission document of Prøst for the definition of its permutation. The same goes for the OTR mode [Min14], as we only need to focus on a small part to describe the attack. Consequently, we just describe how the encryption of the first block of plaintext is performed in Prøst-OTR. Note that this is not necessarily the same as encrypting the first block in every instantiations of OTR, as there is some flexibility in the definition of the mode.

The mode of operation OTR is nonce-based; it takes as input a key k, a nonce n, a message m, possibly empty associated data a, and produces a ciphertext c corresponding to the encryption of the message with k, and a tag t authenticating m and a together with the key k. It is important for the security of the mode to ensure that one cannot encrypt twice using the same nonce. However, there are no specific additional restrictions on the nonces, and we consider that an attacker has full control over their non-repeating values.

The encryption of the first block of ciphertext c_1 by Prøst-OTR is defined as a function $f(k, n, m_1, m_2)$ of k, n, and the first two blocks of plaintext m_1 and m_2: let $\ell := \mathcal{E}(k, n||10^*)$ be the encryption of the padded nonce and $\ell' := \pi(\ell)$, with π a linear permutation, in this case the multiplication by 4 in some finite field, then c_1 is simply equal to $\mathcal{E}(k, \ell' \oplus m_1) \oplus m_2$. We show this schematically along with the encryption of the second block in Figure 5.4. Let us now apply the attack from Section 3.

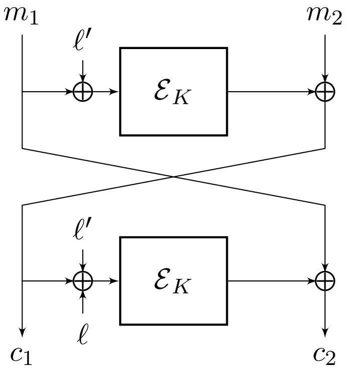


Figure 5.4 – The encryption of the first two blocks of message in Prøst-OTR.

4.1 Step 1: Recovering the most significant half of the key

It is straightforward to see that one can recover the value of the bit $k[i]$ by performing only two queries with related keys and different nonces and messages. One just has to compare $c_1 = f(k, n, m_1, m_2)$ and $\hat{c}_1 = f(k + \Delta_i, n \oplus \Delta_i, m_1 \oplus \Delta_i \oplus \pi(\Delta_i), m_2)$. Indeed, if $k[i] = 0$, then the value $\hat{\ell}$ obtained in the computation of $\hat{c}_1$ is equal to $\ell \oplus \Delta_i$ and $\hat{\ell}' = \ell' \oplus$

$\pi(\Delta_i)$, hence $\hat{c}_1 = c_1 \oplus \Delta_i$. If $k[i] = 1$, the latter equality does not hold with overwhelming probability. We illustrate the propagation of differences in the computation of ℓ' and $\hat{\ell}'$ in Figure 5.5 and Figure 5.6.

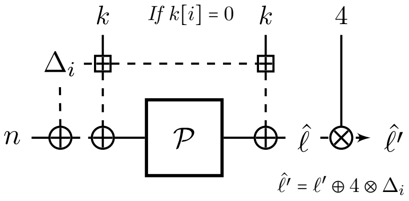


Figure 5.5 – Related-key queries to PRØST-OTR with predictable output difference.

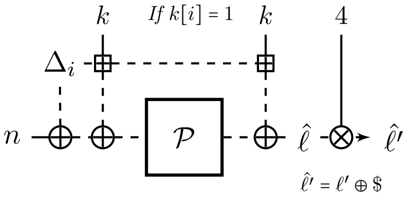


Figure 5.6 – Related-key queries to PRØST-OTR with unpredictable output difference.

Yet this does not allow to recover the whole key because the nonce in PRØST-OTR is restricted to a length half of the width of the block cipher $\mathcal{E}$, or equivalently of the underlying PRØST permutation, *i.e.* $\frac{\kappa}{2}$. It is then possible to recover only half of the bits of k using this procedure, as one cannot introduce appropriate differences in the computation of ℓ for the other half. For the same reason, it is not possible either to finish the attack by looking for collisions in the unknown half of the key using the classical attack on Even-Mansour schemes. The targeted security of the whole primitive being precisely $\frac{\kappa}{2}$ because of the classical single key attack, one does not make a significant gain by recovering only half of the key. Nonetheless, it should still be noted that this yields an attack with very little data requirements and with the same time complexity as the best point on the tradeoff curve of generic attacks, which in that case has a much higher data complexity of $2^{\frac{\kappa}{2}}$.

4.2 Step 2: Recovering the least significant half of the key

Even though the generic attack in its most simple form does not allow to recover the full key of PRØST-OTR, we can use the fact that the padding of the nonce is done on the least significant bits to our advantage, and by slightly adapting the procedure of the first step, we can iteratively recover the value of the least significant half of the key with no more effort than for the most significant half.

Let us first show how we can recover the most significant bit of the least significant half of the key $k[\kappa/2-1]$, *i.e.* the first bit for which we cannot use the previous method, after a single encryption by $\mathcal{E}$.

We note k^{MSB} the known most significant half of the key k. To mount the attack, one queries $\mathcal{E}(k-k^{\mathsf{MSB}}+\Delta_{\kappa/2-1}, p\oplus\Delta_{\kappa/2})$ and $\mathcal{E}(k-k^{\mathsf{MSB}}-\Delta_{\kappa/2-1}, p)$. We can see that the inputs to $\mathcal{P}$ in these two cases are equal iff $k[\kappa/2-1]=1$. Indeed, in that case, the carry in the addition $(k-k^{\mathsf{MSB}})+\Delta_{\kappa/2-1}$ propagates by exactly one position and is cancelled by the difference in p, and there is no carry propagation in $(k-k^{\mathsf{MSB}})-\Delta_{\kappa/2-1}$. Set $C := \mathcal{P}(p\oplus(k-k^{\mathsf{MSB}}-\Delta_{\kappa/2-1}))$, then the result of the two queries are equal to $C\oplus(k-k^{\mathsf{MSB}}+\Delta_{\kappa/2-1}) = C\oplus(k\oplus k^{\mathsf{MSB}}\oplus\Delta_{\kappa/2-1}\oplus\Delta_{\kappa/2})$ and $C\oplus(k-k^{\mathsf{MSB}}-\Delta_{\kappa/2-1}) = C\oplus(k\oplus k^{\mathsf{MSB}}\oplus\Delta_{\kappa/2-1})$. Consequently, the XOR difference between the two results is known and equal to $\Delta_{\kappa/2}$. If on the other hand $k[\kappa/2-1]=0$, the carry in $((k-k^{\mathsf{MSB}})-\Delta_{\kappa/2-1})$ propagates all the way to the most significant bit of k, whereas only two differences are introduced in the input to $\mathcal{P}$ in the first query. This allows to distinguish between the two cases and thus to recover the value of this key bit.

Once the value of $k[\kappa/2-1]$ has been learned, one can iterate the process to recover the remaining bits of k. The only subtlety is that we want to ensure that if there is a carry propagation in $(k-k^{\mathsf{MSB}})+\Delta_{\kappa/2-1-i}$ (resp. $(k-k^{\mathsf{MSB}})-\Delta_{\kappa/2-1-i}$), it should propagate up to $k_{\kappa/2}$, the position where we cancel it with an XOR difference (resp. up to the most significant bit); this can easily be achieved by adding two terms to both keys. Let us define γ_i as the value of the key k only on positions $\kappa/2-1\ldots\kappa/2-i$, completed with zeros left and right; that is $\gamma_i[j]=k[j]$ if $\kappa/2-1\geq j\geq\kappa/2-i$, and $\gamma_i[j]=0$ otherwise. Let us also define $\widetilde{\gamma_i}$ as the binary complement of γ_i on its non-zero support, that is $\widetilde{\gamma_i}[j]=\neg k[j]$ if $\kappa/2-1\geq j\geq\kappa/2-i$, and $\widetilde{\gamma_i}[j]=0$ otherwise. The modified queries then become $\mathcal{E}(k-k^{\mathsf{MSB}}+\Delta_{\kappa/2-1-i}+\widetilde{\gamma_i}, p\oplus\Delta_{\kappa/2})$ and $\mathcal{E}(k-k^{\mathsf{MSB}}-\Delta_{\kappa/2-1-i}-\gamma_i, p)$, for which the propagation of the carries is ensured. Note that the difference between the results of these two queries when $k[\kappa/2-1-i]=1$ is independent of i and always equal to $\Delta_{\kappa/2}$.

We conclude by showing how to apply this procedure to Prøst-OTR. For the sake of readability, let us denote by Δ_i^+ and Δ_i^- the complete modular differences used to recover one less significant bit $k[i]$, that is $\Delta_i^+=-k^{\mathsf{MSB}}+\Delta_{\kappa/2-1-i}+\widetilde{\gamma_i}$ and $\Delta_i^-=-k^{\mathsf{MSB}}+\Delta_{\kappa/2-1-i}-\gamma_i$. We then simply perform the two queries $f(k+\Delta_i^+, n\oplus\Delta_{\kappa/2}, m_1\oplus\Delta_{\kappa/2}, m_2)$ and $f(k+\Delta_i^-, n, m_1\oplus\pi(\Delta_{\kappa/2}), m_2)$, which differ by $\Delta_{\kappa/2}$ iff k_i is one, with overwhelming probability.

All in all, one can retrieve the whole key of size κ using only 2κ chosen-nonce related-key encryption requests with $\kappa+\kappa/2+1$ different keys, ignoring everything in the output apart from the value of the first block of ciphertext. We give the full attack in Algorithm 5.1. Note that it makes use of a procedure Refresh which picks fresh values for two message words and most importantly for the nonce. Because the attack is very fast, it can easily be tested. We give an implementation for a 64-bit toy cipher in Section A.

Remark. If the padding of the nonce in Prøst-OTR were done on the most significant bits, no attack similar to Step 2 could recover the corresponding key bits: the modular

Algorithm 5.1: Related-key key recovery for PRØST-OTR

```
Input: Oracle access to $f(k,\cdot,\cdot,\cdot)$ and $f(\phi^+(k),\cdot,\cdot,\cdot)$ for a fixed unknown key $k$
       of even length $\kappa$
Output: Two candidates for the key $k$
$k' := 0^\kappa$
for $i := \kappa - 2$ to $\kappa/2$ do
    REFRESH($n$, $m_1$, $m_2$)
    $x := f(k, n, m_1, m_2)$
    $y := f(k + \Delta_i, n \oplus \Delta_i, m_1 \oplus \Delta_i \oplus \pi(\Delta_i), m_2)$
    if $x = y \oplus \Delta_i$ then
        $k'[i] := 0$
    else
        $k'[i] := 1$
for $i := \kappa/2 - 1$ to $0$ do
    REFRESH($n$, $m_1$, $m_2$)
    $x := f(k + \Delta_i^+, n \oplus \Delta_{\kappa/2}, m_1 \oplus \Delta_{\kappa/2}, m_2)$
    $y := f(k + \Delta_i^-, n, m_1 \oplus \pi(\Delta_{\kappa/2}), m_2)$
    /* The values of $\Delta_i^+$ and $\Delta_i^-$ are computed as in Section 4.2 */
    if $x = y \oplus \Delta_{\kappa/2}$ then
        $k'[i] := 1$
    else
        $k'[i] := 0$
$k'' := k'$
$k''[\kappa - 1] := 1$
return $(k', k'')$
```

addition is a triangular function, meaning that the result of $a + b$ on a bit i only depends on the value of bits of position less than i in a and b, and therefore no XOR in the nonce in the less significant bits could control modular differences introduced in the padding in the more significant bits. An attack in that case would thus most likely be applicable to general ciphers when using only the ϕ^+ class, and it is proven that no such attack is efficient. However, one could always imagine using a related-key class using an addition operation reading the bits in reverse. While admittedly unorthodox, this would not result in a stronger model than using ϕ^+, strictly speaking.

Discussion. In a recent independent work, Dobraunig, Eichlseder and Mendel use similar methods to produce forgeries for PRØST-OTR by considering related keys with XOR differences [DEM15]. On the one hand, one could argue that the class $\phi^\oplus$ seems more natural than ϕ^+ and could be more likely to arise in actual situations, which would make their attack more applicable than ours. On the other hand, an ideal cipher is expected to give a similar security against RKA using either class, which means that our model is not theoretically stronger than the one of Dobraunig *et al.*, while resulting in a much more powerful key recovery attack.

5 Conclusion

We made a simple observation that allows to convert related-key distinguishing attacks on some Even-Mansour schemes into much more powerful key-recovery attacks, and we used this observation to derive an extremely efficient key-recovery attack on the Prøst-OTR Caesar candidate, in the related-key model.

Primitives based on EM are quite common, and it is natural to wonder if we could mount similar attacks on other ciphers. A natural first target would be Prøst-COPA which is also based on the Prøst/SEM cipher. However, in this mode, encryption and tag generation depend on the encryption of a fixed plaintext $\ell := \mathcal{E}(k, 0)$ which is different for different keys with overwhelming probability and makes our attack fail. The forgery attacks of Dobraunig *et al.* seem to fail in that case for the same reason. Keeping with Caesar candidates, another good target would be Minalpher [STA$^+$14], which also uses a one-round Even-Mansour block cipher as one of its components. The attack also fails in this case, though, because the masking key used in the Even-Mansour cipher is derived from the master key in a highly non-linear way. In fact, Mennink recently proved that both ciphers are resistant to related-key attacks [Men16]. Finally, leaving aside authentication and going back to traditional block ciphers, we could consider designs such as LED [GPPR11]. The attack also fails in that case, however, because the cipher uses an iterated construction with at least 8 rounds and only one or two keys.

This lack of other results is not very surprising, as we only improve existing distinguishing attacks, and this improvement cannot be used without a distinguisher as its basis. Therefore, any primitive for which resistance to related-key attacks is important should already be resistant to the distinguishing attacks and thus to ours. Yet it would be reasonable to allow the presence of a simple related-key distinguisher when designing a primitive, as this is a very weak type of attack; in fact, this is for instance the approach taken by PRINCE, among others [BCG$^+$12], which admits a trivial distinguisher due to its FX construction. What we have shown is that one must be careful when contemplating such a decision for EM, and in fact FX constructions to some extent as well, as in that case it is actually equivalent to allowing key recovery, the most powerful of all attacks.

A Proof-of-concept implementation for a 64-bit permutation

We give in Figure 5.7 the source of a C program that recovers a 64-bit key from a design similar to Prøst-OTR, where the Prøst permutation has been replaced by a small 64-bit ARX for compactness. This allows to verify the correctness of the attack, all the while showing that it is indeed very simple to implement. For the sake of simplicity, we do not ensure that the nonce does not repeat in the queries.

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define ROL32(x,r) (((x) << (r)) ^ ((x) >> (32 - (r))))
#define MIX(hi,lo,r) \
{ (hi) += (lo); (lo) = ROL32((lo),(r)) ; (lo) ^= (hi); }

/* Multiplication by x and x^2 in F_2[x]/x^64 + x^4 + x^3 + x^1 + x */
#define TIMES2(x) ((x & 0x8000000000000000ULL) ? ((x) << 1ULL) ^ \
0x000000000000001BULL : (x << 1ULL))
#define TIMES4(x) TIMES2(TIMES2((x)))

#define DELTA(x) (1ULL << (x))
#define MSB(x) ((x) & 0xFFFFFFFF00000000ULL)
#define LSB(x) ((x) & 0x00000000FFFFFFFFULL)

/* A 64-bit permutation using Skein's MIX function */
uint64_t p64(uint64_t x)
{
  uint32_t hi = x >> 32;
  uint32_t lo = LSB(x);
  unsigned rcon[8] = {1, 29, 4, 8, 17, 12, 3, 14};

  for (int i = 0; i < 32; i++)
  {
     MIX(hi, lo, rcon[i % 8]);
     lo += i;
  }

  return ((((uint64_t)hi) << 32) ^ lo);
}

/* A 64-bit Even-Mansour block cipher using the p64 permutation */
uint64_t em64(uint64_t k, uint64_t p)
{
  return p64(k ^ p) ^ k;
}

/* The encryption of one block of message as done in
 * Proest-OTR, using em64 as the underlying block cipher */
uint64_t potr_1(uint64_t k, uint64_t n, uint64_t m1, uint64_t m2)
{
  uint64_t l, c;

  l = TIMES4(em64(k, n));
  c = em64(k, l ^ m1) ^ m2;

  return c;
}

/* Step 1 of the attack, that recovers the highest
 * 32 bits of the key
 * The ``secret'' key is taken as an argument to
 * implement related-key oracle queries */
uint64_t recover_hi(uint64_t secret_key)
{
  uint64_t kk = 0;

  for (int i = 62; i >= 32; i--)
  {
    uint64_t m1, m2, c11, c12, n;

    m1 = (((uint64_t)arc4random()) << 32) ^ arc4random();
    m2 = (((uint64_t)arc4random()) << 32) ^ arc4random();
    n  = (((uint64_t)arc4random()) << 32) ^ 0x80000000ULL;
    c11 = potr_1(secret_key, n, m1, m2);
    c12 = potr_1(secret_key + DELTA(i), n ^ DELTA(i), m1 ^ DELTA(i) ^
 ↪  TIMES4(DELTA(i)), m2);

    if (c11 != (c12 ^ DELTA(i)))
      kk |= DELTA(i);
  }

  return kk;
}

/* Step 2 of the attack, that recovers the lowest
 * 32 bits of the key */
uint64_t recover_lo(uint64_t secret_key, uint64_t hi_key)
{
  uint64_t kk = hi_key;

  for (int i = 31; i >= 0; i--)
  {
    uint64_t m1, m2, c11, c12, n;
    uint64_t delta_p, delta_m;

    m1 = (((uint64_t)arc4random()) << 32) ^ arc4random();
    m2 = (((uint64_t)arc4random()) << 32) ^ arc4random();
    n  = (((uint64_t)arc4random()) << 32) ^ 0x80000000ULL;

    delta_p = DELTA(i) - MSB(kk) + (((LSB(~kk)) >> (i + 1)) << (i + 1));
    delta_m = DELTA(i) + MSB(kk) + LSB(kk);
    c11 = potr_1(secret_key + delta_p, n ^ DELTA(32), m1 ^ DELTA(32),
 ↪  m2);
    c12 = potr_1(secret_key - delta_m, n, m1 ^ TIMES4(DELTA(32)), m2);

    if (c11 == (c12 ^ DELTA(32)))
      kk |= DELTA(i);
  }

  return kk;
}

/* Selects a random key and tries to recover it thanks to the attack */
int main()
{
  uint64_t secret_key = (((uint64_t)arc4random()) << 32) ^ arc4random();
  uint64_t kk1 = recover_lo(secret_key, recover_hi(secret_key));
  uint64_t kk2 = kk1 ^ 0x8000000000000000ULL;

  printf("The real key is %016llx, the key candidates are %016llx,
 ↪  %016llx    ", secret_key, kk1, kk2);
  if ((kk1 == secret_key) || (kk2 == secret_key))
    printf("SUCCESS!\n");
  else
    printf("FAILURE!\n");

  return 0;
}
```

Figure 5.7 – A proof-of-concept implementation of the related-key attack on Prøst-OTR.

The code of Figure 5.7 is a straightforward implementation of Algorithm 5.1. The attack is split in two functions, `recover_hi` and `recover_lo`, that each recover one half of the key; the rest of the code minimally implements the Prøst-OTR-like cryptosystem.

Chapter 6

Efficient diffusion matrices from algebraic geometry codes

1 Introduction

The use of *MDS* matrices over finite fields as linear mappings in block cipher design is an old trend, followed by many prominent algorithms such as the AES/Rijndael family [DR02] and its predecessors [RDP+96, DKR97]. These matrices are called MDS as they are derived from *maximum distance separable* linear error-correcting codes, which achieve the highest minimum distance possible for a given length and dimension. This notion of minimum distance coincides with the one of *branch number* of a mapping [Dae95], which is a measure of the effectiveness of a diffusion layer. MDS matrices thus have an optimal diffusion, in a cryptographic sense, which makes them attractive for cipher design.

The good security properties that can be derived from MDS matrices are often counter-balanced by the cost of their computation. The standard matrix-vector product is quadratic in the dimension of the vector, and finite field operations are not always efficient. For that reason, there is often a focus on finding matrices allowing efficient implementations. For instance, the AES matrix is circulant and has small coefficients. More recently, the PHOTON hash function [GPP11] introduced the use of matrices that can be obtained as the power of a companion matrix, whose sparsity may be useful in lightweight hardware implementations. The topic of finding such so-called recursive diffusion layers has been quite active in the past years, and led to a series of papers investigating some of their various aspects [SDMS12, WWW12, AF13]. One recent development shows how to systematically construct some of these matrices from BCH codes [AF15]. This allows in particular to construct very large recursive MDS matrices, for instance of dimension sixteen over $\mathbf{F}_{2^8}$. This defines a linear mapping over a full 128-bit block with excellent diffusion properties, at a moderate hardware implementation cost.

As interesting as it may be in hardware, the cost in software of a large linear mapping tends to make these designs rather less attractive than more balanced solutions. An early attempt to use a large matrix was the block cipher SHARK, a Rijndael predecessor [RDP+96]; the same kind of design was also later used for Khazad [BR01]. Both are 64-bit ciphers which use an MDS matrix of dimension eight over $\mathbf{F}_{2^8}$ for their linear diffusion. The usual technique for implementing such a mapping in software is to rely on a table of precomputed multiples of the matrix rows. However, table-based implementations now tend to be frown upon as they may lead to *timing attacks* [TOS10], and this could leave ciphers with a structure similar to SHARK's without reasonable software implementations when resistance to these attacks is required. Yet, such designs also have advantages of their own; their diffusion acts on the whole state at every round, and therefore makes structural attacks harder, while also ensuring that many S-Boxes are kept active. Additionally, the simplicity of the structure makes it arguably easier to analyse than in the case of most ciphers.

Our contributions. In this chapter, we revisit the use of a *SHARK structure* for block cipher design and endeavour to find good matrices and appropriate algorithms to achieve both a linear mapping with very good diffusion and efficient software implementations that are not prone to timing attacks. To be more specific on this latter point, we target software running on 32 or 64-bit CPUs featuring an SIMD vector unit.

A possible way of making a mapping more efficient is to decrease the size of the field from $\mathbf{F}_{2^8}$ to $\mathbf{F}_{2^4}$. However, according to the *MDS conjecture*, there is no MDS code over $\mathbf{F}_{2^4}$ of length greater than seventeen [MS06]. Because a diffusion matrix of dimension n is typically obtained from a code of length $2n$, MDS matrices over $\mathbf{F}_{2^4}$ are therefore restricted to dimensions less than eight. Hence, the prospect of finding an MDS matrix over $\mathbf{F}_{2^4}$ diffusing on more than $8 \times 4 = 32$ bits is hopeless. Obviously, 32 bits is not enough for a large mapping *à la* SHARK. We must therefore search for codes with a slightly smaller minimum distance in the hope that they can be made longer.

Our proposed solution to this problem is to use *algebraic geometry codes*, as they precisely offer this tradeoff. One way of defining these codes is as evaluation codes on algebraic curves; thus our proposal brings a nice connection between these objects and symmetric cryptography: although elliptic and hyperelliptic curves are now commonplace in public-key cryptography, we show an unusual application of a hyperelliptic curve to the design of block ciphers. We present a specific code of length 32 and dimension 16 over $\mathbf{F}_{2^4}$ with minimum distance 15, which is only two less than what an MDS code would achieve; this is also much better than a random code for these alphabet, length and dimension, as such a code would typically have an expected minimum distance of 11. This lets us deriving a very good diffusion matrix on $16 \times 4 = 64$ bits in a straightforward way. Interestingly, this matrix can also be applied to vectors over an extension of $\mathbf{F}_{2^4}$ such as $\mathbf{F}_{2^8}$, while keeping the same good diffusion properties. This allows for instance to increase the diffusion to $16 \times 8 = 128$ bits.

We also study two simple yet efficient algorithms for implementing the matrix-vector multiplication needed in a SHARK structure, when a *vector permute* instruction is available. From one of these, we derive conditions on the matrix to make the matrix-vector product faster to compute, in the form of a cost function; we then search for matrices with a low cost, both randomly, and by using automorphisms of the code and of the hyperelliptic curve on which it is based. The use of codes automorphisms to derive efficient encoders is not new [HLS95, CL04], but it is not generally applied to the architecture and dimensions that we consider in our case.

We conclude the chapter by presenting examples of performance figures of assembly implementations of our algorithms when used as the linear mapping of a block cipher.

2 Preliminaries

We introduce the notation, definitions and background notions that are used in this chapter. We illustrate some of these with classical examples, such as *Reed-Solomon* codes. However, our goal is not to provide a detailed exposition on coding theory, and we refer the reader to any good textbook such as [Lin99] for a thorough treatment.

2.1 Notation

We use "$:=$" to denote equality by definition. We let $\mathbf{F}_q$ be the finite field with q elements, and in particular $\mathbf{F}_{2^m}$ be the field with 2^m elements. We often consider $\mathbf{F}_{2^4}$, and implicitly use this specific field if not mentioned otherwise. We use the representation $\mathbf{F}_2[\alpha]/\langle\alpha^4+\alpha+1\rangle$ for $\mathbf{F}_{2^4}$ when a specific one is needed, for instance for implementation purposes. We freely use "integer representation" for elements of $\mathbf{F}_{2^m} \cong \mathbf{F}_2[\alpha]/I(\alpha)$ by writing $n \in \{0\dots2^m-1\} = \sum_{i=0}^{m-1} a_i 2^i$ to represent the element $x \in \mathbf{F}_{2^m} = \sum_{i=0}^{m-1} a_i \alpha^i$. In the remainder of this section, and especially in Section 3, we usually consider an arbitrary algebraically closed field, written k; it will usually be clear from the context whether k is a field or the dimension of a linear code, see Definition 6.2 below. We let $\mathfrak{S}_n$ denote the group of permutations on n elements.

Bold variables denote vectors, in the sense of elements of a vector space, and subscripts are used to denote their i^{th} coordinate, starting from zero. For instance, let $\mathbf{x} := (1,2,7)$, then $\mathbf{x}_2 = 7$. If M is a matrix of n columns, we call $M_i := (M_{i,j},\ j = 0\dots n-1)$ the row vector formed from the coefficients of its i^{th} row.

Arrays, or tables, in the sense of software data structures, are denoted by regular variables such as x or T, and their elements are accessed by using square brackets. For instance, $T[i]$ is the i^{th} element of the table T, starting from zero.

The operator "$\wedge_n$" denotes the *logical and* between two n-bit values. We use $\mathbf{0}_n$ and $\mathbf{1}_n$ for binary vectors of length n all made of zeros and ones respectively. These may also freely be interpreted as elements of other vector spaces that will be clear from the context. The function $\mathbb{I}(\cdot)$ takes as input a set and returns one if it is non-empty, zero otherwise; $\#$ denotes the cardinality of a set. We may also use subscripts to show the base in which a number is written when it is not ten, *e.g.* 1010_2 is the number 10 written in base two.

2.2 Linear codes

Definition 6.1 (Hamming weight, Hamming distance). Let $\mathbf{x}$ be a vector of k^n. Its *Hamming weight* $\mathrm{wt}(\mathbf{x}) \in [0,\dots,n]$ is $\#\{\mathbf{x}_i, i = 0,\dots,n-1 \mid \mathbf{x}_i \neq 0\}$, the number of its coordinates which are non-zero. The *Hamming distance* $\mathrm{d}(\mathbf{x},\mathbf{y})$ between two vectors is defined as $\mathrm{wt}(\mathbf{x}-\mathbf{y})$.

Definition 6.2 (Linear code). A *linear code* of *length* n, *dimension* k and *minimal distance* d over the alphabet $\mathbf{F}_q$ is a k-dimensional linear subspace of $\mathbf{F}_q^n$ such that $\forall\, \mathbf{x},\mathbf{y} \in \mathcal{C},\ \mathbf{x} \neq \mathbf{y},\ \mathrm{d}(\mathbf{x},\mathbf{y}) \geq d$ and $\exists\, \mathbf{x},\mathbf{y} \in \mathcal{C},\ \mathrm{d}(\mathbf{x},\mathbf{y}) = d$. The last conditions can equivalently be expressed as $\mathbf{x} \neq \mathbf{0} \in \mathcal{C} \Rightarrow \mathrm{wt}(\mathbf{x}) \geq d$ and $\exists\, \mathbf{x} \in \mathcal{C}, \mathrm{wt}(\mathbf{x}) = d$. We use the usual notation to characterise codes: an $[n,k,d]_{\mathbf{F}_q}$ code $\mathcal{C}$ is a code of length n, dimension k and minimum distance d with symbols in $\mathbf{F}_q$. When d is unknown, we simply write $[n,k]_{\mathbf{F}_q}$.

We only consider linear codes in this manuscript and we will therefore omit this qualifier in the remainder of the text. We usually view a code as a set, and call *codewords* its elements. By an abuse of terminology, we may call *messages* the elements of $\mathbf{F}_q^k$.

Definition 6.3 (Dual code). Let $\mathcal{C}$ be an $[n,k,d]_{\mathbf{F}_q}$ code over $\mathbf{F}_q$ equipped with a scalar product $\langle\cdot,\cdot\rangle$. The *dual* $\mathcal{C}^\perp$ of $\mathcal{C}$ is defined as $\{\mathbf{x} \in \mathbf{F}_q | \forall \mathbf{y} \in \mathcal{C}, \langle \mathbf{x},\mathbf{y}\rangle = 0\}$.

A code $\mathcal{C}$ equal to its dual $\mathcal{C}^\perp$ is called *self-dual*.

Definition 6.4 (Generator matrix, systematic form). A *generator matrix* G of an $[n,k,d]_{\mathbf{F}_q}$ code $\mathcal{C}$ is a matrix of $\mathcal{M}_{k,n}(\mathbf{F}_q)$ such that $\mathcal{C} = \{\mathbf{x}G, \mathbf{x} \in \mathbf{F}_q^k\}$. It is in *systematic form* if it is of the form $(I_k\ A)$ with I_k the identity matrix of dimension k.

If $(I_k\ A)$ is a generator matrix for a code $\mathcal{C}$, then $(-A^t\ I_{n-k})$, or equivalently $(I_{n-k}\ -A^t)$, is a generator matrix for its dual $\mathcal{C}^\perp$. If $\mathcal{C}$ is self-dual, A is orthogonal, *i.e.* $A \cdot A^t = I$

Definition 6.5 (MDS code, MDS matrix). An $[n,k,d]_{\mathbf{F}_q}$ code $\mathcal{C}$ is called *maximum distance separable*, or simply *MDS*, if $d = n-k+1$. By abuse of definition, if $n = 2k$ and $(I_k\ A)$ is a generator matrix of $\mathcal{C}$, we call A an *MDS matrix*.

The minimum distance attained by an MDS code corresponds to the Singleton bound. In the case of a linear code defined with a systematic generator matrix, it is easy to see that $n-k+1$ is indeed the maximum possible weight for any row.

A useful alternative characterisation of MDS matrices is given by the following theorem.

Theorem 6.1 ([MS06, Chapter 11, Theorem 8]). *A matrix M is MDS if and only if all of its minors are non-zero,* i.e. *all the square sub-matrices of M are invertible.*

A consequence is that the dual of an MDS code is also MDS.

An important conjecture on MDS codes is the following:

Conjecture 6.1 (MDS conjecture [MS06, Chapter 11, Section 7]). *There is no MDS code $[n,k,n-k+1]_{\mathbf{F}_q}$ of length $n > B$, where $B = q+2$ when q is even and $k = 3$ or $k = q-1$, and $B = q+1$ in all other cases.*

The next definition rephrases and generalises some of the above concepts in a way that is more suitable to cryptographic applications. We assume in this case that $\mathbf{F}_q = \mathbf{F}_{2^m}$ for some m.

Definition 6.6 (Branch number [Dae95]). Let A be the matrix of a linear mapping over $\mathbf{F}_{2^m}$. The *differential branch number* of A is equal to $\min_{\mathbf{x}\neq\mathbf{0}}(\mathrm{wt}(\mathbf{x}) + \mathrm{wt}(\mathbf{x}A))$, and the *linear branch number* of A is equal to $\min_{\mathbf{x}\neq\mathbf{0}}(\mathrm{wt}(\mathbf{x}) + \mathrm{wt}(\mathbf{x}A^t))$.

If A is such that $(I_k\ A)$ is a generator matrix of a code of minimum distance d whose dual has minimum distance d', then A has a differential (resp. linear) branch number of d (resp. d').

2.2.1 Evaluation codes

The algebraic geometry codes that we use in this chapter are conveniently defined as instantiations of the general framework of *evaluation codes*, for which we give a brief overview. We start with a general definition:

Definition 6.7 (Evaluation code). Let $\mathcal{F}$ be an $\mathbf{F}_q$-vector space of dimension k of functions $f : \mathcal{D} \to \mathbf{F}_q$ from a domain $\mathcal{D}$ to a finite field $\mathbf{F}_q$; we call $\mathcal{F}$ the *function space.* Let $\mathcal{P} := (P_0, \dots, P_{n-1})$ be an ordered subset of $\mathcal{D}$ of cardinality n ; we call $\mathcal{P}$ the *evaluation domain.* Let $\mathrm{Ev}_\mathcal{P} : \mathcal{F} \to \mathbf{F}_q^n$ be the *evaluation map* defined as $\mathrm{Ev}_\mathcal{P}(f) := (f(P_0), \dots, f(P_{n-1}))$. The *evaluation code* $\mathcal{C}_{ev}(\mathcal{F}, \mathcal{P})$ for an injective evaluation map $\mathrm{Ev}_\mathcal{P}$ is defined as $\{\mathrm{Ev}_\mathcal{P}(f) \mid f \in \mathcal{F}\}$. It is an $[n, k]_{\mathbf{F}_q}$ code.

A generator matrix G of an evaluation code can easily be constructed by evaluating a basis for $\mathcal{F}$ on $\mathcal{P}$. Call $(f_0, \dots, f_{k-1})$ such a basis, then $G = (\mathrm{Ev}_\mathcal{P}(f_0), \dots, \mathrm{Ev}_\mathcal{P}(f_{k-1}))$. If the code admits a systematic generator matrix, *i.e.* its first k columns span a subspace of rank k, it can simply be obtained by computing the reduced row-echelon form of G.

We use ordered sets for the evaluation domain $\mathcal{P}$ in this definition. Although the order that is chosen does not impact the parameters of the code (hence we may sometimes abuse our definition and talk about *the* code for any of the equivalent codes based on the same $\mathcal{F}$ and $\mathcal{D}$), it may have an influence on the performance of explicit instantiations, through *e.g.* properties of the generator matrices: for instance, one evaluation domain $\mathcal{P}$ may be such that its associated generator matrix cannot be put in systematic form, but there is always a permutation $\mathcal{P}'$ of $\mathcal{P}$ such that it is the case. Most of the work in this chapter is actually based on this fact.

The probably best-known evaluation codes are the *Reed-Solomon codes* (“RS codes”):

Definition 6.8 (Reed-Solomon codes). An $[n, k]_{\mathbf{F}_q}$ *Reed-Solomon* code is an evaluation code obtained by taking $\mathcal{F}$ to be the polynomials $\mathbf{F}_q[x]$ of degree less than $k - 1$ and $\mathcal{P}$ any ordered subset of $\mathbf{F}_q$ of size n.

As we must have $n \geq k$ by definition of a code, all polynomials of $\mathcal{F}$ can be uniquely interpolated given their evaluations on n points, hence $\mathrm{Ev}_\mathcal{P}$ is injective and Reed-Solomon codes are well-defined. Furthermore, for any $\mathcal{P}$, any non-zero polynomial of $\mathcal{F}$ is zero on at most $k-1$ positions. The minimal distance of a Reed-Solomon code is thus $n-(k-1)$, which makes them MDS codes.

3 Algebraic geometry codes

We now present *algebraic geometry codes* ("AG codes"), which are the main object used in this chapter, and we show how they can give rise to diffusion matrices with interesting parameters. We introduce a few notions of algebra and geometry along the way in order to be able to faithfully describe the codes we will be using. We refer the reader to the relevant chapters of [Lin99, TVN07, Sti09, Ful08] for a much more detailed and rigorous treatment of the subject.

AG codes as a generalisation of RS codes. We first give an intuition of the construction used in AG codes. We have seen that RS codes are MDS and that they can be instantiated for many lengths and dimensions. One drawback of the construction however is that the maximal length of the code is limited by the cardinality of the alphabet $\mathbf{F}_q$: we cannot evaluate the functions on more points than there are elements in $\mathbf{F}_q$ without introducing some repetition; we can still slightly increase this maximal length by one, and very rarely two, by defining an *extended* RS code, which is still MDS. However, by the MDS conjecture, we do not expect to be able to do better than that. Under this light, it may seem natural to desire some sort of tradeoff between the maximal length of a code construction and its *designed minimal distance* $d^* \leq d$.

A natural way to increase the maximal length of an evaluation code is to consider functions over domains of higher dimension than the "line" used in RS codes. For instance, one could take $\mathcal{D}$ to be the m-dimensional space $\mathbf{F}_q^m$, and $\mathcal{F}$ to be the m-variate polynomials of $\mathbf{F}_q[x_0, \ldots, x_{n-1}]$ less than a certain degree r. This defines *Reed-Muller codes* ("RM codes") of order r. Although this construction successfully yields codes longer than RS codes, they are not asymptotically good, in the sense that this construction does not allow to have code parameters where neither of d/n and k/n tends to zero when n goes to infinity.

An alternative approach to construct longer variants of RS codes is the one followed by AG codes. Seeing RS codes as working with a line of genus zero, the idea is to take as evaluation domains the points of plane curves of higher genus, which can be assimilated to a subset of $\mathbf{F}_q^2$. A geometrically meaningful way to choose the function spaces $\mathcal{F}$ is then to consider certain subspaces of the function fields of the curves. We will see that for well-chosen $\mathcal{F}$, this construction leads to codes of parameters $[n, k, d \geq n-k+1-g]_{\mathbf{F}_q}$ for curves of genus g. As the maximal number of points on a curve is roughly increasing with the genus, this construction gives us the sort of tradeoff we were searching for, and also yields asymptotically good codes.

3.1 Selected notions from algebraic geometry

We will be needing a few tools from algebraic geometry to make our description of AG codes explicit. We only introduce here the notions that are relevant to this specific application, largely ignoring the wider picture. Note that unlike the remainder of this chapter, the underlying fields used in the definitions are not necessarily finite.

We will describe AG codes as evaluation codes, which means that we need to be able first to describe the evaluation domains, and second to define the function spaces $\mathcal{F}$ associated with given evaluation domains. Both of these need to be defined explicitly, if we want the codes to be useful in practice; in particular, this means that we need to be able to compute the bases of the function spaces used in the constructions.

The construction of the examples of AG codes that we later consider can eventually be described in a purely affine way. However, in order to justify them, we need to be able to reason about points at infinity. Thus we start with the following:

Definition 6.9 (Affine space, projective space)**.** The n-dimensional *affine space* $\mathbf{A}^n(k)$ over a field k is a set of *points* formed by n-tuples over k, $P := (x_0, \dots, x_{n-1})$, $x_i \in k$. The n-dimensional *projective space* $\mathbf{P}^n(k)$ over k is a set of points formed by equivalence classes of $(n+1)$-tuples $P := (x_0 : \dots : x_n)$, $x_i \in k$, where not all x_is are zero, and the equivalence relation $\sim$ is defined by $(x_0 : \dots : x_n) \sim (\lambda x_0 : \dots : \lambda x_n)$, $\lambda \in k^*$. In the usual case where $n = 2$, we let $x := x_0$, $y := x_1$ and $z := x_2$.
The space $\mathbf{A}^n(k)$ can be embedded into $\mathbf{P}^n(k)$ in a natural way as $(x_0, \dots, x_{n-1}) \mapsto (x_0 : \dots : x_{n-1} : 1)$. The set of projective points $(x_0 : \dots : x_{n-1} : 0)$ is called the *hyperplane at infinity.*

We recall the following definitions about ideals.

Definition 6.10 (Ideal, prime ideal, maximal ideal, principal ideal)**.** An *ideal* $\mathcal{I}$ of a ring $\mathcal{R}$ is a subset of $\mathcal{R}$ closed by addition and that is absorbing by multiplication: $a, b \in \mathcal{I} \Rightarrow a + b \in \mathcal{I}$; $a \in \mathcal{I}, b \in \mathcal{R} \Rightarrow ab \in \mathcal{I}$.
An ideal $\mathcal{I}$ is *prime* if for any a, b in $\mathcal{R}$ such that $ab \in \mathcal{I}$, then $a \in \mathcal{I}$ or $b \in \mathcal{I}$.
An ideal $\mathcal{I} \subsetneq \mathcal{R}$ is *maximal* if there is no ideal $\mathcal{J}$ such that $\mathcal{I} \subsetneq \mathcal{J} \subsetneq \mathcal{R}$.
An ideal $\mathcal{I}$ is *principal* if it is generated by a single element: $\exists\, a \in \mathcal{I}$ s.t. $\mathcal{I} = a\mathcal{R} = \{ar,\ r \in \mathcal{R}\}$.

We may now define projective varieties.

Definition 6.11 (Projective variety)**.** A *projective variety* in the projective space $\mathbf{P}^n(k)$ over an algebraically closed field k is the set of common zeros of a prime ideal $\mathcal{I} := \langle f_0, \dots, f_m \rangle$ of homogeneous polynomials of $k[x_0, \dots, x_n]$, *i.e.* polynomials whose monomials all have the same degree.

A *projective curve* is a projective variety of dimension one, where the dimension n of a variety V can formally be defined as the maximal length of the inclusion chain $V = V_0 \supsetneq \dots \supsetneq V_n \neq \varnothing$, with V_i closed in V_{i-1}, for the *Zariski topology* for all $i \in \{1, \dots, n\}$. We will not need this definition in this chapter, as we only consider dimension-one varieties in $\mathbf{P}^2$, *i.e.* plane curves, which are defined by a single homogeneous polynomial E.

Remarks

1. The condition that k be algebraically closed is not as restrictive as it may seem. Say that we wish to work in $\mathbf{F}_q$, with a curve $\mathcal{X}$ defined by an equation E over $\mathbf{F}_q$ or one of its subfields. Then, although we define $\mathcal{X}$ over the algebraic closure of $\mathbf{F}_q$, we will only be interested in $\mathcal{X}(\mathbf{F}_q)$, the points of $\mathcal{X}$ that lie in $\mathbf{F}_q$: its $\mathbf{F}_q$-*rational points*, or simply rational points. For instance, we may wish to only consider curves which have a large number of rational points.

2. For a given projective variety, say $\mathcal{X}$ defined by $E(x, y, z)$, one can consider the corresponding affine variety defined as the zeros of "$E(x, y, 1)$, $E \in k[x, y]$", the "de-homogenised" of E. It will have the same points as $\mathcal{X}$, in the sense of the embedding of Definition 6.9, minus its potential points at infinity. Similarly, an affine variety can be extended to a projective closure.

Our first step towards defining suitable function spaces $\mathcal{F}$ is to define the function field of a variety. We give two definitions, one in the affine and one in the projective case.

Definition 6.12 (Coordinate ring, function field of an affine variety)**.** Let $\mathcal{X}$ be an affine variety in $\mathbf{A}^n(k)$ defined by the ideal $\mathcal{I}$. The *coordinate ring* $k[\mathcal{X}]$ of $\mathcal{X}$ is $k[x_0, \ldots, x_{n-1}]/\mathcal{I}$. The *function field* $k(\mathcal{X})$ of $\mathcal{X}$ is the quotient field of $k[\mathcal{X}]$.

Definition 6.13 (Function field of a projective variety)**.** Let $\mathcal{X}$ be an affine variety in $\mathbf{P}^n(k)$ defined by the ideal $\mathcal{I}$. Let F, G be homogeneous polynomials of equal degrees in $k[x_0, \ldots, x_n]$ with $G \notin \mathcal{I}$. The *function field* $k(\mathcal{X})$ of $\mathcal{X}$ is the set of *rational functions* F/G modulo $\mathcal{I}'$, where $\mathcal{I}' = \{F/G \mid F \in \mathcal{I}\}$.
A rational function F/G is *regular* at P, point of $\mathcal{X}$, if $G(P) \neq 0$.

We will need to be able to talk about the behaviour of functions at specific points of a curve. For this, we first use the following:

Definition 6.14 (Local ring of a point)**.** The *local ring* $\mathcal{O}_P$ of a point P of a variety $\mathcal{X}$ is the set of rational functions regular at P. It has a unique maximal ideal $\mathfrak{m}_P := \{f \in \mathcal{O}_P \mid f(P) = 0\}$.

The local ring and its maximal ideal are in particular useful to define the *tangent space* at a point P, which in turn allows to define smooth and singular points, depending on its dimension. We do not give the general definition here as it is more convenient in our case to define smoothness at a point from properties of the corresponding local ring. Indeed, we have the following convenient direct characterisation in the specific case of curves:

Definition 6.15 (Smooth and singular points of a curve, local parameter)**.** A point P of a curve $\mathcal{X}$ is *smooth* or *non-singular* iff $\mathfrak{m}_P$ is a principal ideal. Any generator of the ideal $t_P \in \mathcal{O}_P$ s.t. $\mathfrak{m}_P = t_P\mathcal{O}_P$ is called a *local parameter* at P.
A point that is not smooth is called *singular*.

To build an AG code from a curve $\mathcal{X}$, we will require that it is smooth, *i.e.* that none of its points are singular. We only consider smooth curves until Section 3.1.1.

The local parameters are useful in defining the order of zeros and poles of arbitrary rational functions at a given point. Let $f \neq 0$ be a rational function regular at P and t_P be a local parameter at P. The *order* of f at P is:

$$\mathrm{ord}_P(f) := \max\{k \mid f \in t_P^k\mathcal{O}_P,\ f \notin t_P^{k+1}\mathcal{O}_P\}.$$

This can be extended to arbitrary rational functions by writing them as quotients of functions of $\mathcal{O}_P$:

$$\mathrm{ord}_P(f/g) = \mathrm{ord}_P(f) - \mathrm{ord}_P(g),\ f, g \in \mathcal{O}_P.$$

For a function f, if $\mathrm{ord}_P(f) \geq 0$, we call this value the *zero order* of f at P; if $\mathrm{ord}_P(f) < 0$ we call $-\mathrm{ord}_P(f)$ the *pole order* of f at P. It is also useful to notice that $\mathrm{ord}(fg) = \mathrm{ord}(f) + \mathrm{ord}(g)$ and that $\mathrm{ord}(1/f) = -\mathrm{ord}(f)$.

In effect, ord_P defines a *discrete valuation* of functions of $k(\mathcal{X})$, associated with the point P, *i.e.* $\mathcal{O}_P$ is a *valuation ring*:

Definition 6.16 (Valuation ring)**.** A ring $\mathcal{O}$ is a *valuation ring* of $k(\mathcal{X})$ if $k \subsetneq \mathcal{O} \subsetneq k(\mathcal{X})$ and for every $f \in k(\mathcal{X})$, $f \in \mathcal{O}$ or $f^{-1} \in \mathcal{O}$.

We will see shortly that the ability to define valuations at any point is essential in the definition of our function space $\mathcal{F}$; this is why we require the curve to be non-singular, as a local ring is a valuation ring iff P is smooth.

The last main objects that we introduce are the divisors on a curve.

Definition 6.17 (Divisor)**.** A *divisor* on a smooth curve $\mathcal{X}$ is a formal finite sum of points P_i of $\mathcal{X}$: $D := \sum a_iP_i$, $a_i \in \mathbf{Z}$, $P_i \in \mathcal{X}$. The *support* of a divisor is the set of points P_i with $a_i \neq 0$.
The set $\mathrm{Div}(\mathcal{X})$ of divisors on $\mathcal{X}$ with formal addition and negation of divisors forms a group, *i.e.* $\sum a_iP_i + \sum b_iP_i := \sum(a_i + b_i)P_i$; $-\sum a_iP_i := \sum -a_iP_i$.
A divisor $D := \sum a_iP_i$ is called *effective* ($D \geq 0$) if $\forall i,\ a_i \geq 0$.
The *degree* $\deg(D)$ of $D = \sum a_iP_i$ is defined as $\sum a_i$.

Definition 6.18 (Divisor of a rational function (principal divisor))**.** The divisor (f) of a rational function $f \in k(\mathcal{X})$, $f \neq 0$, is defined as $\sum \mathrm{ord}_P(f)P$. A divisor D equal to (f) for some function f is called a *principal divisor*.

Principal divisors form a subgroup of $\mathrm{Div}(\mathcal{X})$. It can be shown that they are always of degree zero, which is equivalent to saying that a rational function always has the same number of poles and zeros, counted with multiplicity.

Definition 6.19 (Space associated with a divisor)**.** Let $D \in \mathrm{Div}(\mathcal{X})$. We define $\mathcal{L}(D)$, the *space associated with* D (or *Riemann-Roch space*) as:

$$\mathcal{L}(D) = \{f \in k(\mathcal{X})^* \mid (f) + D \geq 0\} \cup \{0\}.$$

This forms a k-vector space; we write $\ell(D)$ its dimension.

This is finally the kind of function space we will be using to define AG codes. For a divisor $D := \sum a_i P_i - \sum b_j P_j$, $a_i, b_j \geq 0$, a simple reformulation of $\mathcal{L}(D)$ is to say that it consists of the zero function and of all the functions which may have poles only in the P_is of order at most a_i, and which must have zeros in all the P_js of order at least b_j.

It can be shown that for any D, $\mathcal{L}(D)$ is finite-dimensional. In fact, this is a consequence of two major theorems the first from Riemann and the second being an extension due to Roch, which allow in particular to compute the explicit value $\ell(D)$ for a given divisor D ; this is expressed by the following corollary:

Corollary 6.1 (Corollary of Riemann and Roch)**.** *Let D be a divisor on a curve of* genus *g, then $\ell(D) \geq \deg(D) + 1 - g$, with equality when* $\deg(D) > 2g - 2$.

The formulation of this theorem uses the notion of *genus* of a curve, that we have not defined yet. A possible, purely algebraic definition is precisely to take $g(\mathcal{X}) := \max\{\deg(D) - \ell(D) + 1,\ D \in \mathrm{Div}(\mathcal{X})\}$. Although this is not very satisfying, we will not provide a better definition here, and computing the genus in this way is not as hard as it may seem; in practice, the genus of the curves we use will be known beforehand.

3.1.1 Working with singular curves

In the last few paragraphs, we have assumed that we were working with a smooth curve. We justified this by the need of being able to define a valuation for functions at any point of the curve, which allowed us to define divisors and their associated vector spaces. Unfortunately, some of the curves we may wish to use in practice are not smooth. We briefly and informally sketch here how it is possible to nonetheless work with some of these curves. We refer the reader to *e.g.* Fulton [Ful08, Chapter 7] for a serious treatment of the matter.

A first remark is that what effectively matters in the definition of the function space is not so much the points on the curves as their associated valuation rings. It is in fact possible to define entirely what we have presented above in a purely algebraic way, as is done for instance by Stichtenoth [Sti09]. In this case, the notion of points is replaced by the one of *places* of a function field, written F/k, not necessarily associated with a curve, where a place is simply the necessarily unique maximal ideal of a valuation ring of the function field:

Definition 6.20 (Place of a function field)**.** A place P of a function field F/k is the maximal ideal of a valuation ring $\mathcal{O}$ of F/k.

Divisors can then be redefined as sums of places instead of sums of points.

For a smooth curve $\mathcal{X}$, there is a bijection between its points and the valuation rings (the places) of $k(\mathcal{X})$, and the theory as presented above is sufficient. On the other hand, if $\mathcal{X}$ is singular, we would like to be able to find a way of additionally defining places at its singularities. This can be done by constructing a *non-singular model* $\widetilde{\mathcal{X}}$ of $\mathcal{X}$, and taking as places of $\mathcal{X}$ the points of $\widetilde{\mathcal{X}}$. A useful theorem is that for a projective curve $\mathcal{X}$, there is a unique (up to isomorphism) non-singular curve $\widetilde{\mathcal{X}}$ that is birationally equivalent to it, which means that there are *rational maps* from $\mathcal{X}$ to $\widetilde{\mathcal{X}}$ and vice-versa, that compose to the identity wherever they are defined, a rational map $\mathcal{X} \subseteq \mathbf{P}^m \to \mathbf{P}^n$ being an $(n+1)$-tuple of homogeneous polynomials s.t. the polynomials do not jointly vanish at any point of the curve. Furthermore, the function fields $k(\mathcal{X})$ and $k(\widetilde{\mathcal{X}})$ are the same, up to isomorphism, so defining the places of $\mathcal{X}$ in this way is meaningful. We will not address here the problem of computing these non-singular models *per se*.

The map $\mathcal{X} \to \widetilde{\mathcal{X}}$ may not be defined on every point of $\mathcal{X}$, but the number of points where it is not defined is finite; this makes sense, as a curve has at most finitely many singular points. We say that a place is *centered* at a point P if P is its image by the inverse map $\widetilde{\mathcal{X}} \to \mathcal{X}$. Every non-singular point of $\mathcal{X}$ has exactly one place centered at it, while finitely many places may be centered at singular points.

We conclude by defining curves that are said to be in *special position*, cf. *e.g.* [SH95], which are convenient to use for several reasons.

Definition 6.21 (Curve in special position)**.** We recall our notation: a point $P \in \mathbf{P}^2(k)$ is written $(x : y : z)$, and is at infinity iff $z = 0$. A projective plane curve $\mathcal{X}$ is in *special position* if:

1. it has exactly one point P_∞ at infinity;
2. there is exactly one place centered at P_∞;
3. the affine curve $\mathcal{X}_A := \mathcal{X} \backslash P_\infty$ is smooth;
4. the pole order of x/z and y/z at P_∞ are not equal.

Curves in special positions are useful as there is a bijection between their places and their (potentially singular) points; thus they can always be considered "as if smooth" w.r.t. to the discussions of this section. Additionally, if one considers the space $\mathcal{L}(aP_\infty)$, $a > 0$, of functions having poles only at the point at infinity, its elements can be mapped to polynomials of the coordinate ring of $\mathcal{X}_A$. In other words, one can perform all the computations in $\mathcal{L}(aP_\infty)$ with the affine curve only; we discuss how to explicitly compute bases for such spaces next.

The main curve used in this chapter is a singular curve in special position.

3.1.2 Computation of a basis of $\mathcal{L}(aP_\infty)$ for a curve in special position

We conclude this brief presentation of algebraic curves and their function fields by showing how to compute a basis of $\mathcal{L}(aP_\infty)$, $a > 0$ for a plane curve in special position.

Consider $\mathcal{X}$ defined by a homogeneous equation $E(x,y,z)$. From a local parameterisation at the place P_∞, we can compute the pole order of $f \coloneqq x/z$ and $g \coloneqq y/z$ at P_∞, which is enough to determine the pole order of any fraction of x, y and z. An element of $\mathcal{L}(aP_\infty)$ is then a linear combination of monomials of the form $f^\alpha g^\beta$ such that we always have $\alpha \operatorname{ord}_{P_\infty}(f) + \beta \operatorname{ord}_{P_\infty}(g) \le a$, *i.e.* the pole order at P_∞ of the monomials is less than a. A generating family for $\mathcal{L}(aP_\infty)$ is thus $\{f^i g^j \mid i \times \operatorname{ord}_{P_\infty}(f) + j \times \operatorname{ord}_{P_\infty}(g) \le a\}$; if we keep only a single monomial with any given pole order, it forms a basis. As P_∞ is the only place at infinity, we can switch to an affine view for all computations by considering points in affine coordinates and dehomogenising E and the basis functions; notably, the latter effectively become polynomials rather than rational fractions. Indeed, a generating family for $\mathcal{L}(aP_\infty) \subseteq k[\mathcal{X}]$ is formed by $\{x^i y^j \mid i \times \operatorname{ord}_{P_\infty}(f) + j \times \operatorname{ord}_{P_\infty}(g) \le a\}$.

We conclude by presenting a simple technique to compute the pole order of x/z and y/z at the place at infinity for a curve in special position, in a way that does not require to explicitly find a local parameter at P_∞. In this setting, we know by definition that the valuation $\operatorname{ord}_{P_\infty}$ is well-defined and that it is negative and distinct for x/z and y/z. Thus $\exists\, \alpha, \beta \in \mathbf{N}^*$ s.t. $(x/z)^\beta (y/z)^{-\alpha}$ is regular and non-zero at P_∞. The smallest such α and β are the pole orders of x/z and y/z respectively.

We give two examples of a computation with this technique, corresponding to the first two curves presented in Section 3.2.

Example 6.1 ($\boldsymbol{x^2z + xz^2 = y^3 + yz^2}$). We consider the curve of equation $E(x,y,z) \coloneqq x^2z + xz^2 = y^3 + yz^2$ over a field of characteristic two. It has a unique place at infinity $P_\infty \coloneqq [1:0:0]$, *i.e.* a unique projective point with $z = 0$ that verifies the curve equation; indeed, setting z to zero in E we see that we must have $y = 0$ as well. From the curve equation, we may guess 3 and 2 to be the orders of x/z and y/z, as the fraction x^2z^3/y^3z^2 seems to be the one of least degree that can be simplified using E. For this guess to be validated, we need to show that x^2z^3/y^3z^2 is regular and non-zero at P_∞, which can be done by finding a suitable equivalent expression for this function.

Using E, we may rewrite x^2z^3 in the four following ways:

1. $z^2(x^2z)$;
2. $z^2(y^3 + yz^2 + xz^2)$;
3. $xz(xz^2)$;
4. $xz(y^3 + yz^2 + x^2z)$.

Similarly, y^3z^2 can be rewritten as:

i. $z^2(y^3)$;

ii. $z^2(x^2z + xz^2 + yz^2)$;

iii. $y^2(yz^2)$;

iv. $y^2(y^3 + x^2z + xz^2)$.

We can then notice that "1/ii" $= z^2(x^2z)/z^2(x^2z + xz^2 + yz^2)$ simplifies to $x^2z/(x^2z + xz^2 + yz^2) = x^2/(x^2 + xz + yz)$, which evaluates to 1 at $[1:0:0]$. Hence we indeed have $\mathrm{ord}_{P_\infty}(x/z) = 3$ and $\mathrm{ord}_{P_\infty}(y/z) = 2$.

Example 6.2 **($x^5 = y^2z^3 + yz^4$).** We consider the curve of equation $E(x, y, z) := x^5 = y^2z^3 + yz^4$ over a field of characteristic two. It has a unique place at infinity $P_\infty := [0:1:0]$. As in Example 6.1, we may guess that x/z and y/z have order 2 and 5 respectively. We thus consider the fraction x^5z^2/y^2z^5. This can be rewritten as $(y^2z^5 + yz^6)/y^2z^5$ which readily simplifies to $(y^2 + yz)/y^2 = (y + z)/y$ which evaluates to 1 at $[0:1:0]$. Hence we indeed have $\mathrm{ord}_{P_\infty}(x/z) = 2$ and $\mathrm{ord}_{P_\infty}(y/z) = 5$.

3.2 Construction of AG codes

We are now ready to define the family of AG codes that we use in the remainder of this chapter. We refer to [TVN07, Chapter 4] for the description of other ways of defining codes from algebraic geometry.

Let $\mathcal{X}$ be a smooth projective curve defined by an equation in $\mathbf{F}_q$ such that the set of its rational places $\mathcal{X}(\mathbf{F}_q)$ is non-empty. Let $D \in \mathrm{Div}(\mathcal{X})$ be a divisor on $\mathcal{X}$ and $\mathcal{P} := (P_0, \dots, P_{n-1}) \subsetneq \mathcal{X}(\mathbf{F}_q)$ be n places not in the support of D. This defines a code $\mathcal{C}_{ev}(\mathcal{L}(D), \mathcal{P})$ through the evaluation map $\mathrm{Ev}_{\mathcal{P}}(f) := (f(P_0), \dots, f(P_{n-1}))$, $f \in \mathcal{L}(D)$.

For the definition to be valid, the evaluation map from $\mathcal{L}(D)$ at $\mathcal{P}$ has to be injective, *i.e.* its kernel must be equal to $\{0\}$. This condition can easily be expressed in terms of divisors; if we call $S := P_0 + \dots + P_{n-1}$ the divisor of the evaluation support, we need $\mathcal{L}(D - S)$ (*i.e.* functions of $\mathcal{L}(D)$ with zeros in all of the P_is, *i.e.* the kernel of Ev) to be reduced to the zero function. One can see that this will always be the case as long as $\deg(D) < \deg(S)$.

Assuming that we defined a code using this construction, one of our main concerns would be to determine its parameters k and d, or at least to provide an estimate for them. By definition of the minimal distance, $n-d$ is the maximal number of zeros that a function of $\mathcal{L}(D)$ can have over $\mathcal{P}$; we already showed that the latter is upper-bounded by $\deg(D)$, so we have $d \geq n - \deg(D)$. This lower-bound $n - \deg(D)$ for d is the designed minimum distance d^* for the code.

The dimension of the code is simply $\ell(D)$, which can be computed from the Riemann-Roch theorem. In particular, for $\deg(D) > 2g - 2$, g the genus of $\mathcal{X}$, we have $k = \ell(D) = \deg(D) + 1 - g$. We can then rephrase the lower-bound on d in terms of k as $d \geq n - k + 1 - g$, which is indeed what we promised at the beginning of the section.

Deriving a lower bound on the parameters of an algebraic geometry code as we just did is quite straightforward, but the overall theory is much richer. A useful result in particular is that the dual of an AG code from the family we described is also an AG code whose parameters are related through the same equation $d \geq n - k + 1 - g$ [TVN07, Chapter 4]. This is an interesting property in our context, as it means that diffusion matrices derived from AG codes have identical differential and linear branch numbers.

3.2.1 Examples

We give three concrete examples of AG codes, the second of which being the code used to define the diffusion matrices presented in this chapter. All of the examples are built from curves with a maximal number of $\mathbf{F}_{2^4}$-rational points given their respective genus.

Example 6.3 (An AG code from a curve of genus 1). Let $\mathcal{X}$ be the plane projective curve of genus 1 of equation $x^2z + xz^2 = y^3 + yz^2$ defined over $\mathbf{F}_{2^4}$. It is smooth, in special position, with a unique point at infinity $P_\infty := [1:0:0]$, and it has 24 $\mathbf{F}_{2^4}$-rational affine points. The pole order of x/z and y/z at P_∞ is 3 and 2 respectively, as computed in Example 6.1.

Let $D := 12P_\infty$. From the Riemann-Roch theorem, the dimension $\ell(D)$ of $\mathcal{L}(D)$ is $12 + 1 - g = 12$. We can thus define a $[24, 12]_{\mathbf{F}_{2^4}}$ AG code $\mathcal{C}_{ev}(\mathcal{L}(D), \mathcal{P})$ with $\mathcal{P}$ the affine rational points of $\mathcal{X}$ put in any order. A generating matrix for this code can be computed by evaluating a basis of $\mathcal{L}(D)$ on $\mathcal{P}$, one such basis being for instance $(1, y, x, y^2, xy, y^3, xy^2, y^4, xy^3, y^5, xy^4, y^6)$.

This code and its dual have minimum distance at least 12. Either code can thus be used to define a diffusion matrix of dimension 12 over $\mathbf{F}_{2^4}$, which diffuses over $12 \times 4 = 48$ bits and has differential and linear branch number at least 12 (in fact exactly 12).

Example 6.4 (An AG code from a curve of genus 2). Let $\mathcal{X}$ be the plane projective curve of genus 2 of equation $x^5 = y^2z^3 + yz^4$ defined over $\mathbf{F}_{2^4}$. It is singular, in special position, with a unique point at infinity $P_\infty := [0 : 1 : 0]$ at which a unique place is centered. It has 32 $\mathbf{F}_{2^4}$-rational affine points. The pole order of x/z and y/z at P_∞ is 2 and 5 respectively, as computed in Example 6.2.

Let $D := 17P_\infty$. From the Riemann-Roch theorem, the dimension $\ell(D)$ of $\mathcal{L}(D)$ is $17+1-g = 16$. We can thus define a $[32, 16]_{\mathbf{F}_{2^4}}$ AG code $\mathcal{C}_{ev}(\mathcal{L}(D), \mathcal{P})$ with $\mathcal{P}$ the affine rational points of $\mathcal{X}$ put in any order.

This code is self-dual and has minimum distance at least 15. It can thus be used to define an orthogonal diffusion matrix of dimension 16 over $\mathbf{F}_{2^4}$, which diffuses over $16 \times 4 = 64$ bits and has differential and linear branch number at least 15 (in fact exactly 15).

Example 6.5 (An AG code from a curve of genus 6). Let $\mathcal{X}$ be the plane projective curve of genus 6 of equation $x^5 = y^4z + yz^4$ defined over $\mathbf{F}_{2^4}$. It is smooth, in special position, with a unique point at infinity $P_\infty := [0 : 1 : 0]$ at which a unique place is centered. It has 64 $\mathbf{F}_{2^4}$-rational affine points. The pole order of x/z and y/z at P_∞ is 4 and 5 respectively.

Let $D := 37P_\infty$. From the Riemann-Roch theorem, the dimension $\ell(D)$ of $\mathcal{L}(D)$ is $37+1-g = 32$. We can thus define a $[64, 32]_{\mathbf{F}_{2^4}}$ AG code $\mathcal{C}_{ev}(\mathcal{L}(D), \mathcal{P})$ with $\mathcal{P}$ the affine rational points of $\mathcal{X}$ put in any order.

This code is self-dual and has minimum distance at least 27. It can thus be used to define an orthogonal diffusion matrix of dimension 32 over $\mathbf{F}_{2^4}$, which diffuses over $32 \times 4 = 128$ bits and has differential and linear branch number at least 27 (in fact exactly 27).

4 Efficient algorithms for matrix-vector multiplication

We now turn to implementation considerations, and present two algorithms for matrix-vector multiplication in $\mathcal{M}_n(\mathbf{F}_{2^4})$. We let the dimension n be equal to 16 in this presentation, but the algorithms apply equally well to other close values.

Targeted architecture. The algorithms in this section target CPUs featuring vector instructions, including in particular a *vector shuffle* instruction such as Intel's `pshufb` from the SSSE3 instruction set extension [Int14]. These instructions are now widespread and have already been used successfully in fast cryptographic implementations, see *e.g.* [Ham09, SMMK12, BGLP13]. We mostly considered SSSE3 when designing the algorithms, but other processor architectures do feature vector instructions. This is for instance the case of ARM's NEON extensions, which may also yield efficient implementations, see *e.g.* [BS12]. We do not consider these explicitly in this chapter, however.

Because it plays an important role in our algorithms, we briefly recall the semantics of `pshufb`. This instruction takes two 128-bit inputs[1]. The first (the destination operand) is an `xmm` SSE vector register which logically represents a vector of sixteen bytes. The second (the source operand) is either a similar `xmm` register, or a 128-bit memory location. The result of calling `pshufb` x y is to overwrite the input x with the vector x' defined by:

$$x'[i] = \begin{cases} x[\lfloor y[i] \rfloor_4] & \text{if the most significant bit of } y[i] \text{ is not set} \\ 0 & \text{otherwise} \end{cases}, \; 0 \leq i < 16$$

where $\lfloor \cdot \rfloor_4$ denotes truncation to the four least significant bits. This instruction allows to arbitrarily *shuffle* a vector according to a mask, with possible repetition and omission of some of the vector values. We will use the word *shuffle* with this precise meaning in the remainder of this chapter. Notice that this instruction can in particular be used to perform sixteen parallel 4-to-8-bit table lookups: let us call T the table; take as first operand to `pshufb` the vector $x := (T[i],\ i = 0, \ldots, 15)$, as second operand the vector $y := (a, b, c, d, \ldots)$ on which to perform the lookup; then we see that the first byte of the result is $x[y[0]] = T[a]$, the second is $x[y[1]] = T[b]$, etc.

Finally, let us mention that there is a three-operand variant of this instruction in the more recent AVX instruction set and onward [Int14], which allows not to overwrite the first operand.

Targeted properties. In this chapter we focus solely on algorithms that can easily be implemented in a way that makes them immune to timing-attacks, see *e.g.* [TOS10]. Specifically, we consider the matrix as a known constant but the vector as a secret, and we wish to perform the multiplication without secret-dependent branches or memory accesses. It might not always be important to be immune, or even partially resistant to this type of attacks, but we consider that it should be important for any cryptographic

[1] It can actually also be used on 64-bit operands, but we do not consider this possibility here.

primitive or structure to *possibly* be implemented in such a way. Hence we try to find efficient such implementations for the SHARK structure and therefore for dense matrix-vector multiplications.

We now go on to describe the algorithms. In all of the remainder of this section, $\mathbf{x}$ and $\mathbf{y}$ are two column vectors of $\mathbf{F}_{2^4}^{16}$, and M a matrix of $\mathcal{M}_{16}(\mathbf{F}_{2^4})$. We first briefly recall the principle of table implementations which are generic and efficient but unsatisfactory when timing attacks are taken into account.

4.1 Table implementation

We wish to compute $\mathbf{y} = M\mathbf{x}$. The idea of table implementations is to use table lookups to perform the equivalent multiplication $\mathbf{y}^t = \mathbf{x}^t M^t$, *i.e.* $\mathbf{y}^t = \sum_{i=0}^{15} \mathbf{x}_i M_i^t$. This can be computed efficiently by tabulating beforehand the products $\lambda M_i^t, \lambda \in \mathbf{F}_{2^4}$, resulting in sixteen tables, each of sixteen entries of 64 bits, and then for each multiplication by accessing the table for M_i^t at the index $\mathbf{x}_i$ and summing all the retrieved table entries together. This only requires sixteen table lookups per multiplication. However, the memory accesses depend on the value of $\mathbf{x}$, which makes this algorithm inherently vulnerable to timing attacks.

Note that there is a more memory-efficient alternative implementation of this algorithm which consists in explicitly computing the multiplications for each term λM_i^t instead of using a table lookup. This can be done with a single `pshufb` instruction per matrix row, taking as first operand the table of multiplication by λ, which is made of sixteen 4-bit entries, and can be precomputed, and as second operand a register representing M_i^t. In that case, only the sixteen multiplication tables need to be stored, but their accesses still depend on the secret value $\mathbf{x}$, as one needs to look up the appropriate multiplication table for each $\mathbf{x}_i$.

4.2 A generic constant-time algorithm

We now describe our first algorithm, which can be seen as a variant of table multiplication that is immune to timing attacks. The idea consists again in computing the right multiplication $\mathbf{y}^t = \mathbf{x}^t M^t$, *i.e.* $\mathbf{y}^t = \sum_{i=0}^{15} \mathbf{x}_i M_i^t$. However, instead of tabulating the results of the scalar multiplication of the matrix rows M_i^t, those are always recomputed, in a way that does not explicitly depend on the value of the scalar.

4.2.1 Description of Algorithm 6.1

We start with a high-level description of the algorithm. Assume we want to perform the scalar multiplication $\lambda\mathbf{z}$ for an unknown scalar λ and a known, constant vector $\mathbf{z} \in \mathbf{F}_{2^4}^{16}$. Let us write λ as the polynomial $\lambda_3 \cdot \alpha^3 + \lambda_2 \cdot \alpha^2 + \lambda_1 \cdot \alpha + \lambda_0$ with the λ_i in $\mathbf{F}_2$. Then, the result of $\lambda \cdot \mathbf{z}$ is simply $\lambda_3 \cdot (\alpha^3 \cdot \mathbf{z}) + \lambda_2 \cdot (\alpha^2 \cdot \mathbf{z}) + \lambda_1 \cdot (\alpha \cdot \mathbf{z}) + \lambda_0 \cdot \mathbf{z}$. Thus we just need to precompute the products $\alpha^i \cdot \mathbf{z}$, select the right ones with respect to the binary representation of λ, and add these together. This can easily be achieved thanks to a *broadcast* function defined as:

$$broadcast(x,i)_n = \begin{cases} \mathbf{1}_n & \text{if the } i^{\text{th}} \text{ bit of } x \text{ is set} \\ \mathbf{0}_n & \text{otherwise} \end{cases},$$

The full algorithm then just consists in using this scalar-vector multiplication sixteen times, one for each row of the matrix.

This results in Algorithm 6.1. Note that a similar algorithm was used by Käsper and Schwabe for binary matrices of dimension 128 [KS09].

Algorithm 6.1: Broadcast-based matrix-vector multiplication

```
Input: $\mathbf{x} \in \mathbf{F}_{2^4}^{16}$, $M \in \mathcal{M}_{16}(\mathbf{F}_{2^4})$
Output: $\mathbf{y} = \mathbf{x}^t \cdot M^t$
Offline phase
for $i := 0,\ i < 16$ do
    $8M_i := \alpha^3 \cdot M_i$
    $4M_i := \alpha^2 \cdot M_i$
    $2M_i := \alpha \cdot M_i$
Online phase
$\mathbf{y} := \mathbf{0}_{64}$
for $i := 0,\ i < 16$ do
    $v_i^8 := 8M_i \wedge_{64} broadcast(\mathbf{x}_i, 3)_{64}$
    $v_i^4 := 4M_i \wedge_{64} broadcast(\mathbf{x}_i, 2)_{64}$
    $v_i^2 := 2M_i \wedge_{64} broadcast(\mathbf{x}_i, 1)_{64}$
    $v_i^1 := M_i \wedge_{64} broadcast(\mathbf{x}_i, 0)_{64}$
    $v_i := v_i^8 + v_i^4 + v_i^2 + v_i^1$
    $\mathbf{y} := \mathbf{y} + v_i$
return y
```

4.2.2 Implementation of Algorithm 6.1 with SSSE3 instructions

We now consider how to efficiently implement Algorithm 6.1 in practice. The only non-trivial operation that is used is the *broadcast* function, and we show that it can be performed with only one or two `pshufb` instructions.

To compute $broadcast(\lambda, i)_{64}$, with λ a 4-bit value, we can use a single `pshufb` with first operand x, such that $x[j] = 11111111_2$ if the i^{th} bit of j is set and 0 otherwise, and with second operand $y := (\lambda, \lambda, \ldots)$. The result of `pshufb` x y is indeed $(x[\lambda], x[\lambda], \ldots)$ which is $\mathbf{1}_{64}$ if the i^{th} bit of λ is set and $\mathbf{0}_{64}$ otherwise, that is $broadcast(\lambda, i)_{64}$. In fact, the quantities actually computed this way are rather $\mathbf{0}_{128}$ and $\mathbf{1}_{128}$. Adapting this to $\mathbf{0}_{64}$ and $\mathbf{1}_{64}$ is however simple to do.

In practice, the input x to `pshufb` can conveniently be constructed offline and stored in memory, but the second input y might not be readily available before performing this computation, and because it depends on what we assume to be a secret value, it cannot either be fetched from memory. However, it can easily be computed thanks to an additional `pshufb`. Alternatively, if the above computation is done with a vector $y := (\lambda, ?, ?, \ldots)$ instead, with ? denoting unknown values, and call z its result $(x[\lambda], ?, ?, \ldots)$, then we have $\mathit{broadcast}(\lambda, i)_n =$ `pshufb` z $(0, 0, \ldots)$.

In the specific case of matrices of dimension sixteen over $\mathbf{F}_{2^4}$, one can take advantage of the 128-bit wide `xmm` registers by interleaving, say, $8\mathbf{x}$ with $4\mathbf{x}$, and $2\mathbf{x}$ with $\mathbf{x}$, and by computing a slightly more complex version of the *broadcast* function $\mathit{broadcast}(x, i, j)_{2n}$ which interleaves $\mathit{broadcast}(x, i)_n$ with $\mathit{broadcast}(x, j)_n$. In that case, an implementation of one step of Algorithm 6.1 only requires two *broadcast* calls, two logical *and*, folding back the interleaved vectors (which only needs a couple of logical shift and exclusive or), and adding the folded vectors together. We give a snippet of such an implementation in Section C.1.

Scalar implementation of broadcast. On an architecture without access to `pshufb`, the *broadcast* function can still be implemented rather efficiently with a few arithmetic and logical instructions. Let x be an 64-bit value, w.l.o.g., represented by an unsigned integer, then the following sequence of C instructions computes $\mathit{broadcast}(x, i)_{64}$:

```c
x = ~x;
x >>= i;
x &= 1;
x -= 1ull;
```

In this snippet, after the third line, x is equal to 0 if its i^{th} bit was initially one, and 1 otherwise. The subtraction by one in line four thus produces the desired result.

4.3 A faster algorithm exploiting the matrix structure

Algorithm 6.1 is already reasonably efficient, and has the advantage of being completely generic w.r.t. the matrix. Yet, better solutions may exist in more specific cases. We present here an alternative that can be much faster when the matrix possesses a particular structure.

The idea behind this second algorithm is to take advantage of the fact that many coefficients of a matrix may be identical, thence many of the finite-field multiplications performed in a matrix-vector product consist in multiplications by the same constants. Thus, we may be tempted to take advantage of this fact by performing those in parallel. A motivation of this approach is that the multiplications being by constants, *i.e.* the known coefficients of the matrix, they can be performed by `pshufb`-implemented table lookups, which are more efficient than a broadcast-based algorithm.

4.3.1 Description of Algorithm 6.2

We first give a high-level idea of the algorithm. Let us first consider a small example, and compute $M\mathbf{x}$ defined as:

$$\begin{pmatrix}1 & 0 & 2 & 2\\ 3 & 1 & 2 & 3\\ 2 & 3 & 3 & 2\\ 0 & 2 & 3 & 1\end{pmatrix} \cdot \begin{pmatrix}x_0\\ x_1\\ x_2\\ x_3\end{pmatrix}. \tag{6.1}$$

It is obvious that this is equal to:

$$\begin{pmatrix}x_0\\ x_1\\ 0\\ x_3\end{pmatrix} + 2\cdot\begin{pmatrix}x_2\\ x_2\\ x_0\\ x_1\end{pmatrix} + 2\cdot\begin{pmatrix}x_3\\ 0\\ x_3\\ 0\end{pmatrix} + 3\cdot\begin{pmatrix}0\\ x_0\\ x_1\\ x_2\end{pmatrix} + 3\cdot\begin{pmatrix}0\\ x_3\\ x_2\\ 0\end{pmatrix},$$

where the constant multiplications of the vector $(x_0, x_1, x_2, x_3)^t$ and the shuffles of its coefficients can both be computed with a single `pshufb` instruction each, while none of these operations directly depends on the value of the vector. This type of decomposition can be done for any matrix, but the number of operations depends on the value of its coefficients.

We now sketch one way of obtaining an optimal decomposition as above. We consider a matrix product $M\mathbf{x}$ with M constant and $\mathbf{x}$ seen as the formal arrangement of indeterminates $\mathbf{x}_i$. Let us define $\mathcal{S}(M,\lambda)$ as one of the minimal sets of shuffles of coefficients of $\mathbf{x}$, such that there exists a unique vector $\mathbf{z} \in \mathcal{S}(M,\lambda)$ such that $\mathbf{z}_i = \mathbf{x}_j$ iff $M_{i,j} = \lambda$. For instance, in the above example, we have $\mathcal{S}(M,2) = \{(x_2, x_2, x_0, x_1)^t, (x_3, 0, x_3, 0)^t\}$. Equivalently, we could have taken $\mathcal{S}(M,2) = \{(x_3, x_2, x_3, 0)^t, (x_2, 0, x_0, x_1)^t\}$. These sets are straightforward to compute for this particular matrix, and so are they in the general case.

From the definition of $\mathcal{S}$, it is clear that we have $M\mathbf{x} = \sum_{\lambda\in\mathbf{F}_{2^4}^*} \sum_{\mathbf{z}\in\mathcal{S}(M,\lambda)} \lambda\cdot\mathbf{z}$. Once the values of the sets $\mathcal{S}$ have been determined, it is clear that we only need to compute this sum to get our result, and this is precisely what this second algorithm does.

We give a complete description of this shuffle-based process in Algorithm 6.2.

4.3.2 Cost of Algorithm 6.2

The cost of computing a matrix-vector product with Algorithm 6.2 depends on the coefficients of the matrix, since the size of the sets $\mathcal{S}(M,\lambda)$ depends both on the density of the matrix and of how its coefficients are arranged.

If we suppose that a vector implementation of this algorithm is used, and if the dimension and the field of the matrix are well-chosen, we can assume that both the scalar multiplication of $\mathbf{x}$ by a constant and a shuffle can be computed with a single `pshufb` each, and a few ancillary instructions. We take the number of calls to `pshufb`

Algorithm 6.2: Shuffle-based matrix-vector multiplication

```
Input: $\mathbf{x} \in \mathbf{F}_{2^4}^{16}$, $M \in \mathcal{M}_{16}(\mathbf{F}_{2^4})$
Output: $\mathbf{y} = M \cdot \mathbf{x}$
Offline phase
for $i := 1,\ i < 16$ do
    $s_i[\,] := \mathcal{S}(M, i)$  ▷ Initialise the array $s_i$ with one of the possible sets of shuffles.
Online phase
$\mathbf{y} := \mathbf{0}_{64}$
for $i := 1,\ i < 16$ do
    for $j := 1,\ j < \#s_i$ do
        $\mathbf{y} := \mathbf{y} + i \cdot s_i[j]$
return $\mathbf{y}$
```

as a basis to define a cost function for matrices with respect to their implementations with Algorithm 6.2. It is defined as:

$$cost2(M) := \sum_{\lambda \in \mathbf{F}_{2^4}^*} \#\mathcal{S}(M, \lambda) + \sum_{\lambda \in \mathbf{F}_{2^4}^* \setminus \{1\}} \mathbb{I}(\mathcal{S}(M, \lambda)).$$

The first term corresponds to the number of shuffles that need to be performed for each constant value, and the second to the number of constant multiplications to do. We may notice that $\#\mathcal{S}(M, \lambda)$ is equal to the maximum number of occurrence of λ in a single row of M, and the *cost2* function is therefore easy to compute. As an example the cost of the matrix M from Equation 6.1 is 7. In the special case of a matrix that has a diagonal with identical non-zero coefficients, the actual cost is one less than computed with *cost2*.

A matrix with minimal cost w.r.t. *cost2* is one that minimises the sum of the maximum number of occurrence of λ in any row, for every $\lambda \in \mathbf{F}_{2^4}^*$. A simple observation is that for matrices with the same number of non-zero coefficients, this amount is the smallest when every row can be deduced by permutation of a single one; this is for instance the case for circulant matrices, by definition. More generally, we can heuristically hope that the cost of a matrix will be low if all of its rows can be obtained from the permutation of, say, only two rows.

We can try to estimate the minimum cost of an arbitrary dense circulant matrix of dimension sixteen over $\mathbf{F}_{2^4}$. If the coefficients of the unique row, up to permutation, are chosen uniformly at random, we should expect them to be made of about $1-(1-1/16)^{16} \approx 2/3$ of the elements of $\mathbf{F}_{2^4}$. Additionally, fifteen permutations of the input will need to be computed: all rows need to be different for the matrix to be of full rank —which is something we will desire in our applications— but the coefficients on the diagonal are constant. Thus we can estimate the cost of such a random matrix to be about 25.

Before concluding this section, let us notice that special cases of this algorithm have already been used for circulant matrices, namely in the case of the AES MixColumn matrix [Ham09].

4.3.3 Implementation of Algorithm 6.2 with SSSE3 instructions

The implementation of Algorithm 6.2 is quite straightforward. We give nonetheless a small code snippet in Section C.2.

4.4 Performance

In Table 6.2 and Table 6.3 of Section 6, we give a few performance figures for ciphers with a SHARK structure using assembly implementations of Algorithm 6.1 and Algorithm 6.2 for their linear mapping. From there it can be seen without surprise that Algorithm 6.2 is more efficient if the matrix is well-chosen. However, Algorithm 6.1 still performs reasonably well, without imposing any condition on the matrix.

5 Diffusion matrices from algebraic geometry codes

We now discuss how to design diffusion matrices based on the code from Example 6.4 of Section 3 with efficient implementations with respect to Algorithm 6.2 of the previous section.

A diffusion matrix associated to this code is simply the right block A of a generator matrix in systematic form $G \coloneqq (I_{16}\ A)$. Recall that G is the result of computing the reduced row-echelon form of the matrix obtained by evaluating a basis for the function space $\mathcal{F}$ on the evaluation domain $\mathcal{P}$. Using the row-echelon form is simply a convenient way of computing the basis that results in G being in systematic form, so it is clear that the initial choice of the basis for $\mathcal{F}$ prior reduction does not impact G, and hence A. However, changing the order of the elements of the evaluation domain does result in different matrices.

The problem for the rest of the section is thus to find good orderings $\mathcal{P}$, that result in matrices with a low cost, *i.e.* efficient *encoders*, that is implementations of a generator matrix. For simplicity, we write $\mathcal{C}_2$ to denote any code obtained from Example 6.4, with the precise choice of the evaluation domain $\mathcal{P}$ left unspecified.

5.1 Compact encoders using code automorphisms

Our first objective is to find matrices obtained from $\mathcal{C}_2$ that can be generated by permutations of a small number of rows. The main tool we use to achieve this goal are *automorphisms* of $\mathcal{C}_2$; these are the injective morphisms from the code to itself.

Definition 6.22 (Automorphism of a code)**.** The automorphism group $\mathrm{Aut}(\mathcal{C})$ of a code $\mathcal{C}$ of length n is a subgroup of $\mathfrak{S}_n$ such that $\pi \in \mathrm{Aut}(\mathcal{C}) \Leftrightarrow (c \in \mathcal{C} \Rightarrow \pi(c) \in \mathcal{C})$.

As $\mathcal{C}_2$ is an evaluation code, we can equivalently define its automorphisms as being permutations of the elements of $\mathcal{P}$. If π is an automorphism of $\mathcal{C}_2$, if $\{O_0, \ldots, O_\ell\}$ are its orbits, and if our instance of the code has an evaluation support s.t. elements of a same orbit are consecutive, then the action of π on a codeword of $\mathcal{C}_2$ is to cyclically permute its coordinates locally, along each orbit.

To see that this can be useful, assume that there is an automorphism π with two orbits O_0 and O_1 of size $n/2$ each. Then, if $M := (I_{n/2}\ A)$ is built from $\mathcal{P} := (O_0, O_1)$, each row of M can be obtained by the repeated action of π on, say, M_0, and it follows that A is circulant and therefore has a low cost w.r.t. Algorithm 6.2. More generally, if an automorphism can be found such that it has orbits of sizes summing up to $n/2$, and if the two partitions defined by the orbits have maximal rank, then the corresponding matrix M can be deduced from a small set of rows. We give two toy examples with Reed-Solomon codes, which can easily be verified.

$\pi : \mathbf{F}_{2^4} \to \mathbf{F}_{2^4},\ x \mapsto 8x$. This automorphism has $O_0 := (1, 8, 12, 10, 15)$ and $O_1 := (2, 3, 11, 7, 13)$ for orbits, among others. The systematic matrix for the $[10, 5, 6]_{\mathbf{F}_{2^4}}$ code obtained with the points in that order is then such that A is circulant and obtained from the cyclic permutation of the row $(12, 10, 2, 6, 3)$.

$\pi : \mathbf{F}_{2^4} \to \mathbf{F}_{2^4},\ x \mapsto 7x$. This automorphism has $O_0 := (1, 7, 6)$, $O_1 := (2, 14, 12)$, $O_2 := (4, 15, 11)$, and $O_3 := (8, 13, 5)$ for orbits, among others. The systematic matrix for the $[12, 6, 7]_{\mathbf{F}_{2^4}}$ code obtained with the points in that order is then of the form $\begin{pmatrix} I_3 & 0_3 & A & B \\ 0_3 & I_3 & C & D \end{pmatrix}$ with A, B, C and D circulant matrices. It can thus be obtained by cyclic permutation of only two rows.

5.1.1 Application to $\mathcal{C}_2$

Automorphisms of $\mathcal{C}_2$ may be quite harder to find than ones of RS codes. They can however be found within automorphisms of the curve $\mathcal{X}$ on which it is based [Sti09], or rather its function field. This is quite intuitive, as these will precisely permute places of the curve, which are the elements on which the code is defined. We mostly need to be careful to only use automorphisms that fix the places of the divisor D used to define $\mathcal{F}$. In our case, this means fixing the place at infinity.

We considered the degree-one automorphisms of the curve $\mathcal{X}$ of Example 6.4, for instance described by Duursma [Duu99]. They have two generators: $\pi_0 : \mathbf{F}^2_{2^4} \to \mathbf{F}^2_{2^4},\ (x, y) \mapsto (\zeta x, y)$ with $\zeta^5 = 1$, and $\pi_{1_{(a,b)}} : \mathbf{F}^2_{2^4} \to \mathbf{F}^2_{2^4},\ (x, y) \mapsto (x + a, y + a^8 x^2 + a^4 x + b^4)$, with (a, b) an affine point of $\mathcal{X}$. These generators span a group of order 160. When considering their orbit decomposition, the break-up of the size of the orbits can only be of one of five types, given in Table 6.1.

From these automorphisms, it is possible to define partitions of $\mathcal{P}$ in two sets of size sixteen which are union of orbits. We may therefore hope to obtain systematic matrices of the type we are looking for. Unfortunately, after an extensive search both on $\mathcal{C}_2$ and

Table 6.1 – Possible combination of orbit sizes of automorphisms of $\mathcal{C}_2$ spanned by π_0 and π_1. A number n in column c means that an automorphism of this type has n orbits of size c.

Orbit size	1	2	4	5	10
Type 1	32	0	0	0	0
Type 2	0	16	0	0	0
Type 3	0	0	8	0	0
Type 4	2	0	0	6	0
Type 5	0	1	0	0	3

on the smaller elliptic code of Example 6.3 using its own automorphisms, it appears that ordering $\mathcal{P}$ in this fashion *never* results in obtaining a systematic matrix, *i.e.* computing the row-reduced of the initial matrix never results in a left square submatrix of full rank.

5.1.2 Extending the automorphisms with the Frobenius mapping

We extend the previous automorphisms with the Frobenius mapping $F : \mathbf{F}_{2^4}^2 \to \mathbf{F}_{2^4}^2$, $(x, y) \mapsto (x^2, y^2)$; this adds another 160 automorphisms for $\mathcal{X}$. However, these will not anymore be automorphisms for the *code* $\mathcal{C}_2$ in general, and we will therefore obtain matrices of a form slightly different from what we first hoped to achieve.

The global strategy is still the same, however, and consists in ordering the points along orbits of the curve automorphisms. By using the Frobenius, we can obtain new combination of orbit sizes, notably four of size eight. We study below the result of ordering $\mathcal{P}$ along the orbits of one such automorphism. We take the example of $\sigma := F \circ \sigma_2 \circ \sigma_1$, with $\sigma_1 : (x, y) \mapsto (x + 1, y + x^2 + x + 7)$ and $\sigma_2 : (x, y) \mapsto (12x, y)$. An observation is that in this case, only σ^0 and σ^4 are automorphisms of $\mathcal{C}_2$. Note that not all orbits orderings of σ for $\mathcal{P}$ yield a systematic matrix. However, unlike as above, we were able to find some orders that do. In these cases, the right matrix "A" of the full generator matrix $(I_{16}\ A)$ is of the form:

$$\begin{gathered}(A_0, A_1, A_2, A_3, \sigma^4(A_0), \sigma^4(A_1), \sigma^4(A_2), \sigma^4(A_3),\\ A_8, A_9, A_{10}, A_{11}, \sigma^4(A_8), \sigma^4(A_9), \sigma^4(A_{10}), \sigma^4(A_{11})),\end{gathered}$$

with $A_0, \ldots, A_3$, $A_8, \ldots, A_{11}$ row vectors of dimension 16. For instance, the first and fifth row of one such matrix are:

$$\begin{aligned} A_0 &= (5, 2, 1, 3, \mathbf{8}, \mathbf{5}, \mathbf{1}, \mathbf{5}, 12, 10, 14, 6, \mathbf{7}, \mathbf{11}, \mathbf{4}, \mathbf{11})\\ A_4 = \sigma^4(A_0) &= (\mathbf{8}, \mathbf{5}, \mathbf{1}, \mathbf{5}, 5, 2, 1, 3, \mathbf{7}, \mathbf{11}, \mathbf{4}, \mathbf{11}, 12, 10, 14, 6).\end{aligned}$$

We give the full matrix corresponding to these rows in Figure 6.2.

We have therefore partially reached our goal of being able to describe A from a permutation of a subset of its rows. However this subset is not small, as it is of size

8 —half of the matrix dimension. Consequently, these matrices have a moderate cost according to the *cost2* function, when implemented with Algorithm 6.2, but it is not minimal. Interestingly, all the matrices of this form that we found have the same cost of 52.

5.2 Fast random encoders

We conclude this section by presenting the results of a very simple random search for efficient encoders of $\mathcal{C}_2$ with respect to Algorithm 6.2. Unlike in the above discussion, this search does not exploit any kind of algebraic structure. Indeed, it only consists in repeatedly generating a random permutation of the affine points of the curve, building a matrix for the code with the corresponding point order, tentatively putting it in systematic form $(I_{16}\ A)$, and if successful evaluating the *cost2* function from Section 4.3 on A. We then collect matrices with a minimum cost.

Because there are $32! \approx 2^{117.7}$ possible point orders, we can only explore a very small part of the search space. However, matrices of low cost can be found even after a moderate amount of computation, and we found many matrices of cost 43, though none of a lower cost. We present in Table 6.4 from Section B the number of matrices of cost strictly less than 60 that we found during a search of 2^{38} encoders. We give an example of a matrix of cost 43 in Figure 6.3, which is only about a factor 1.7 away from the estimate of the minimum cost of a circulant matrix given in Section 4.3. We observe that the transpose of this matrix, *i.e.* its inverse, also has a cost of 43.

6 Applications and performance

This last section presents the performance of straightforward assembly implementations of both of our algorithms when applied to a fast encoder of the code $\mathcal{C}_2$ from Section 5. In all of the remainder, $\mathcal{M}$ denotes the matrix from Figure 6.3; it is of dimension 16 over $\mathbf{F}_{2^4}$ and has a differential and linear branch number of 15. We use this matrix in the design of the Samneric couple of toy block ciphers; Sam is a 64-bit block cipher that uses $\mathcal{M}$ *as is* with a four-bit S-box, and Eric uses $\mathcal{M}$ over $\mathbf{F}_{2^8}$ together with an eight-bit S-box to define a 128-bit block cipher.

We do not actually specify complete block ciphers, in particular we do not give key schedules; our objective is rather to evaluate the performance of round functions following these structures, in order to assess the utility of large diffusion mappings for block cipher design. Consequently, our security analysis of Samneric is minimal, and only serves to give a realistic estimate of how many rounds are needed to achieve good security.

6.1 The Sam block cipher

The round function of Sam is taken to be $\mathcal{M} \circ \mathcal{S} \circ \mathcal{K}$, where $\mathcal{K}$ adds a 64-bit round key to its input and $\mathcal{S}$ is the parallel application of sixteen identical 4-bit S-boxes, taken to be any optimal S-box of this size, see *e.g.* [Saa11, LP07].

Table 6.2 – Performance of software implementations of Sam, given in cycles per byte (cpb), for implementations using Algorithm 6.1 and Algorithm 6.2. Figures in parentheses are for an AVX implementation when applicable.

Processor type	# rounds	cpb (Alg. 6.1)	cpb (Alg. 6.2)
Intel Xeon E5-2650 @ 2.00GHz	8	66.5 (60.2)	44.5 (31.9)
Intel Xeon E5-2609 @ 2.40GHz	8	95.3 (84.7)	63.3 (45.6)
Intel Xeon E5649 @ 2.53GHz	8	111.3	59.8

We use standard *wide-trail* considerations to study differential and linear properties of a Sam instance [DR02]. This is very easy to do thanks to the simple structure of the cipher. The branch number of $\mathcal{M}$ is 15, which means that at least 15 S-boxes are active in any two rounds of a differential or linear trail, while the best 4-bit S-boxes have a maximum differential probability of 2^{-2} and a squared correlation of at most 2^{-2} for any linear approximation; we will discuss such notions in more details in the next Chapter 7. By using such S-boxes, one can upper-bound the expected probability of a single differential trail over the choice of the key (respectively the squared correlation of a single fixed-key linear trail) for $2n$ rounds by $2^{-2 \cdot 15 n}$. This is smaller than 2^{-64} as soon as $n > 2$. Hence we conjecture that 6 to 8 rounds are enough to make a cipher resistant to standard statistical attacks.

The most powerful attacks against Sam may in fact not be statistical; algebraic attacks may be more efficient in exploiting the fact that the round function is defined at a granularity of four bits, and that the degree of an invertible 4-bit S-box is at most three. Applying the upper-bound from Boura *et al.* [BCD11, Theorem 2], we get that seven rounds are necessary to obtain a permutation of maximal degree. Thus, eight full rounds plus an additional half-round consisting of only $\mathcal{S}$ would probably be necessary to avoid attacks based on the low degree of the cipher.

On the other hand, we would expect Sam to be more resistant than most ciphers to structural attacks such as impossible-differential or integral attacks, as its round function diffuses over the full block.

We give software performance figures for an 8-round version of Sam in Table 6.2. We include data both for a strict SSSE3 implementation and for one using AVX extensions, which can be seen to bring a considerable benefit. Note that the last round is complete and includes the linear mapping, unlike *e.g.* AES. As the parallel application of the S-Boxes of $\mathcal{S}$ can be implemented very efficiently with a single `pshufb`, this part of the round function has virtually no impact on the performance, and one could for instance add an extra half-round essentially for free.

6.2 The Eric block cipher

Although the code $\mathcal{C}_2$ from which the matrix $\mathcal{M}$ is built was initially defined with $\mathbf{F}_{2^4}$ as an alphabet, this latter can be replaced by an algebraic extension of $\mathbf{F}_{2^4}$ such as $\mathbf{F}_{2^8}$, to yield a code $\mathcal{C}_2{}'$ with the same parameters, namely a $[32, 16, 15]_{\mathbf{F}_{2^8}}$ code. One way of seeing this is that by constructing $\mathbf{F}_{2^8}$ as an extension of $\mathbf{F}_{2^4}$, for instance by letting $\mathbf{F}_{2^4} \cong \mathbf{F}_2[\alpha]/\langle \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1\rangle$ and $\mathbf{F}_{2^8} \cong \mathbf{F}_{2^4}[t]/\langle t^2 + t + \alpha\rangle$; an element of $\mathbf{F}_{2^8}$ is represented as a degree-one polynomial $at + b$ over $\mathbf{F}_{2^4}$. It follows that the minimum weight of a codeword $w := (a_i t + b_i)$, $i \in \{0 \ldots 31\}$ of $\mathcal{C}_2{}'$ is at least equal to the minimum weight of words (a_i), $i = 0 \ldots 31$ and (b_i), $i \in \{0 \ldots 31\}$. If those are taken among codewords of $\mathcal{C}_2$, their minimum weight is 15, and thus so is the one of w.

An efficient encoder can also be built for $\mathcal{C}_2{}'$, with only small changes from an encoder for $\mathcal{C}_2$. If we consider finite field multiplications as performed by linear feedback shift registers (LFSRs), the multiplication by a constant is done by summing shifts of the multiplicand and possibly shifted copies of the feedback polynomial. Such a multiplication, say on eight bits can obviously be tabulated in a table of 256 8-bit entries, but smaller tables can also be used, exploiting the linearity. For instance, one can compute the multiplication from two tables of 16 8-bit entries which store the partial multiplication of the constant by the four most and least significant bits respectively; performing the complete multiplication simply consists in accessing these tables with the right inputs and summing their results. We give an example of such an approach in Figure 6.1.

```
uint8_t m71(uint8_t x)
{
   uint8_t lo[16] = {0x00,0x71,0xE2,0x93,0xDF,0xAE,0x3D,0x4C
                    ,0xA5,0xD4,0x47,0x36,0x7A,0x0B,0x98,0xE9};
   uint8_t hi[16] = {0x00,0x51,0xA2,0xF3,0x5F,0x0E,0xFD,0xAC
                    ,0xBE,0xEF,0x1C,0x4D,0xE1,0xB0,0x43,0x12};

   return (lo[x & 0xF] ^ hi[x >> 4]);
}
```

Figure 6.1 – Multiplication by `0x71` in $\mathbf{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$

Table lookups from four to eight bits can be performed efficiently with `pshufb`; multiplication by a constant in $\mathbf{F}_{2^8}$ is thus twice as expensive as multiplication in $\mathbf{F}_{2^4}$, but it operates on twice the amount of data. An implementation of a diffusion matrix $\mathcal{M}'$ from $\mathcal{C}_2{}'$ is thus comparatively as efficient as for the original code.

The round function of Eric is taken to be $\mathcal{M}' \circ \mathcal{S}' \circ \mathcal{K}'$, where $\mathcal{K}'$ adds a 128-bit round key to its input and $\mathcal{S}'$ is the parallel application of sixteen identical cryptographically strong 8-bit S-boxes.

From wide-trail considerations, the resistance of an Eric cipher to statistical attacks is comparatively better than Sam, when an appropriate 8-bit S-Box is used. For in-

Table 6.3 – Performance of software implementations of Eric, in cycles per byte (cpb), for implementations using Algorithm 6.1 and Algorithm 6.2. Figures in parentheses are for an AVX implementation when applicable.

		128-bit Block	
Processor type	# rounds	cpb (Alg. 6.1)	cpb (Alg. 6.2)
Intel Xeon E5-2650 @ 2.00GHz	6	58 (52.3)	32.7 (26.5)
	8	76.8 (69.6)	43.8 (35.7)
Intel Xeon E5-2609 @ 2.40GHz	6	79.8 (75.6)	47.1 (36.8)
	8	106.6 (97.1)	62.1 (50.3)
Intel Xeon E5649 @ 2.53GHz	6	84.5	47
	8	111	61.9

stance, the AES S-box has a maximal differential probability of 2^{-6}, and it cannot be approximated linearly with a squared correlation more than 2^{-6} [DR02]. This implies that the expected probability (respectively squared correlation) of a single differential trail (respectively linear trail) for $2n$ rounds is upper-bounded by $2^{-6 \cdot 15n}$, which is already much smaller than 2^{-128} as soon as $n > 1$. Eric is also likely to be more resistant to algebraic attacks; at least five rounds are necessary to reach full degree for an S-box of degree seven, which is two less than Sam. Thus, eight and a half rounds of an Eric cipher should bring adequate security, and six and a half rounds might be enough. We provide performance figures for SSSE3 and an AVX implementations in Table 6.3, for both six and eight full rounds. Note that in the case of 8-bit S-Boxes, the S-Box application is usually rather more complex and expensive a step than with 4-bit S-Boxes. In these test programs, we used the efficient vector implementation of the AES S-Box from Hamburg [Ham09]; better performance may be attained by using cryptographically weaker but more efficient S-boxes, such as Whirlpool's second S-box [BR03].

6.3 Discussion

The performance figures given in Table 6.2, Table 6.3 are average for a block cipher. For instance, it compares favourably with the optimized vector implementations of 64-bit ciphers LED and Piccolo in sequential mode from [BGLP13], which run at speeds between 70 and 90 cpb., depending on the CPU. It is however slower than Hamburg's vector implementation of AES, with reported speeds of 6 to 22 cpb. (9 to 25 for the inverse cipher) [Ham09, SMMK12].

Yet, we conjecture that the strong structure of the round function makes Samneric more suitable to speed-ups from derivative constructions. For instance, the *leak extraction* approach used in the stream cipher LEX allowed a 2.5-time speed-up from the AES by leaking a quarter of the state of the cipher as keystream every round [Bir06]. Although

LEX was later broken [DK08], we believe that the stronger diffusion in Samneric may protect leak extraction variants against similar attacks; these could for instance lead to two-time speed-ups for an eight-round Eric with leaks of the same amount as the ones of LEX.

Finally, a particular advantage of large mappings over small alphabets is that they allow to build ciphers with a small AND complexity, in terms of the number and depth of AND gates necessary to implement it. This may be useful in certain contexts such as masking at high order [GGNS13] and fully-homomorphic encryption [ARS+15].

7 Conclusion

We revisited the SHARK structure by replacing the MDS matrix of its linear diffusion layer by matrices built from an algebraic geometry code. Although this code is not MDS, it still has a very high minimum distance, all the while being quite long. This allowed us to define an efficient full-state diffusion layer for a 64-bit block cipher, operating on 4-bit values.

We studied algorithms suitable for a vector implementation of the multiplication by this matrix, and how to find matrices that are most efficiently implemented with those algorithms. Finally, we gave performance figures for assembly implementations of hypothetical SHARK-like ciphers using this matrix as a linear layer.

This work provided the first generalisation of SHARK that are not vulnerable to timing attacks as is the original cipher, and also the first generalisation to 128-bit blocks. It also showed that even if not the fastest, such potential design could be implemented efficiently in software.

As a future work, it would be interesting to investigate how to use the full automorphism group of the code to design matrices with a lower cost. In that case, we would not restrict ourselves to derive the rows from a single row and the powers of a *single* automorphism, but could use several independent automorphisms instead.

A Examples of diffusion matrices of dimension 16 over $\mathbf{F}_{2^4}$

A.1 A matrix of cost 52

The $\mathcal{C}_2$ matrix of Figure 6.2 was found thanks to the automorphisms described in Section 5.1. It has a cost 52 w.r.t. Section 4.3, and both a differential and a linear branch number of 15. It is a block matrix made of sixteen square matrices of dimension four, eight of them being distinct. The different blocks are highlighted by different colours in the figure.

A.2 A matrix of cost 43

The $\mathcal{C}_2$ matrix of Figure 6.3 was found by randomly testing permutations of the points of the curve. It has a cost 43, and so does its transpose.

$$\begin{pmatrix}
5 & 2 & 1 & 3 & 8 & 5 & 1 & 5 & 12 & 10 & 14 & 6 & 7 & 11 & 4 & 11 \\
2 & 2 & 4 & 1 & 5 & 12 & 2 & 1 & 9 & 15 & 8 & 11 & 7 & 6 & 9 & 3 \\
1 & 4 & 4 & 3 & 1 & 2 & 15 & 4 & 5 & 13 & 10 & 12 & 9 & 6 & 7 & 13 \\
3 & 1 & 3 & 3 & 5 & 1 & 4 & 10 & 14 & 2 & 14 & 8 & 15 & 13 & 7 & 6 \\
8 & 5 & 1 & 5 & 5 & 2 & 1 & 3 & 7 & 11 & 4 & 11 & 12 & 10 & 14 & 6 \\
5 & 12 & 2 & 1 & 2 & 2 & 4 & 1 & 7 & 6 & 9 & 3 & 9 & 15 & 8 & 11 \\
1 & 2 & 15 & 4 & 1 & 4 & 4 & 3 & 9 & 6 & 7 & 13 & 5 & 13 & 10 & 12 \\
5 & 1 & 4 & 10 & 3 & 1 & 3 & 3 & 15 & 13 & 7 & 6 & 14 & 2 & 14 & 8 \\
12 & 9 & 5 & 14 & 7 & 7 & 9 & 15 & 7 & 6 & 11 & 3 & 15 & 5 & 13 & 7 \\
10 & 15 & 13 & 2 & 11 & 6 & 6 & 13 & 6 & 6 & 7 & 9 & 5 & 10 & 2 & 14 \\
14 & 8 & 10 & 14 & 4 & 9 & 7 & 7 & 11 & 7 & 7 & 6 & 13 & 2 & 8 & 4 \\
6 & 11 & 12 & 8 & 11 & 3 & 13 & 6 & 3 & 9 & 6 & 6 & 7 & 14 & 4 & 12 \\
7 & 7 & 9 & 15 & 12 & 9 & 5 & 14 & 15 & 5 & 13 & 7 & 7 & 6 & 11 & 3 \\
11 & 6 & 6 & 13 & 10 & 15 & 13 & 2 & 5 & 10 & 2 & 14 & 6 & 6 & 7 & 9 \\
4 & 9 & 7 & 7 & 14 & 8 & 10 & 14 & 13 & 2 & 8 & 4 & 11 & 7 & 7 & 6 \\
11 & 3 & 13 & 6 & 6 & 11 & 12 & 8 & 7 & 14 & 4 & 12 & 3 & 9 & 6 & 6
\end{pmatrix}$$

Figure 6.2 – A diffusion block matrix.

It can easily be obtained as follows. First, define a basis of $\mathcal{L}(17P_\infty)$, for instance $(1, x, x^2, y, x^3, xy, x^4, x^2y, x^5, x^3y, x^6, x^4y, x^7, x^5y, x^8, x^6y)$. Then, arrange the affine points of the curve on which $\mathcal{C}_2$ is defined in the order ((8, 7), (13, 2), (4, 5), (0, 0), (14, 5), (15, 6), (7, 3), (1, 7), (2, 2), (11, 3), (3, 3), (10, 7), (6, 4), (5, 5), (9, 5), (12, 6), (3, 2), (11, 2), (12, 7), (13, 3), (4, 4), (0, 1), (1, 6), (7, 2), (9, 4), (6, 5), (2, 3), (5, 4), (15, 7), (10, 6), (14, 4), (8, 6)). Finally, evaluate the basis of $\mathcal{L}(17P_\infty)$ on these points and compute the reduced row echelon form of this matrix; the right square matrix of dimension 16 of the result is the matrix of Figure 6.3.

$$\begin{pmatrix}
11 & 6 & 1 & 6 & 10 & 14 & 10 & 9 & 13 & 3 & 3 & 12 & 9 & 15 & 2 & 9 \\
6 & 12 & 0 & 4 & 2 & 8 & 9 & 2 & 5 & 11 & 9 & 5 & 4 & 1 & 15 & 6 \\
9 & 11 & 2 & 2 & 1 & 11 & 13 & 15 & 13 & 3 & 2 & 1 & 14 & 1 & 3 & 10 \\
0 & 0 & 9 & 8 & 11 & 6 & 2 & 1 & 11 & 10 & 15 & 10 & 10 & 15 & 1 & 14 \\
13 & 13 & 3 & 15 & 3 & 1 & 11 & 2 & 9 & 2 & 10 & 14 & 1 & 11 & 1 & 2 \\
1 & 9 & 8 & 4 & 14 & 10 & 2 & 5 & 15 & 2 & 12 & 12 & 9 & 10 & 1 & 9 \\
5 & 9 & 11 & 2 & 15 & 1 & 12 & 4 & 6 & 0 & 6 & 4 & 5 & 8 & 2 & 9 \\
1 & 4 & 14 & 9 & 13 & 2 & 10 & 12 & 0 & 6 & 6 & 9 & 2 & 0 & 11 & 10 \\
13 & 10 & 3 & 9 & 2 & 15 & 6 & 6 & 11 & 1 & 9 & 9 & 12 & 14 & 10 & 3 \\
0 & 10 & 6 & 12 & 11 & 0 & 4 & 9 & 1 & 14 & 10 & 2 & 9 & 2 & 13 & 6 \\
2 & 0 & 5 & 6 & 9 & 0 & 1 & 5 & 15 & 12 & 13 & 15 & 1 & 11 & 13 & 11 \\
11 & 2 & 10 & 1 & 1 & 15 & 0 & 8 & 0 & 9 & 14 & 10 & 10 & 6 & 11 & 15 \\
12 & 14 & 10 & 11 & 3 & 10 & 6 & 0 & 5 & 11 & 1 & 8 & 2 & 9 & 2 & 3 \\
15 & 2 & 2 & 5 & 1 & 10 & 9 & 4 & 1 & 8 & 9 & 9 & 12 & 10 & 14 & 12 \\
15 & 1 & 12 & 5 & 13 & 11 & 0 & 6 & 2 & 5 & 11 & 1 & 15 & 0 & 9 & 13 \\
5 & 6 & 11 & 0 & 2 & 9 & 14 & 11 & 12 & 10 & 3 & 2 & 8 & 10 & 3 & 1
\end{pmatrix}$$

Figure 6.3 – An unstructured diffusion matrix

B Statistical distribution of the cost of matrices of $\mathcal{C}_2$

Table 6.4 gives the repartition of random matrices by their cost w.r.t. Section 4.3.

Table 6.4 – Statistical distribution of the cost of 2^{38} randomly-generated generator matrices of $\mathcal{C}_2$.

cost	#matrices	cumulative #matrices	cumulative proportion of the search space
43	146 482	146 482	0.00000053
44	73 220	219 702	0.00000080
45	218 542	438 244	0.0000016
46	879 557	1 317 801	0.0000048
47	1 978 159	3 295 960	0.000012
48	5 559 814	8 855 774	0.000032
49	21 512 707	30 368 481	0.00011
50	93 289 020	123 657 501	0.00045
51	356 848 829	480 506 330	0.0017
52	1 282 233 658	1 762 739 988	0.0064
53	3 534 412 567	5 297 152 555	0.019
54	8 141 274 412	13 438 426 967	0.049
55	15 433 896 914	28 872 323 881	0.11
56	24 837 735 898	53 710 059 779	0.20
57	33 794 051 687	87 504 111 466	0.32
58	38 971 338 149	126 475 449 615	0.46
59	38 629 339 524	165 104 789 139	0.60

C Excerpts of assembly implementations of matrix-vector multiplication

We give two snippets of assembly implementation of encoders for $\mathcal{C}_2$ using the two algorithms of Section 4. They allow both to make the description of the algorithms more explicit and to illustrate their respective complexities in terms of number of instructions.

C.1 Excerpt of an implementation of Algorithm 6.1

```
; cleaning mask
cle: dq 0x0f0f0f0f0f0f0f0f, 0x0f0f0f0f0f0f0f0f
; mask for the selection of v, 2v
oe1: dq 0xff0ff000ff0ff000, 0xff0ff000ff0ff000
; mask for the selection of 4v, 8v
oe2: dq 0xf0f0f0f000000000, 0xfffffff0f0f0f0f
; m0 and 2m0 interleaved
c01: dq 0x91a7efa76c126cb5, 0x9124fd91cb3636d9
; 4m0 and 8m0 interleaved
```

```
c02: dq 0x24efd9efb548b5a7, 0x248391245acbcb12
; etc.
c11: dq 0x249183244800cb6c, 0x6cfd12485a91b55a
c12: dq 0x83246c8336005ab5, 0xb59148367e24a77e
c21: dq 0xfdd9b5122424b591, 0xa73612ef122436d9
c22: dq 0x9112a7488383a724, 0xefcb48d94883cb12
; [...]

; macro for one matrix row multiplication and accumulation
; (nasm syntax)
; 1 is input
; 2, 3, 4, 5 are storage
; 6 is constant zero
; 7 is accumulator
; 8 is index
%macro m_1_row 8
  ; selects the right double-masks
  movdqa %2, [oe1]
  movdqa %3, [oe2]
  pshufb %2, %1
  pshufb %3, %1
  ; shift the input for the next round
  psrldq %1, 1
  ; expand
  pshufb %2, %6
  pshufb %3, %6
  ; select the rows with the double-masks
  pand   %2, [c01 + %8*16*2]
  pand   %3, [c02 + %8*16*2]

  ; shift and xor the rows together
  movdqa %4, %2
  movdqa %5, %3
  psrlq  %2, 4
  psrlq  %3, 4
  pxor   %4, %5
  pxor   %2, %3
  ; accumulate everything
  pxor   %2, %4
  pxor   %7, %2
%endmacro

; the input is in xmm0, with only the four lsb of each byte set
```

```
_m64:
  ; constant zero
  pxor xmm5, xmm5
  ; accumulator
  pxor xmm6, xmm6
.mainstuff:
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 0
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 1
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 2
  ; [...]
.fin:
  ; the result is in xmm0, and still of the same form
  pand xmm6, [cle]
  movdqa xmm0, xmm6
  ret
```

Figure 6.4 – Part of an encoder for $\mathcal{C}_2$ using Algorithm 6.1

C.2 Excerpt of an implementation of Algorithm 6.2

```
; multiplication tables (in the order where we need them)
ttim2 : dq 0x0e0c0a0806040200, 0x0d0f090b05070103
ttim8 : dq 0x0d050e060b030800, 0x0109020a070f040c
; [...]

; shuffles
; 0xff for not selected nibbles (it will zero them)
pp10: dq 0x0005000507040d02, 0x0f01040a03060809
pp11: dq 0xffff0e0c0e0bffff, 0xff0b08ff040cffff
pp12: dq 0xffffff0eff0dffff, 0xffffffffffffffff
pp20: dq 0x050306070602040e, 0x0408010c01000b04
pp21: dq 0x0c0e0909ff0307ff, 0x0bff020effff0dff
pp22: dq 0xffffff0fff0affff, 0xffffffffffffffff
pp30: dq 0xffffff02ff09ff09, 0x0affff04ffffff02
pp31: dq 0xffffff04ff0eff0a, 0x0effff0fffffff0f
; [...]

; Two useful macros (nasm syntax)
; 1 is the location of the vector,
; 2 is storage for the multiplied thingy,
; 3 is the constant index
```

```
%macro vec_mul 3
  movdqa %2, [ttim2 + %3*16]
  pshufb %2, %1
  movdqa %1, %2
%endmacro

; 1 is the accumulator,
; 2 is the multiplied vector,
; 3 is storage for the shuffled thingy,
; 4 is the index for the shuffle
%macro pp_accu 4
  movdqa %3, %2
  pshufb %3, [pp10 + %4*16]
  pxor   %1, %3
%endmacro

; the input is in xmm0, with only the four lsb of each byte set
_m64:
  pxor xmm1, xmm1 ; init the accumulator
.mainstuff:
  ; 1.x
  pp_accu xmm1, xmm0, xmm2, 0
  pp_accu xmm1, xmm0, xmm2, 1
  pp_accu xmm1, xmm0, xmm2, 2
  ; 2.x
  vec_mul xmm0, xmm2, 0
  pp_accu xmm1, xmm0, xmm2, 3
  pp_accu xmm1, xmm0, xmm2, 4
  pp_accu xmm1, xmm0, xmm2, 5
  ; 3.x
  vec_mul xmm0, xmm2, 1
  pp_accu xmm1, xmm0, xmm2, 6
  pp_accu xmm1, xmm0, xmm2, 7
; [...]
.fin:
  ; the result is in xmm0, and still of the same form
  movdqa xmm0, xmm1
  ret
```

Figure 6.5 – Part of an encoder for $\mathcal{C}_2$ using Algorithm 6.2

Chapter 7

The Littlun S-box and the Fly block cipher

1 Introduction

Since the late 1990's and the end of the AES competition, the academic community and the industry have been provided with excellent block ciphers. In most cases where a cipher is needed, AES [NIS01] can readily be used and there is currently little need for a replacement. Consequently, the symmetric cryptographic community shifted focus to *e.g.* the wider picture of *authenticated encryption* through the Caesar competition, or to more specific applications of block ciphers. In the latter case, an important topic is the design of "lightweight" block ciphers intended to be implemented on low-cost, resource-constraint devices. An early successful example following this trend is the block cipher Present [BKL+07], which can be implemented in small hardware circuits. Most lightweight algorithms similarly target a few platforms on which they are expected to perform particularly well; good performance in other cases are however not usually expected and lightweight ciphers are in general not very versatile. Typical platforms of interest include hardware circuits and 8-bit to 32-bit microcontrollers. Recently, NIST released a draft report on lightweight cryptography with the objective of creating a portfolio of lightweight primitives through an open process [NIS16]. The topic of lightweight cryptography is thus ever more timely.

In this chapter, we design a conceptually simple block cipher targeting efficient *light* implementations on 8-bit microcontrollers, where by *light* implementations, we mean in particular that the size of the code is small, typically of the order of 200 bytes. If more resources are available, the best current block cipher is probably the AES, see *e.g.* [BSS+15]. The chief academic proposal to date for this scenario is the Pride block cipher, that was presented at CRYPTO 2014 [ADK+14]; notable "non-academic" ciphers for the same scenario are the "NSA ciphers" Simon and Speck [BSS+13]. Our block cipher is built around Littlun-1, a compact 8-bit S-box with branch number 3. This allows to define a round function similar to a scaled-up variant of Present, composing the S-box application with a simple bit permutation. As a bit permutation obviously does not increase the number of active bits, an important part of the diffusion in such a cipher is played by the S-box. The typical measure of the quality of the diffusion of an S-box is its "branch number" which plays a role similar to the minimum distance of the linear codes used in AES-like designs and in the previous chapter Chapter 6. We thus get a trade-off between hardware and light software implementations: Littlun-1 is more expensive in hardware than two applications of the S-box of Present, but the bit permutation is simple to implement with 8-bit rotations. Owing to Golding, we name our block cipher "Fly".

Excluding on-the-fly key expansion, the round function of Fly costs four instructions less to implement than Pride's on AVR. Using the good branch number of Littlun-1, we can show that with a similar number of rounds, Fly is more resistant than Pride to statistical attacks. This is all the more relevant as the security margin of Pride seems to be quite thin [ZWWD14]. Taking the key-schedule into account, one round of Fly costs eight more instructions than one round of Pride. However, unlike Pride, we do not use an FX construction for the key-schedule and thus the generic security of Fly does not decrease with the amount of data available to the adversary. Dinur also showed how the FX construction can lead to more efficient time-memory-data trade-offs [Din15].

As implementations on resource-constraint devices are more likely to be vulnerable to side-channel attacks, one should also consider the additional cost of protection against, say, differential power analysis when evaluating schemes that target such platforms. In that respect, the small number of gates necessary to implement the Littlun-1 S-box as well as its simple expression in terms of light 4-bit S-boxes allow one to produce masked implementations of Fly with limited overhead.

Related work. The block cipher literature is so numerous that most new proposal will bear some similarity with past designs. In that respect, apart from Present, Fly is quite similar to Rectangle [ZBL$^+$14], which also combines a Serpent-like bitsliced application of an S-box [BAK98] with a rotation-implemented bit permutation. However, the S-box in Rectangle is on 4 bits, it does not have a branch number of 3 and the rotations are on 16-bit words. The construction of the Littlun S-box uses the Lai-Massey structure from the IDEA block cipher [LMM91]; this structure was already used to build the second S-box of the Whirlpool hash function [BR03] and the S-box of the block cipher Fox [JV04].

2 Preliminaries

We start by giving an overview of the main notions that will be used in evaluating the cryptographic properties of our construction. A more detailed and rigorous treatment of the following can for instance be found in [Rou15].

Although we will mostly consider S-boxes as defined over binary strings we may see an n-bit S-box as a mapping $\mathbf{F}_2^n \to \mathbf{F}_2^n$ whenever convenient, in particular so that the addition of a difference to an S-box input is a well-defined operation.

Definition 7.1 (Differential uniformity of an S-box)**.** Let $\mathcal{S}$ be an n-bit S-box. We define its *difference distribution table* (or DDT) as the function $\delta_{\mathcal{S}}$ defined extensively by:

$$\delta_{\mathcal{S}}(a, b) \coloneqq \#\{x \in \mathbf{F}_2^n | \mathcal{S}(x) + \mathcal{S}(x + a) = b\}.$$

The *differential uniformity* $\mathcal{D}$ of $\mathcal{S}$ is defined as:

$$\max_{(a,b)\neq(0,0)} \delta_{\mathcal{S}}(a, b).$$

Put another way, an n-bit S-box with differential uniformity $\mathcal{D}$ has a maximal differential probability of $\mathcal{D}/2^n$ over its inputs.

Definition 7.2 (Linearity of an S-box)**.** Let $\mathcal{S}$ be an n-bit S-box. We define its *linear approximation table* (or LAT) as the function $\ell_{\mathcal{S}}$ defined extensively by:

$$\ell_{\mathcal{S}}(a, b) \coloneqq \sum_{x\in\mathbf{F}_2^n} (-1)^{\langle b,\mathcal{S}(x)\rangle+\langle a,x\rangle}.$$

The *linearity* $\mathcal{L}$ of $\mathcal{S}$ is defined as:

$$\max_{(a,b)\neq(0,0)} |\ell_{\mathcal{S}}(a, b)|.$$

Roughly speaking, the linearity measures the maximum absolute difference between how many times a non-trivial linear approximation takes the value 1 and how many times it takes the value 0. For an n-bit S-box, it is therefore twice the difference between 2^{n-1} and how many times either value is taken. In particular, if we define the *bias* b of a probability p as $p-1/2$, it means that the absolute bias of any linear approximation of an n-bit S-box of linearity $\mathcal{L}$ is upper-bounded by $(\mathcal{L}/2)/2^n$. We will in fact mostly use the alternative notion of squared *correlation* of linear approximations, where the correlation of a given approximation is defined as $\mathrm{C}_{\mathcal{S}}(a, b) \coloneqq \ell_{\mathcal{S}}(a, b)/2^n$, *i.e.* twice the bias.

Definition 7.3 (Differential branch number of an S-box)**.** The *differential branch number* of an S-box $\mathcal{S}$ is:

$$\min_{\{(a,b)\neq(0,0)\,|\,\delta_{\mathcal{S}}(a,b)\neq 0\}} \mathrm{wt}(a) + \mathrm{wt}(b),$$

where $\mathrm{wt}(x)$ is the Hamming weight of x.

Definition 7.4 (Linear branch number of an S-box)**.** The *linear branch number* of an S-box $\mathcal{S}$ is:

$$\min_{\{a\neq 0,b\neq 0\,|\,\ell_{\mathcal{S}}(a,b)\neq 0\}} \mathrm{wt}(a) + \mathrm{wt}(b).$$

Definition 7.5 (Algebraic normal form)**.** Let $f : \mathbf{F}_2^n \to \mathbf{F}_2$ be an n-bit Boolean function, its *algebraic normal form* (or ANF) is defined as the unique polynomial $g \in \mathbf{F}_2[x_0, x_1, \ldots x_{n-1}]/\langle x_i^2 - x_i\rangle_{i<n}$ such that for all $x \in \mathbf{F}_2^n$, $f(x) = g(x[0], \ldots, x[n-1])$. Similarly, the ANF of an n-bit S-box $\mathcal{S}$ is the sequence of the ANFs of its n constituent Boolean functions $\langle\mathcal{S}(\cdot), e_i\rangle$, with (e_i) the canonical basis of $\mathbf{F}_2^n$.

The previous definitions focused on properties of individual S-boxes; in particular they involved no key or other kind of parameterisation. As we are eventually interested in the properties of block ciphers, *i.e.* families of permutations indexed by a key, we may also require definitions that average some of the above properties in a meaningful way. We then get the two following definitions, where we naturally extend the notion of differential uniformity and linearity to any function rather than just S-boxes.

Definition 7.6 (Maximum expected differential probability of a family of functions)**.** The *Maximum expected differential probability* (MEDP) of $\mathrm{F} : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ is defined as:

$$\max_{(a,b)\neq(0,0)} \frac{1}{\#\mathcal{K}} \cdot \sum_{k\in\mathcal{K}} \frac{\delta_{\mathrm{F}(k,\cdot)}(a,b)}{2^n}.$$

Definition 7.7 (Maximum expected linear potential of a family of functions)**.** The *Maximum expected linear potential* (MELP) of $\mathrm{F} : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ is defined as:

$$\max_{(a,b)\neq(0,0)} \frac{1}{\#\mathcal{K}} \cdot \sum_{k\in\mathcal{K}} \left(\frac{\ell_{\mathrm{F}(k,\cdot)}(a,b)}{2^n}\right)^2.$$

The MEDP and MELP of (reduced versions of) a block cipher are good indicators of its resistance against differential and linear cryptanalysis. Unfortunately, it is usually a hard problem to compute the MEDP and MELP for a non-trivial number of rounds of a block cipher. However, in the case of SPN ciphers, it is generally easier to derive upper bounds on the probability and potential of given differential and linear *trails* (or *characteristics*), for which the intermediate differences and linear masks are specified at every round. For some SPN ciphers, it may even be possible to provide bounds for *any* trail on a given number of rounds. These notions are then often used instead of the MEDP and MELP to argue about the security of the cipher.

In the following, we consider an iterative n-bit block cipher $\mathcal{E}$ with round function $\mathcal{R}$. If we write $\mathcal{R}_k$ the round function with pre-addition of a subkey k, *i.e.* $\mathcal{R}_k(x) = \mathcal{R}(x \oplus k)$, then the r-round $\mathcal{E}$ is defined as $\mathcal{E}(K,x) = (\mathcal{R}_{k_{r-1}} \circ \cdots \circ \mathcal{R}_{k_0}(x)) \oplus k_r$, where the tuple $(k_0,\ldots,k_r)$ is the image of K by some key expansion mapping. We then have:

Definition 7.8 (Differential trail)**.** An r-round *differential trail* for a block cipher $\mathcal{E}$ is an $(r+1)$-tuple $(a_0,\ldots,a_r)$ of n-bit *differences.* An input x to $\mathcal{E}$ is said to *follow* the trail for the subkeys $(k_0,\ldots,k_r)$ if $\forall\, i < r$, $\mathcal{R}_{k_i}(x) \oplus \mathcal{R}_{k_i}(x \oplus a_i) = a_{i+1}$.
When the key (equivalently the tuple of subkeys) of $\mathcal{E}$ is fixed, we call *differential trail probability* (DTP) of a trail the probability that it is followed by an input to $\mathcal{E}$.

Definition 7.9 (Linear trail)**.** An r-round *linear trail* for $\mathcal{E}$ is an $(r+1)$-tuple $(a_0,\ldots,a_r)$ of n-bit *linear selection masks.*
For a fixed key of $\mathcal{E}$, the *linear trail correlation* (LTC) of a trail is given by:

$$\prod_{i=0}^{r-1} \mathrm{C}_{\mathcal{R}_{k_i}}(a_i, a_{i+1}).$$

In the same spirit as Definition 7.6 and Definition 7.7, we will usually be interested in the expected values of the DTP and the squared LTC over the choice of the key.

The above definitions are in fact not completely specific to SPN block ciphers. However, for such ciphers, one can additionally define the notion of *active* S-box in a trail, which we define informally as follows:

Definition 7.10 (Active S-box in a trail)**.** An i^{th}-round S-box is *active* in an r-round differential (resp. linear) trail $(a_0, \ldots, a_r)$ if it has a non-zero input difference as per a_i (resp. if some of its input bits are selected by the mask a_i).

For example, taking the block and S-box size to be respectively 32 and 4 and using hexadecimal notation, all S-boxes are active in the 2-round trails $(\texttt{0xFFFFFFFF}, \texttt{0xFFFFFFFF}, \texttt{0xFFFFFFFF})$ and $(\texttt{0x12345678}, \texttt{0x9ABCDEFE}, \texttt{0xDCBA9876})$, and only one S-box is active at every round of the trail $(\texttt{0x10000000}, \texttt{0x30000000}, \texttt{0x00000004})$.

We can define the *weight* of a trail (a_n) as the number of S-boxes active for (a_n). Note that whether an S-box is active or not in a trail solely depends on the definition of the latter and not, say, on subkey values. Consequently, even though the probability of the differential trails of a block cipher (say) may in fact depend on the key, the possible weights of trails of non-zero probability are only a consequence of the round function.

If we allow ourselves to make some independence hypotheses, assuming that consecutive round functions behave independently of each other, then the number of active S-boxes in a differential trail can be used to upper-bound its expected probability. This is expressed informally as the following:

Assumption 7.1. *Let (a_n) be a differential trail of weight w for the SPN block cipher $\mathcal{E}$ which uses a unique b-bit S-box $\mathcal{S}$ of differential uniformity $\mathcal{D}$. Call $p_{\mathcal{S}} \coloneqq \mathcal{D}/2^b$ the maximum differential probability of $\mathcal{S}$. Then the expected probability over the choice of K that an input to $\mathcal{E}(K, \cdot)$ follows the trail (a_n) is upper-bounded by $(p_{\mathcal{S}})^w$.*

In other words, Assumption 7.1 is a consequence of the assumption that the expected probability of an r-round differential trail is equal to the product of the expected probabilities of the one-round trails it is made of, *i.e.* the so-called Markov assumption [LMM91].

The case of linear trails is treated somewhat similarly. In particular, from Definition 7.9, the correlation of a trail for a fixed key is easy to compute from the correlation of the one-round trails it is made of. Hence, we get:

Proposition 7.1. *Let (a_n) be a linear trail of weight w for the SPN block cipher $\mathcal{E}$ which uses a unique b-bit S-box $\mathcal{S}$ of linearity $\mathcal{L}$. Call $\mathrm{C}_{\mathcal{S}} \coloneqq \mathcal{L}/2^b$ the maximal absolute correlation of a linear approximation for $\mathcal{S}$. Then, fixing the key K, the absolute correlation of the trail (a_n) for $\mathcal{E}(K, \cdot)$ is upper-bounded by $(\mathrm{C}_{\mathcal{S}})^w$ and thus the squared correlation by $(\mathrm{C}^2_{\mathcal{S}})^w$.*

Depending on the properties of the S-box and on the block size of a cipher, one can use the upper bounds from Assumption 7.1 and Proposition 7.1 to set an objective for the minimum number of active S-boxes in any trail.

For resistance against differential cryptanalysis, one typically requires trails to have at least wd differentially active S-boxes so that $(p_{\mathcal{S}})^{wd} < D2^{-n}$, with n the block size of the cipher under consideration and D a number. This comes from the fact that differentials of probability 2^{-n+1} are bound to exist, and one wishes both that no single trail has a higher expected probability and more importantly that no set of more than D trails with a minimum number of active S-boxes and equal starting and ending differences can be found.

In the case of linear cryptanalysis, one similarly typically requires at least wl linearly active S-boxes in any trail so that $(\mathrm{C}_{\mathcal{S}}^2)^{wl} < L2^{-n}$. This comes from the fact that the data complexity required to exploit an approximation of correlation C grows with the square of C, and that we do not want the net number of trails with equal starting and ending masks, activating the minimum number of S-boxes, and interfering constructively, to be likely more than L for some fixed keys.

The entire approach based on counting the number of active S-boxes becomes particularly useful if one is able to use the structure of the round function of a block cipher to lower-bound the number of differentially and linearly active S-boxes for any trail of a given number of round. If such a bound can be computed, the objective on the number of S-boxes then becomes one on the number of rounds.

3 The Littlun S-box construction

3.1 The Lai-Massey structure

Our S-box uses the *Lai-Massey structure*, which was proposed in 1991 for the design of the block cipher IDEA [LMM91]. The structure is similar in its objective to a Feistel or Misty structure (see *e.g.* [CDL15] for definitions of the Feistel and Misty structures), as it allows to construct n-bit functions out of smaller components. It is in particular well-suited to build efficient 8-bit S-boxes from 4-bit S-boxes all the while amplifying the good cryptographic properties of the 4-bit S-boxes. It was already used as such for the design of the second S-box of the Whirlpool hash function [BR03] (an early version of Whirlpool used a randomly-generated S-box) using five 4-bit S-boxes, see Figure 7.1a, and for the design of the S-box of the Fox block cipher [JV04] which uses a three-round iterated structure. In our construction, we use only one round of the more classical variant of the structure, with only three S-boxes, see Figure 7.1b, which allows nonetheless to square the differential probability and the linearity of the underlying 4-bit S-box.

The choice of the Lai-Massey structure was mainly motivated by our objective of building an S-box with a branch number of three, both differential and linear; this will in turn be useful to design a good lightweight round function. Indeed, it is easy to see that the S-box will have this property by construction for the differential branch number as soon as the 4-bit S-boxes have differential branch number three, and such S-boxes are well-known, see *e.g.* Serpent [BAK98]. So much cannot be said however for the linear branch number, as no 4-bit S-box with optimal resistance to differential and linear cryptanalysis exists with this property, as demonstrated by an exhaustive search we performed on the optimal classes described *e.g.* in [LP07]. In fact, we are not aware of previous examples of 8-bit S-boxes with this feature either.

Other good properties of the structure are that it yields S-boxes with a circuit depth of two S-boxes and it allows for efficient vector implementations using SIMD instructions, see Section B.2. On the downside, it requires the 4-bit S-boxes to be permutations if we want the 8-bit S-box to be one. Canteaut, Duval and Leurent recently showed how the absence of such a restriction for Feistel structures could be used to build compact S-boxes with particularly low differential probability [CDL15]. We should note however that for the applications we have in mind, see Section 5, the linearity of the S-box is as important as the differential probability, and the linearity of the S-box of Canteaut *et al.* is average, and in particular not better than ours. Finally, we should mention that the good and bad points of the Lai-Massey structure cited so far are shared with the Misty structure. Choosing Lai-Massey in our case was mainly due to a matter of taste, though it is noteworthy that Misty yields S-boxes with a rather sparse algebraic expression, meaning that the polynomials in the ANF of such S-boxes tend to have many zero coefficients.

3.2 An instantiation: Littlun-1

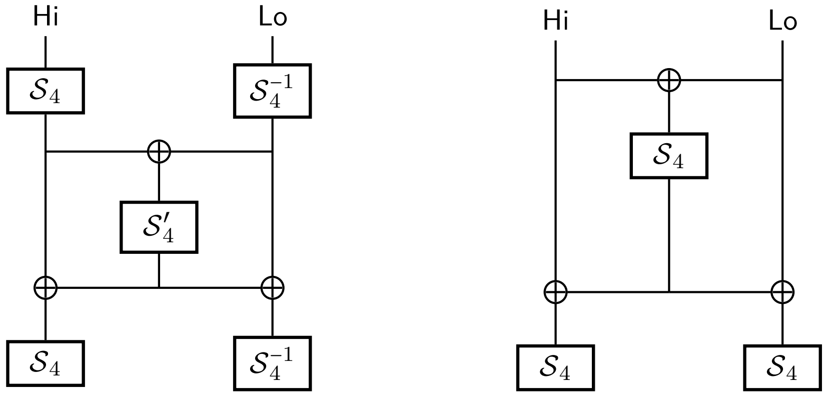
We now define Littlun-1, a concrete instantiation of the Lai-Massey structure which achieves a differential and linear branch number of three. Although we have seen that we could guarantee this in the differential case by using a 4-bit S-box of differential branch number three, this is actually not necessary and we use instead a very compact member of the *class 13* of Ullrich *et al.* [UDI+11]. This S-box uses only 4 non-linear and 4 XOR gates, which is minimal for an optimal S-box of this size. This leads to an 8-bit S-box using 12 non-linear and 24 XOR gates. We give the table of the 4-bit S-box "Littlun-S4" in Figure 7.9 and of the complete 8-bit S-box in Figure 7.10, in Section B.1, and conclude this section by a summary of the cryptographic properties of Littlun-1.

Proposition 7.2 (Statistical properties). *The differential uniformity of* Littlun-1 *and of its inverse is 16 and its linearity is 64, as proven by a direct computation. Its DDT and LAT are shown in Figure 7.7 and Figure 7.8 respectively.*

In essence, Proposition 7.2 means that the probability (taken over all the inputs) of any non-trivial differential relation going through the S-box is upper-bounded by 2^{-4} and the squared correlation of any non-trivial linear approximation is upper-bounded by 2^{-4}.

Proposition 7.3 (Diffusion properties). *The differential and linear branch number of* Littlun-1 *and of its inverse is 3.*

As we already mentioned, several 4-bit S-boxes from the literature have a differential branch number of 3, and it is not hard to construct 8-bit S-boxes with this property from them. This is not the case for the linear branch number, and we find the fact that Littlun-1 has such a property to be quite more remarkable.



(a) Lai-Massey as in the Whirlpool second S-box (b) Lai-Massey as in the Littlun construction

Figure 7.1 – The Lai-Massey structure

Proposition 7.4 (Algebraic properties). *The maximal degree of the ANF of* Littlun-1 *is 5 in four of the eight output bits, 4 in two other and 3 in the remaining two. The maximal degree of its inverse is 5 in six of the eight output bits and 4 in the other two.*

4 Implementation of Littlun-1

4.1 Hardware implementation

We give a circuit using OR, AND and XOR gates implementing Littlun-S4 in Figure 7.12 in Section B.3. A hardware implementation of the entire S-box can easily be deduced by plugging this circuit into the one of Figure 7.1b. As previously mentioned, Littlun-1 can be implemented with 12 non-linear (OR and AND) gates and 24 XOR gates. With a typical cell library such as the Virtual Silicon standard cell library, OR and AND gates cost 1.33 gate equivalent (GE), and XOR gates 2.67 GE. Thus synthesising the S-box with this library would cost 80 GE.

4.2 Bitsliced software implementation

One of our main objective w.r.t. implementation is to obtain an S-box with an efficient *bitsliced* implementation in software. This is closely related to the simplicity of the circuit of the S-box, though not exactly equivalent. We purposefully choose a 4-bit S-box from the *class 13* of Ullrich *et al.* [UDI+11] because of its very efficient bitsliced implementation that requires only 9 instructions on a wide variety of platforms. Such an implementation is given in Figure 7.2. From this, it is easy to obtain an efficient bitsliced implementation for the whole S-box, as shown in Figure 7.3. This implementation typically requires 43 instructions and 13 registers.

```
t  = b;    b |= a;    b ^= c; // (B): c ^ (a | b)
c &= t;    c ^= d;            // (C): d ^ (c & b)
d &= b;    d ^= a;            // (D): a ^ (d & B)
a |= c;    a ^= t;            // (A): b ^ (a | C)
```

Figure 7.2 – Snippet for a bitsliced `C` implementation of Littlun-S4 with input and output in registers a, b, c, d (the word holding the most significant bit is taken to be a), using one extra register t.

```
t = a ^ e;
u = b ^ f;
v = c ^ g;
w = d ^ h;
S4(t,u,v,w); // uses one more extra register x
a ^= t;    e ^= t;
b ^= u;    f ^= u;
c ^= v;    g ^= v;
d ^= w;    h ^= w;
S4(a,b,c,d); // reuses t as extra
S4(e,f,g,h); // reuses u as extra
```

Figure 7.3 – Snippet for a bitsliced `C` implementation of Littlun-1, using the code of Figure 7.2 as subroutine. The input and output registers are a, b, c, d, e, f, g, h (with the most significant bit in word a), the five extra registers are t, u, v, w, x.

4.3 Masking

The low number of non-linear gates needed to implement Littlun-1 makes it a suitable choice for applications where counter-measures against side-channel attacks are required. Indeed, it directly implies a lower cost when using Boolean masking schemes, both hardware and software, which represent the primitive to be masked as a circuit [ISW03, CGP+12]. In particular, Littlun-1 is competitive with the S-boxes proposed by Grosso *et al.* [GLSV14]: it has the same gate count as the S-box used for Robin and only one more non-linear and one less XOR gate than the one used for Fantomas. All three S-boxes are comparable in terms of cryptographic properties. The S-box of Canteaut *et al.* is slightly more expensive, requiring two more non-linear gates [CDL15]; it is however stronger against differential cryptanalysis, its differential uniformity being equal to 8.

One could alternatively consider that the chief non-linear component to take into account in that context is actually Littlun-S4, the 4-bit S-box underlying Littlun-1, rather than the full S-box. Indeed, any cryptosystem using Littlun-1 in combination with an arbitrary linear layer can be re-written as using only Littlun-S4 for its non-linear part. In that respect, the number of non-linear gates to consider for masking would only be 4. One could however object that additional factors need to be taken into account, such as for instance the total number of application of the S-box in an execution of the cipher. Yet if we jump a little ahead and consider the block cipher Fly of Section 5, we can see that in terms of the 4-bit S-box, Fly needs $20 \times 8 \times 3 = 480$ calls to Littlun-S4, which is comparable to the $31 \times 16 = 496$ of Present, discounting the key-schedule.

The ability to express Littlun-1 only in terms of a 4-bit S-box is also convenient when considering threshold implementations, although these chiefly apply to hardware implementations, which are not the focus of this chapter. For instance, it allows one to benefit from the recent progresses in such protected implementations of small S-boxes [BNN+15].

We further discuss the cost of masking a concrete block cipher instance using Littlun-1 in Section 5.3.

4.4 Inverse S-box

The inverse Littlun-1^{-1} of Littlun-1 is slightly costlier to implement, because of a more expensive inverse for Littlun-S4. As a circuit, the latter requires 5 XOR gates, 4 non-linear (OR and AND) gates and one NOT gate costing 0.67 GE. The total hardware cost of Littlun-1^{-1} is thus 90 GE.

Software bitsliced implementations are also more expensive. We give a snippet for the inverse of Littlun-S4 in Figure 7.4 that requires 11 instructions and 5 registers. The complete inverse can be implemented with 49 instructions and 13 registers in a straightforward adaptation of Figure 7.3. However, because the output registers form a non-trivial permutation of the input ones, additional instructions may also be needed in the cases where this cannot be dealt with implicitly.

```
t  = c;     c &= b;     c ^= d; // (A): d ^ (b & c)
d |= t;     d ^= a;             // (B): a ^ (c | d)
a &= c;     a ^= b;     a ^= d; // (C): b ^ B ^ (a & A)
b  = ~b;    b &= d;     b ^= t; // (D): c ^ (~b & B)
```

Figure 7.4 – Snippet for a bitsliced C implementation of the inverse of Littlun-S4 with inputs in registers a, b, c, d (the word holding the most significant bit is taken to be a), using one extra register t. The output is in c, d, a, b.

5 An application: the Fly block cipher

In this section, we present the Fly block cipher as an application of the Littlun-1 S-box. It is a 64-bit block cipher with 128-bit keys. Thanks to the branch number of the S-box, it is easy to design a round function with good resistance to statistical attacks by combining its bitsliced application with a simple bit permutation. This results in a cipher with a structure similar to Present [BKL⁺07] with a tradeoff: the S-box is bigger, and thus more expensive to implement, in particular in hardware, but the permutation is simpler, and thus cheaper to implement in software. This cipher was designed to be used in the same cases as Pride, and its chief implementation target is 8-bit microcontrollers.

5.1 Specifications

We first give the specification of the round function $\mathcal{R}_{\text{Fly}}$ of Fly. It takes a 64-bit block and 64-bit round key as input. Let $x := (x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7)$, $rk := (rk_0||rk_1||rk_2||rk_3||rk_4||rk_5||rk_6||rk_7)$ be such an input, with x_i, rk_i 8-bit words. The big endian convention is used to convert from x and rk to the x_is and rk_is.

Let us first define $f_i(t) := \iota_i(t_0)||\kappa(t_1)||t_2||t_3||t_4||t_5||t_6||t_7$, with $\kappa(x) := x \oplus \texttt{0xFF}$; $\iota_i(x) := x \oplus \mathcal{C}(i)$, $\mathcal{C}(i)$ being a round constant produced as the i^{th} iteration of the affine LFSR $\mathcal{C}$ shown in Figure 7.5, initialised with zero. Algebraically speaking, $\mathcal{C}$ implements the mapping $x \mapsto \alpha(x+\alpha^7)$ in $\mathbf{F}_2[\alpha]/\langle\alpha^8+\alpha^7+\alpha^3+\alpha^2+1\rangle$. Note however that the mapping of elements of this field to the state of Figure 7.15, or equivalently 8-bit machine words, uses the inverse ordering of the usual convention, *i.e.* the highest coefficient is stored in the LSB. This, as well as the addition of α^7 prior to the multiplication by α, is done to ease software implementations of this round constant generation. An example of such an implementation is given in Figure 7.15.

```
flip = r[0] ^ 1;     r[n] = r[n + 1], n = 6...0;     r[7] = 0;
r[0] = r[0] ^ flip; r[4] = r[4] ^ flip;
r[5] = r[5] ^ flip; r[7] = r[7] ^ flip;
```

Figure 7.5 – The affine LFSR whose i^{th} iteration starting from a zero state defines the i^{th} round constant. This pseudo-code assumes an 8-bit state r, whose entry $r[0]$ maps to the LSB in a machine representation.

We write Ark_i the addition of the i^{th} round key: $\text{Ark}_i(rk_i, x) := f_i(x \oplus rk_i)$; $\text{Bls}(x)$ a bitsliced application of Littlun-1 such as *e.g.* the one shown in Figure 7.3, with x_0 holding the most significant bits of the input to the S-boxes; and Rot the "ShiftRow" word-wise rotation with $\circlearrowleft$ denoting bitwise rotation to the left

$$\text{Rot}(x) := (x_0||x_1 \circlearrowleft 1||x_2 \circlearrowleft 2||x_3 \circlearrowleft 3||x_4 \circlearrowleft 4||x_5 \circlearrowleft 5||x_6 \circlearrowleft 6||x_7 \circlearrowleft 7)$$

which can alternatively be defined at the bit level as the permutation $\mathrm{P}(i) := (i + 8(i \bmod 8)) \bmod 64$ applied to a suitable binary representation of $x = b_0 \ldots b_{63}$. Then we simply have $\mathcal{R}_{\text{Fly}}(\cdot,\cdot) := \text{Rot} \circ \text{Bls} \circ \text{Ark}$. We give a graphical representation of the SPN structure of this round function at the bit level in Figure 7.6.

We propose two key-schedules, KS1 and KS2, depending on whether resistance to related-key attacks is required, in the case of KS2, or not. In order to distinguish between the two block ciphers, we write Fly for the default case where KS1 is used and Fly_{RK} when KS2 is used. We describe KS1 first, which in fact performs only an

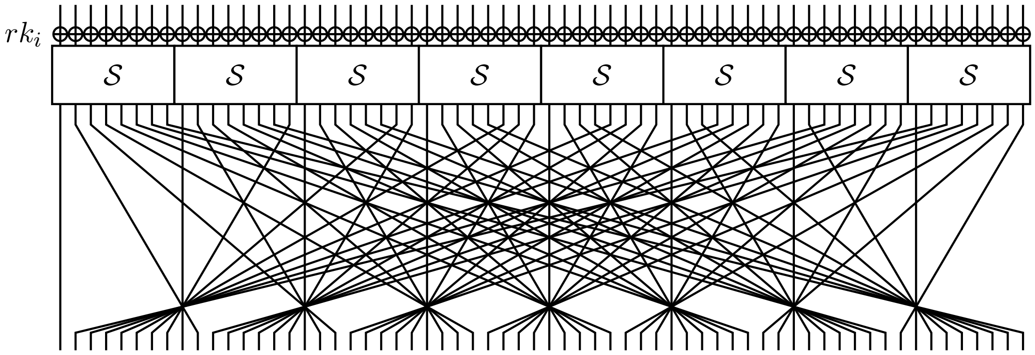


Figure 7.6 – The round function of Fly. Bits are numbered left to right from 0 to 63 w.r.t. the bit permutation. The addition of the round constant is omitted.

elementary scheduling: let $k \coloneqq k_0||k_1$ be the 128-bit master key, and k_0 (resp. k_1) its first (resp. second) half; then the sequence (rk_n) of round keys of KS1 is the simple alternation of k_0 and $k_0 \oplus k_1$ defined as $rk_i \coloneqq k_0 \oplus i \times k_1$, where $i \times k_1$ means the all-zero key for even values of i and k_1 otherwise. A round constant is also added in Ark through the function f_i to prevent self-similarity attacks.

For Fly to be resistant to related-key attacks, we use the same approach as Noekeon [DPAR00] to define KS2 as follows. Let us denote by $\text{Fly}(0,\cdot)/12$ twelve applications of the round function of Fly with the all-zero 128-bit key and define $k' \coloneqq k'_0||k'_1 = \text{Fly}(0,k_0)/12||\text{Fly}(0,k_1)/12$. Then KS2 is defined through Fly_{RK} as $\text{Fly}_{RK}(k,\cdot) \coloneqq \text{Fly}(k',\cdot)$.

The round function of Fly is applied 20 times, the same as Pride. The entire cipher can thus finally be defined as $\text{Fly}(k,\cdot) \coloneqq \text{Ark}(rk_{20},\cdot) \circ \mathcal{R}_{\text{Fly}}(rk_{19},\cdot) \circ \ldots \circ \mathcal{R}_{\text{Fly}}(rk_1,\cdot) \circ \mathcal{R}_{\text{Fly}}(rk_0,\cdot)$.

Design rationale

The core of Fly is the Littlun-1 S-box, which was designed to have a branch number of three. This allows to achieve a good diffusion when combining the S-box application with a simple bit permutation. The latter was chosen so that all eight bits at the output of an S-box go to one different S-box each; similarly, all input bits come from a different S-box. Unlike in Present, this permutation also has cycles of different lengths, namely 2 (on 8 values), 4 (on 16) and 8 (on 32). This might reduce the impact of linear and differential trail clustering. The round constants break the self-similarity and self-symmetry of the round function (through ι) and the self-symmetry of the S-box (through κ). This latter symmetry is actually already broken by ι and most of the times by the round keys, but using an extra constant allows for a simple and clean argument at a negligible cost.

The two components of the round function can be efficiently implemented on an 8-bit architecture through a bitsliced application of the S-box and word rotations respectively, cf. Section 5.3.

The two key-schedules were designed according to different possible scenarios. Most applications do not require resistance to related-key attacks and a simple alternating key-schedule is enough in that case. We chose not to use an FX construction as in Pride as we did not consider the slight gain in efficiency it offers to be worth the generic security loss it implies. In the spirit of Noekeon, we propose a second key-schedule to offer resistance to related-key attacks that consists in “scrambling” the master key with a permutation of good differential uniformity before it is used as in the first key-schedule.

5.2 Preliminary cryptanalysis

We now analyse the security of Fly against various types of attacks. Considering the similarity of the design with Present and the published analysis on this cipher, the most efficient attacks on Fly are likely to be variants of classical statistical (differential and linear) attacks, which we analyse first, in the single-key setting. We then give an overview of the resistance against other attack techniques.

5.2.1 Statistical attacks

We can use the branch number of Littlun-1 together with the properties of the bit permutation Rot to easily derive a lower bound on the number of differentially and linearly active S-boxes. Indeed, as the branch number is 3, we are guaranteed to have at least 6 active S-boxes every four rounds of any non-trivial differential or linear trail. This is a consequence of the following proposition:

Proposition 7.5. *There is no 3-round trail on* Fly *activating 1, then n, then 1 S-box, for any value of n.*

Proof. In a round following one round with a single active S-box, all n active S-boxes are active in a single bit of their input, and consequently each of their outputs activates at least 2 S-boxes. □

Following Section 2, the block size of Fly being 64 bits and the differential uniformity and linearity of its 8-bit S-box being 16 and 64 respectively, we want any differential trail to have more than 16 active S-boxes to preclude differential distinguishers, as $2^{-64} = (16/256)^{16}$; we also want any linear trail to have more than 16 active S-boxes, as $2^{-64} = ((64/256)^2)^{16}$. From Proposition 7.5, this happens in both cases after at most 12 rounds, after which at least 18 S-boxes are guaranteed to be active, both linearly and differentially. Even by discounting the additional 2 S-boxes and assuming that a distinguisher can be found for this amount of rounds, this gives a very comfortable margin of 8 rounds, which we estimate to be much beyond the ability of an attacker to convert the distinguisher into, say, key-recovery; in particular, this is twice the number of rounds needed for full diffusion. This also leaves some margin to ensure that even in the case where Fly would exhibit a strong differential or linear hull effect it would be unlikely for an attacker to be able to mount a meaningful attack. For instance, after 16 rounds, an attacker would need about 2^{32} "optimal" contributing differential trails to obtain a distinguisher with non-trivial probability, and would still be facing 4 rounds to mount an attack.

Thus, we conjecture that Fly with 20 rounds offers good resistance to statistical attacks.

Brief comparison with Present. The best attacks to date on Present are based on multidimensional linear cryptanalysis [Cho10, BTV16]. These attacks exploit the presence of linear trails that constantly activate only one S-box per round, *i.e.* "single-bit trails". As there are no such trails in Fly, we believe that these attacks would be less effective on the latter. Similarly, some other good attacks on Present exploit the fact that half of the bits of some groups of S-boxes remain in the same group [CS09], and there is no such property for Fly.

5.2.2 Other attacks

Algebraic attacks. We would like to estimate how many rounds of Fly are necessary for the degree of the cipher to reach the maximum of 63, as a lower degree could be exploited in algebraic attacks. Computing the exact degree of an iterated function is a difficult problem in general, but we should at least compute the upper bound of Boura, Canteaut and De Cannière to estimate how quickly the degree increases [BCD11]. In our case, this bound states that $\deg(G \circ F) \leq n - (n - \deg(G))/(n_0 - 2)$, where n is the block size and n_0 the size of the S-box. Combining this bound with the fact that the degree of the S-box is 5, and thus that $\deg(\text{Bls} \circ F) \leq 5 \cdot \deg(F)$, we can see that 5 rounds of Fly are necessary to reach a full degree; this increases to 6 rounds if we take 3 as the degree of the S-box, which is the minimum degree of its projections on one output bit. If we assume the latter bound to be an equality, any algebraic distinguisher on more than about twice this number, *i.e.* twelve rounds, is unlikely to exist. Even when relaxing this latter assumption, 20 rounds seem to be well enough to make Fly resistant to algebraic attacks.

Meet-in-the-middle attacks. We analysed how many rounds are necessary to ensure that every bit of the intermediate ciphertext depends on every bit of the key, as a basic way to estimate the resistance of Fly to meet-in-the-middle (MitM) attacks, which typically exploit the opposite effect. We did this by performing random trials with 2^{20} pairs of random keys and random plaintexts and found that this happens after at most 5 rounds. Any MitM attack on more than about twice this number of rounds is unlikely to exist, and we therefore conjecture that Fly is resistant to such attacks. Which key-schedule is used is irrelevant, as KS2 is equivalent to using KS1 with a different effective master key, which a MitM attacker can recover in the exact same way as the true master key produced by KS1.

Invariant subspace attacks. Invariant subspace attacks exploit the propensity of a round function to map inputs from a certain non-trivial affine coset to another, when in addition a trivial key-schedule and sparse round constants are used [LMR15]. As the two latter points appear in Fly, we analysed its round function in order to see if it could also meet the first, critical condition. We ran the automated search tool provided by the authors of [LMR15][1] for about 2^{36} iterations without finding any invariant subspace; this shows that with good probability, no such subspace of dimension greater than $64 - 36 = 28$ exists.

Integral attacks, impossible differentials, zero correlation. We did not analyse in detail the security of Fly against integral attacks, nor against impossible differentials and zero correlation attacks. Indeed, none of these techniques seem to be able to attack a significant number of rounds of bit-oriented ciphers such as Present (see *e.g.* [ZRHD08, CJF$^+$16, BC16], where the obtained distinguishers reach significantly less rounds than the best statistical ones) or Fly and we do not consider them to be a threat for our cipher.

Related-key attacks. We now study the resistance of Fly against XOR-induced, differential related-key attacks. Fly equipped with the simple key-alternating key-schedule KS1 offers nearly no resistance to related-key attacks. With KS2, however, an attacker is unable to control the differences between two different effective master keys $k'_0||k'_1$ and $\widetilde{k'_0||k'_1}$ with a probability much better than $2^{-2 \cdot 64}$, as each difference pair $(k_0, \widetilde{k_0})$ and $(k_1, \widetilde{k_1})$ goes through a permutation with maximum differential probability not significantly above 2^{-64}. Furthermore, unlike single-key differential attacks, which introduce differences on the plaintext, we do not expect an attacker to easily be able to force a change of keys if their effective master keys fail to verify a difference relation. Thus, even if a differential on KS2 with probability p higher than 2^{-128} were found, it would only lead to a related-key attack on a weak-key class of size $\approx p/2^{-128}$ or to an attack requiring a huge amount of keys. Putting everything together, we believe Fly_{RK} to be resistant to XOR-induced related-key attacks.

Known- and chosen-key distinguishers, compression function mode. We do not claim any resistance of Fly against known-key and chosen-key distinguishers. We do not make any claim about its suitability to build a cryptographically strong compression function.

5.3 Implementation

5.3.1 Microcontrollers implementations

The S-box application can take advantage of the bitsliced expression of Littlun-1 from Section 4, which can easily be implemented with instructions available on the cheapest ATtiny chips [Atm07]. It is even possible to save 2 instructions from the 43 quoted in Section 4 on higher-end architectures such as the ATmega family [Atm13] by using word-wise 16-bit `movw` instructions, resulting in the implementation given in Figure 7.13 of

[1] Available at `http://invariant-space.gforge.inria.fr/`.

Section C. A straightforward implementation of the inverse S-box application requires 59 instructions —a significant overhead of 44 %. However, as a lightweight cipher is precisely used in cases where the available resources are limited, we would mostly expect it to be used in a mode of operation that only uses encryption, such as *e.g.* CTR for encryption only or CLOC [IMGM14] for authenticated-encryption. Hence we do not believe that a slower inverse is a significant drawback.

Even though the AVR instruction set does not include rotations by an arbitrary constant, the permutation Rot can still be compactly implemented with only 11 instructions, as shown in Figure 7.14 of Section C.

The entire substitution and permutation layers of Fly can therefore be implemented with only 52 instructions on ATmega (54 on ATtiny), which is 4 less than the 56 of Pride [ADK+14], while at the same time having at least 1.5 times more *equivalent* active S-boxes every four rounds: there are at least four active 4-bit S-boxes of maximum differential probability 2^{-2} and best squared correlation 2^{-2} every two rounds of Pride and there are at least 6 active 8-bit S-boxes of maximum differential probability 2^{-4} and best squared correlation 2^{-4} every four rounds of Fly.

On-the-fly computation of one round-key of the key-schedule KS1 can be done in 8 instructions. The complete key expansion and round constant addition can be done in 24 instructions as shown in Figure 7.15.

The total round function of Fly including the key-schedule can thus be implemented in 76 instructions, which is eight more than Pride. Note however that the conjectured security margin of Fly is much larger, and unlike Pride, its resistance to generic attacks does not decrease with the amount of data available to the adversary. In contrast, Pride uses an FX construction, where one half of its 128-bit key is only used for pre- and post-whitening. This leads to a simple way to merge on-the-fly round-key generation with the round function, but significantly degrades the security of the cipher to generic attacks from 128 bits to $128 - \log(D)$, with D the amount of data available to an attacker [KR01], while also leading to more efficient time-memory-data trade-offs [Din15].

5.3.2 Masked implementations

We have just considered the good performance of Fly on AVR processors. However, on such platforms, discounting the overhead of protection against side-channel attacks may be misleading. Indeed, these devices are rather prone to leakage, and it might not be entirely reasonable to deploy unprotected cryptosystems on them [SOR+14]. Consequently, we consider here the cost of masked implementations of Fly at various orders, and compare them to Pride and the “NSA ciphers” Simon and Speck [BSS+13] in the same setting. All the masked implementations have been generated automatically by using the compiler of Barthe *et al.* [BBD+15]. This compiler takes a C implementation of a cipher as an input and generates C code for a corresponding masked implementation. This code is then instrumented to count the number of basic instructions, *e.g.* logical and arithmetic, executed in the encryption of one block.

We report the cost in terms of number of instructions for the studied ciphers and configurations in Table 7.1. The first column gives the name of the cipher, followed

Cipher	Unmasked	Order 2	Order 4	Order 7	Order 11
Fly (8)	1909	10741	27253	66421	145525
Simon64-128 (8)	3400	14056	30344	65336	131704
Simon64-128 (32)	1012	3926	8240	17336	34364
Pride (8)	1374	22922	60550	150592	333368
Speck64-128 (32)	486	48198	132652	337843	757983

Table 7.1 – Count of operations needed to encrypt one block with each cipher, masked at various orders.

by the basic word-size used in the implementation; for 8-bit microcontrollers, the most relevant value for this number is understandingly 8. The next columns give the number of instructions over the same word-size as the cipher taken to encrypt one block with an unmasked implementation, and then masked implementations at various orders; an order-t implementation ensures security in the t-probing model [ISW03].

From these results, a first important remark we can make is that neither Pride nor Speck seem to be well suited to masked implementations. This is due to their conjoined use of bitwise operations and integer modular addition, namely eighty 8-bit additions for Pride used in its key-schedule and fifty-four 32-bit additions for Speck64-128. Masking bitwise operations can be done relatively efficiently by using a Boolean sharing scheme but it is costly to do so with an additive scheme, while the converse holds for modular additions. In practice, the compiler of Barthe *et al.* uses the algorithm of Coron *et al.* from CHES 2014 to mask modular additions with a Boolean scheme [CGV14]. If we were to restrict ourselves to first-order masking, it would be possible to use the more efficient algorithm of Coron *et al.* from FSE 2015 [CGTV15].

On the contrary, both Fly and Simon are quite efficient to mask with a Boolean scheme. Note that we implemented Simon64-128 in two ways: one using 8-bit words, suitable for 8-bit microcontrollers and thus directly comparable with the intended use of Fly, and one using 32-bit words which is more straightforward in a way as all instances of Simon on $2n$-bit blocks can be expressed naturally with n-bit arithmetic.

On 8-bit platforms, our unmasked implementation of Fly is more efficient than Simon64-128. This advantage is maintained up to a small number of shares, but starting from 8, *i.e.* an implementation secure at order 7, Simon becomes more efficient. This behaviour can be explained by the breakdown of the cost for the two ciphers: although an unmasked Simon64-128 is costlier than Fly overall, our masked implementation of the former uses only 176 *refresh* and *and* operations, while Fly needs 240. As the cost of these operations is quadratic in the number of shares, implementing Simon at high order becomes cheaper than implementing Fly, but the latter starts with a significant initial advantage.

In conclusion both Fly and Simon are well suited to masked implementations on

8-bit microcontrollers. While the latter is the most efficient of the two for 7+-order implementations, Fly is cheaper in the in our opinion more relevant case of low-order ones.

We give the code of a masked version of Fly in Section F.

A Complement on the properties of Littlun-1

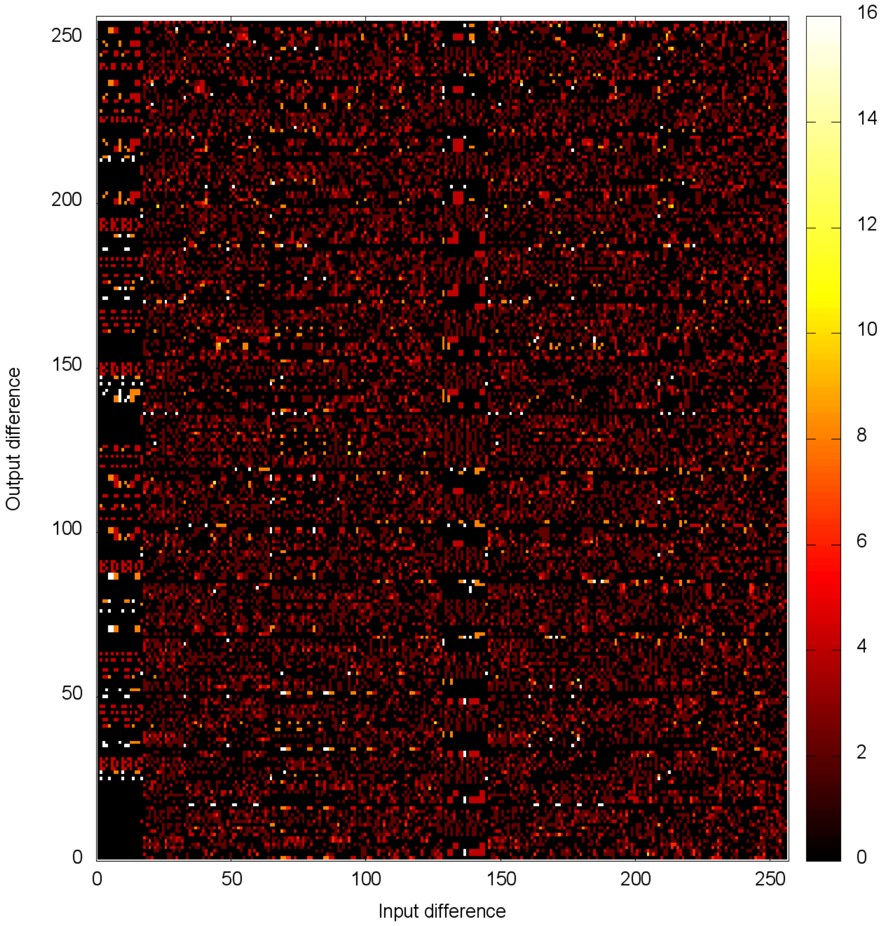


Figure 7.7 – DDT of Littlun-1 .

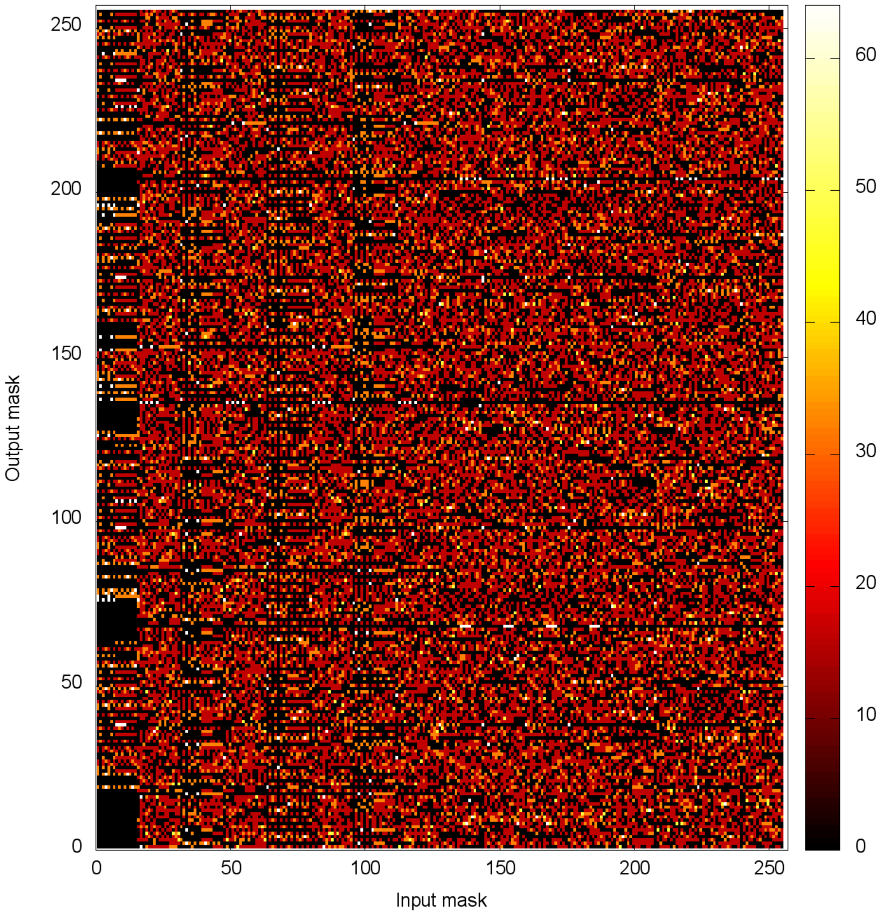


Figure 7.8 – LAT of Littlun-1 .

B Examples of implementation of Littlun-1

B.1 Tables for Littlun-1 and Littlun-S4

```
uint8_t littlun1_s4[16] =
        {0x0, 0xa, 0x4, 0xf, 0xc, 0x7, 0x2, 0x8,
         0xd, 0xe, 0x9, 0xb, 0x5, 0x6, 0x3, 0x1};
```

Figure 7.9 – The 4-bit S-box Littlun-S4 used in Littlun-1 as a C array

```
uint8_t littlun1[256] =
        {0x00, 0x9b, 0xc2, 0x15, 0x5d, 0x84, 0x4c, 0xd1,
         0x67, 0x38, 0xef, 0xb0, 0x7e, 0x2b, 0xf6, 0xa3,
         0xb9, 0xaa, 0x36, 0x78, 0x2f, 0x6e, 0xe3, 0xf7,
         0x12, 0x5c, 0x9a, 0xd4, 0x89, 0xcd, 0x01, 0x45,
         0x2c, 0x63, 0x44, 0xde, 0x02, 0x96, 0x39, 0x70,
         0xba, 0xe4, 0x18, 0x57, 0xa1, 0xf5, 0x8b, 0xce,
         0x51, 0x87, 0xed, 0xff, 0xb5, 0xa8, 0xca, 0x1b,
         0xdf, 0x90, 0x6c, 0x32, 0x46, 0x03, 0x7d, 0x29,
         0xd5, 0xf2, 0x20, 0x5b, 0xcc, 0x31, 0x04, 0xbd,
         0xa6, 0x41, 0x8e, 0x79, 0xea, 0x9f, 0x68, 0x1c,
         0x48, 0xe6, 0x69, 0x8a, 0x13, 0x77, 0x9e, 0xaf,
         0xf3, 0x05, 0xcb, 0x2d, 0xb4, 0xd0, 0x37, 0x52,
         0xc4, 0x3e, 0x93, 0xac, 0x40, 0xe9, 0x22, 0x56,
         0x7b, 0x8d, 0xf1, 0x06, 0x17, 0x62, 0xbf, 0xda,
         0x1d, 0x7f, 0x07, 0xb1, 0xdb, 0xfa, 0x65, 0x88,
         0x2e, 0xc9, 0xa5, 0x43, 0x58, 0x3c, 0xe0, 0x94,
         0x76, 0x21, 0xab, 0xfd, 0x6a, 0x3f, 0xb7, 0xe2,
         0xdd, 0x4f, 0x53, 0x8c, 0xc0, 0x19, 0x95, 0x08,
         0x83, 0xc5, 0x4e, 0x09, 0x14, 0x50, 0xd8, 0x9c,
         0xf4, 0xee, 0x27, 0x61, 0x3b, 0x7a, 0xa2, 0xb6,
         0xfe, 0xa9, 0x81, 0xc6, 0xe8, 0xbc, 0x1f, 0x5a,
         0x35, 0x72, 0x99, 0x0a, 0xd3, 0x47, 0x24, 0x6d,
         0x0b, 0x4d, 0x75, 0x23, 0x97, 0xd2, 0x60, 0x34,
         0xc8, 0x16, 0xa0, 0xbb, 0xfc, 0xe1, 0x5e, 0x8f,
         0xe7, 0x98, 0x1a, 0x64, 0xae, 0x4b, 0x71, 0x85,
         0x0c, 0xb3, 0x3d, 0xcf, 0x55, 0x28, 0xd9, 0xf0,
         0xb2, 0xdc, 0x5f, 0x30, 0xf9, 0x0d, 0x26, 0xc3,
         0x91, 0xa7, 0x74, 0x1e, 0x82, 0x66, 0x4a, 0xeb,
         0x6f, 0x10, 0xb8, 0xd7, 0x86, 0x73, 0xfb, 0x0e,
         0x59, 0x2a, 0x42, 0xe5, 0x9d, 0xa4, 0x33, 0xc7,
         0x3a, 0x54, 0xec, 0x92, 0xc1, 0x25, 0xad, 0x49,
         0x80, 0x6b, 0xd6, 0xf8, 0x0f, 0xbe, 0x7c, 0x11};
```

Figure 7.10 – The Littlun-1 S-box as a `C` array

B.2 SIMD software implementation of Littlun-1

In the context of 4 to 8-bit S-boxes, the Lai-Massey structure of the Littlun construction also allows to conveniently use vector Single Instruction Multiple Data (or SIMD) instructions for efficient implementations. We discuss here an implementation of Littlun-1 based mostly on the `pshufb` instruction from Intel's SSSE3 instruction set. The `pshufb` instruction can easily be used in an implementation either by directly writing the relevant part of the program in assembler or by using compiler intrinsics for a language such as `C`. In the latter case, the intrinsic corresponding to the use of `pshufb` is usually named `_mm_shuffle_epi8`. We give a small function implementing the Littlun-1 S-box using `C` intrinsics in Figure 7.11. Even without further tuning of the code, this function compares favourably with vector implementations of other S-boxes in terms of efficiency. For instance, it needs about half the number of instructions of Hamburg's hand-written vector implementation of the AES S-box [Ham09], although this must be moderated by the fact that the AES S-box is cryptographically stronger.

```
__m128i littlun_ps(__m128i x)
{
  __m128i xlo, xhi, xmid;

  __m128i LO_MASK = _mm_set1_epi8(0x0f);
  __m128i LO_SBOX = _mm_set_epi32(0x01030605, 0x0b090e0d, 0x0802070c,
 ↪  0x0f040a00);
  __m128i HI_SBOX = _mm_set_epi32(0x10306050, 0xb090e0d0, 0x802070c0,
 ↪  0xf040a000);

  xhi  = _mm_srli_epi16(x, 4);
  xhi  = _mm_and_si128(xhi, LO_MASK);
  xlo  = _mm_and_si128(x, LO_MASK);
  xmid = _mm_xor_si128(xlo, xhi);

  xmid = _mm_shuffle_epi8(LO_SBOX, xmid);
  xlo  = _mm_xor_si128(xlo, xmid);
  xhi  = _mm_xor_si128(xhi, xmid);

  xlo = _mm_shuffle_epi8(LO_SBOX, xlo);
  xhi = _mm_shuffle_epi8(HI_SBOX, xhi);
  x   = _mm_xor_si128(xlo, xhi);

  return x;
}
```

Figure 7.11 – Snippet for an SSE `C` implementation of Littlun-1 using compiler intrinsics.

B.3 Hardware circuit for Littlun-S4

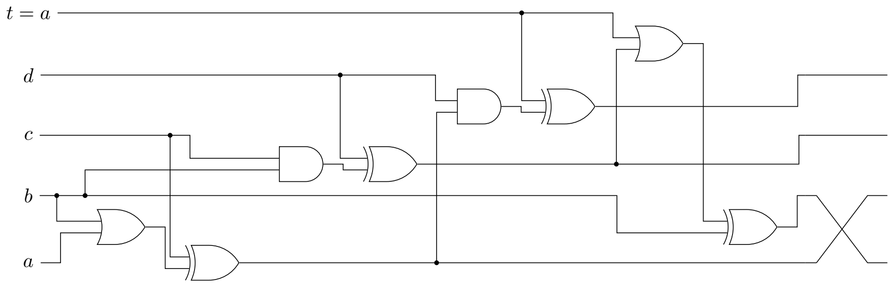


Figure 7.12 – A circuit implementation of Littlun-S4. The symbols ⊃, ⊳ and ⋑ represent the AND, OR and XOR gates respectively.

C AVR implementation of the Fly round function

We give pseudo AVR assembly code for the S-box layer, the permutation and on-the-fly computation of the key-schedule of Fly. All the state, key and temporary variables fit in the 32 registers of an ATtiny or ATmega.

```
; input/output in s0,...,s7 (with MSBs in s0)
; temporary values held in t0,...,t4
; top XOR
movw t0, s0 ; moves t1:s0 <- s1:s0
movw t2, s2
eor t0, s4
eor t1, s5
eor t2, s6
eor t3, s7
; middle S-box
mov t4, t1
or  t1, t0
eor t1, t2
and t2, t4
eor t2, t3
and t3, t1
eor t3, t0
or  t0, t2
eor t0, t4
; bottom XOR
eor s0, t0
eor s1, t1
eor s2, t2
eor s3, t3
eor s4, t0
eor s5, t1
eor s6, t2
eor s7, t3
; bottom S-boxes
mov t0, s1
or  s1, s0
eor s1, s2
and s2, t0
eor s2, s3
and s3, s1
eor s3, s0
or  s0, s2
eor s0, t0

mov t0, s5
or  s5, s4
eor s5, s6
and s6, t0
eor s6, s7
and s7, s5
eor s7, s4
or  s4, s6
eor s4, t0
```

Figure 7.13 – The LITTLUN-1 S-box on ATmega, using 41 instructions.

```
; input/output in s0,...,s7
rol s1
rol s2
rol s2
swap s3
ror s3
swap s4
swap s5
rol s5
ror s6
ror s6
ror s7
```

Figure 7.14 – The ROT permutation on ATmega/ATtiny, using 11 instructions.

```
; key input in k0,...,k15
; cipher state in s0,...,s7
; round constant in c0
; temporary register in t0

; add the current round key & round constant to the state
eor s0, k0
eor s1, k1
eor s2, k2
eor s3, k3
eor s4, k4
eor s5, k5
eor s6, k6
eor s7, k7

eor s0, c0
eor s1, 255

; update k0,...,k7 to the next round key
eor k0, k8
eor k1, k9
eor k2, k10
eor k3, k11
eor k4, k12
eor k5, k13
eor k6, k14
eor k7, k15

; update c0 to the next round constant
mov  t0, c0
andi t0, 1
dec  t0
andi t0, 177
lsr  c0
eor  c0, t0
```

Figure 7.15 – Key addition, and the KS1 key-schedule on ATmega/ATtiny, using 24 instructions.

Taken together, Figure 7.13, Figure 7.14 and Figure 7.15 entirely define the round function of Fly. They thus demonstrate that true to its lightweight nature, Fly can be implemented in a very compact way. This implementation is also easy to obtain from the description of the cipher, and no particular optimisation work is necessary to derive it.

D Hardware implementation of Fly

Fly was not designed to be particularly efficient in hardware, and there are clearly better alternatives in that setting. Thus we did not implement Fly in hardware, but it might be informative to very roughly estimate the cost (in GE) of such an implementation. This can be done by looking at the cost of Present, given the similarity of their structures. A round-based ASIC implementation of Present-128 can be done for 1884 GE [Pos09], of which 27×16 are dedicated to implementing the 16 S-boxes. If we make the assumption that the key-schedule of Fly does not use significantly more area than the one of Present-128, we can estimate that a similar round-based implementation of Fly would cost in the area of $1884 - 27 \times 16 + 80 \times 8 = 2092$ GE, meaning that the overhead is about 11 %.

E Test vectors for Fly

All numbers are given in big endian, *i.e.* those are arrays of bytes, with the byte of lowest address on the left.

```
k0: 0x0000000000000000 k1: 0x0000000000000000
p : 0x0000000000000000
FLY(k0||k1,p)          : 0x40A942D3FB302724
```

```
k0: 0x0001020304050607 k1: 0x08090A0B0C0D0E0F
p : 0xF7E6D5C4B3A29180
FLY(k0||k1,p)          : 0x0D3FE2BF9650AE34
```

```
k0: 0x0000000000000000 k1: 0x0000000000000000
p : 0x0000000000000000
FLY(0,k0)/12           : 0x228F5762975E5B43
FLY(0,k1)/12           : 0x228F5762975E5B43
FLY_RK(k0||k1,p)       : 0x7C5B37DC56F4829A
```

```
k0: 0x0001020304050607 k1: 0x08090A0B0C0D0E0F
p : 0xF7E6D5C4B3A29180
FLY(0,k0)/12           : 0x68F5FC8290A95219
FLY(0,k1)/12           : 0x58F242AC38C00E6B
FLY_RK(k0||k1,p)       : 0x8EE2EA8B0A63DE6D
```

F C masked implementation of Fly

We give here the code of a masked version of the Littlun-1 S-box and of the Fly block cipher in Figure 7.17 and Figure 7.18 respectively. These implementations use masked versions of basic logical operations defined in Figure 7.16. This further shows that Fly can easily be implemented and that its structure does not need to be changed if masking is added.

```
typedef uint8_t bint8_t[NUM_SHARES];

#define NOT(x)       (~x)
#define XOR(x,y)     (x ^ y)
#define AND(x,y)     (x & y)

/* Masked logical NOT */
void bint8_not(bint8_t r, bint8_t a) {
  r[0] = NOT(a[0]);
  // This can be removed if we know that a and r are equal
  memcpy(r+1, a+1, NUM_SHARES - 1);
  return;
}

/* Masked exclusive OR */
void bint8_xor(bint8_t r, bint8_t a, bint8_t b) {
  int i;
  for (i = 0; i < NUM_SHARES; i++) r[i] = XOR(a[i], b[i]);
  return;
}

/* Masked exclusive OR with a public value */
void bint8_xor_pub(bint8_t r, bint8_t a, uint8_t b) {
  r[0] = XOR(a[0], b);
  memcpy(r+1, a+1, NUM_SHARES - 1);
  return;
}

/* Masked logical AND */
void bint8_and(bint8_t rc, bint8_t a,bint8_t b) {
  int i, j;
  uint8_t zij, zji, tmp;
  bint8_t r;

  // Diagonal
  for (i = 0; i < NUM_SHARES; i++) {
    r[i] = AND(a[i],b[i]);
  }
  // Triangles + Row-Wise Sums
  for (i = 0; i < NUM_SHARES; i++) {
    for (j = i + 1; j < NUM_SHARES; j++) {
      zij  = uint8_rand();
      zji  = AND(a[i],b[j]);
      zji  = XOR(zij,zji);
      tmp  = AND(a[j],b[i]);
      zji  = XOR(zji,tmp);
      r[i] = XOR(r[i],zij);
      r[j] = XOR(r[j],zji);
    }
  }

  memcpy(rc,r,NUM_SHARES);

  return;
}

/* Masked rotation */
void bint8_rotl(bint8_t r, bint8_t a, uint8_t b) {
  int i;
  for (i = 0; i < NUM_SHARES; i++) r[i] = ROTL8(a[i], b);
  return;
}

/* Masked logical OR as a series of NOTs and ANDs */
void bint8_or (bint8_t r, bint8_t x, bint8_t y) {
  bint8_t aux, x0, y0;

  bint8_not(x0, x);
  bint8_not(y0, y);
  bint8_and(aux, x0, y0);
  bint8_not(r, aux);
  return;
}

/* The mask-refreshing function  */
void bint8_refresh(bint8_t r, bint8_t a) {
  int i, j;
  uint8_t aux;
  memcpy(r,a,NUM_SHARES);

  for (i = 0; i < NUM_SHARES; i++) {
    for (j = i + 1; j < NUM_SHARES; j++) {
      aux = uint8_rand();
      r[i] = XOR(r[i], aux);
      r[j] = XOR(r[j], aux);
    }
  }

  return;
}
```

Figure 7.16 – Masked implementations of logical operations.

The masked implementations of logical operations defined in the above code can be used to implement masked version of any logical circuit. It can be directly observed from these functions that the cost of a masked XOR is linear in the number of shares NUM_SHARES, as can be seen in bint8_xor, while it is quadratic for the non-linear AND function bint8_and.

The masked implementation of the Littlun S-box shown in the next Figure 7.17 follows in a straightforward way the standard bitsliced implementation of Figure 7.3. The main difference is that the masked versions of Figure 7.16 are used to implement the logical operations and that regular calls to bint8_refresh need to be made to refresh the random masks.

```
void s8 (bint8_t x[8]) {
  bint8_t t, a, b, c, d, aux;

  bint8_xor(a, x[0], x[4]);
  bint8_xor(b, x[1], x[5]);
  bint8_xor(c, x[2], x[6]);
  bint8_xor(d, x[3], x[7]);
  bint8_copy(t, b);
  bint8_refresh(aux, a);
  bint8_or(b, b, aux);
  bint8_xor(b, b, c);
  bint8_refresh(aux, t);
  bint8_and(c, c, aux);
  bint8_xor(c, c, d);
  bint8_refresh(aux, b);
  bint8_and(d, d, aux);
  bint8_xor(d, d, a);
  bint8_refresh(aux, c);
  bint8_or(a, a, aux);
  bint8_xor(a, a, t);
  bint8_xor(x[0], x[0], a);
  bint8_xor(x[1], x[1], b);
  bint8_xor(x[2], x[2], c);
  bint8_xor(x[3], x[3], d);
  bint8_xor(x[4], x[4], a);
  bint8_xor(x[5], x[5], b);
  bint8_xor(x[6], x[6], c);
  bint8_xor(x[7], x[7], d);
  bint8_copy(t, x[1]);
  bint8_refresh(aux, x[0]);
  bint8_or(x[1], x[1], aux);
  bint8_xor(x[1], x[1], x[2]);
  bint8_refresh(aux, t);
  bint8_and(x[2], x[2], aux);
  bint8_xor(x[2], x[2], x[3]);
  bint8_refresh(aux, x[1]);
  bint8_and(x[3], x[3], aux);
  bint8_xor(x[3], x[3], x[0]);
  bint8_refresh(aux, x[2]);
  bint8_or(x[0], x[0], aux);
  bint8_xor(x[0], x[0], t);
  bint8_copy(t, x[5]);
  bint8_refresh(aux, x[4]);
  bint8_or(x[5], x[5], aux);
  bint8_xor(x[5], x[5], x[6]);
  bint8_refresh(aux, t);
  bint8_and(x[6], x[6], aux);
  bint8_xor(x[6], x[6], x[7]);
  bint8_refresh(aux, x[5]);
  bint8_and(x[7], x[7], aux);
  bint8_xor(x[7], x[7], x[4]);
  bint8_refresh(aux, x[6]);
  bint8_or(x[4], x[4], aux);
  bint8_xor(x[4], x[4], t);
  return;
}
```

Figure 7.17 – C masked implementation of the LITTLUN-1 S-box.

The masked implementation of Fly given in Figure 7.18 is a simple implementation of the cipher whose only notable fact is that it uses a masked implementation for the S-box.

```c
void fly (bint8_t x[8], bint8_t k[16]) {
  uint8_t rcon, t, i, j, l;

  rcon = 0;
  for(i = 0; i < 20; i++) {
    for(j = 0; j < 8; j++) {
      bint8_xor(x[j], x[j], k[j]);
    }
    bint8_xor_pub(x[0], x[0], rcon);
    bint8_xor_pub(x[1], x[1], 0xFF);
    s8(x);
    for(l = 1; l < 8; l++) {
      bint8_rotl(x[l], x[l], l);
    }
    t = rcon & 1;
    t = t - 1;
    t = t & 177;
    rcon = rcon >> 1;
    rcon = rcon ^ t;
    for(j = 0; j < 8; j++) {
      bint8_xor(k[j], k[j], k[8 + j]);
    }
  }
  for(j = 0; j < 8; j++) {
    bint8_xor(x[j], x[j], k[j]);
  }
  bint8_xor_pub(x[0], x[0], rcon);
  bint8_xor_pub(x[1], x[1], 0xFF);
  return;
}
```

Figure 7.18 – C masked implementation of Fly.

Part Three

New attacks on SHA-1

Overview

The SHA-1 hash function was one of the first widely-deployed hash functions. It was designed by the NSA in 1995 as a modified version of 1993's SHA-0, which itself was more loosely based on 1990's MD4.

In 2005, Wang *et al.* presented the first major attack on SHA-1, giving a method to find collisions for the hash function in time equivalent to $\approx 2^{69}$ evaluations of the compression function, which is significantly faster than the $\approx 2^{80}$ required by a generic process. However, the high cost of the attack implied that no explicit collision was computed at the time. This remains the case today, even if improvements to the attack and ever more efficient hardware would make such an endeavour more affordable than it was in 2005. The considerable amount of work that followed Wang *et al.*'s original contribution then mostly focused on first getting a better understanding of the attack process, and secondly on bringing innovations allowing to explicitly provide collisions for reduced versions of SHA-1 with the highest number of steps.

Considering the efficient collision attacks developed since 2005, it may seem remarkable that SHA-1 in fact still offers a comfortable security margin against first preimage attacks. Until our own work, the attack reaching the highest number of steps was due to Knellwolf and Khovratovich, who showed in 2012 how to attack 57/80 of the function, with an estimated complexity of $2^{158.8}$ evaluations of the compression function.

The third and last part of this thesis describes new attacks on SHA-1.

We start by presenting explicit collision attacks for the full compression function of SHA-1. While this comes short of a practical attack on the *hash function*, this is the first explicit result on SHA-1 for a standard security notion. Furthermore, the efficient implementation framework that was developed for this work could also be reused for hash function attacks, which have a very similar and in some respect simpler structure than our freestart case.

Collision attacks on SHA-1 *à la* Wang have a rather specific structure, full of small technicalities. Thus, after briefly introducing SHA-1 in Chapter 8, we start with a presentation of this topic in Chapter 9, that we hope to be self-contained. Our contributions on freestart collisions are presented next in Chapter 10.

We then conclude this thesis by presenting in Chapter 11 the current best preimage attacks on SHA-1, in terms of number of attacked steps. By extending the framework developed by Knellwolf and Khovratovich to higher-order differential attacks, we are able to increase by five the number of steps of SHA-1 for which a preimage can be computed faster than with a generic method.

Contents

8 The SHA-1 hash function

9 A brief history of collision attacks on SHA-1

10 Freestart collision attacks for SHA-1

11 Preimage attacks for SHA-1

Chapter 8

The SHA-1 hash function

1 Notation

We start by introducing some notation that is used through this entire part. Table 8.1 gives the meaning of various standard symbols and Table 8.2 shows the signification of the symbols used to denote bit differences between two states. Additionally, we use the following conventions: SHA-1 states, messages, and expanded messages are respectively denoted by $\mathcal{A}$, $\mathcal{M}$, $\mathcal{W}$; for a variable x, the corresponding variable related by a specific difference is noted $\widetilde{x}$, the difference itself is denoted by $\Delta(x,\widetilde{x})$ or $\Delta\, x$; two different variables of the same type are noted x, x' when their difference is not known; a variable that can be seen as an array can have its words accessed through a subscript, indices starting from zero, *e.g.* x_2 is the third word of x; a variable that can be seen as a fixed-size binary word can have its bits accessed through a bracket notation, indices starting from zero, *e.g.* $x[31]$ is the thirty-second bit of x; numbers written in hexadecimal use a fixed-space font and the *0x* prefix, *e.g.* `0x1337`; we sometimes also write numbers in base two with a subscript 2, *e.g.* 1010_2, in the case where the basis cannot immediately be inferred. Finally, ":=" is used to denote equality by definition. Various additional shorthands are introduced throughout the text.

2 The SHA-1 hash function

This section gives a brief description of the SHA-1 hash function, going again over some features already presented in Chapter 4. We refer to the most recent NIST standard document [NIS15a] for a more thorough presentation.

SHA-1 is a hash function from the MD-SHA family which produces digests of 160 bits. Its high-level structure follows the Merkle-Damgård framework [Mer89, Dam89]: the input message to the function is first padded to a length multiple of the block size, which is 512 bits, defining k similarly-sized blocks. Each block m_i is then fed to a compression function H which is used to update a 160-bit chaining value c_i: $\mathsf{c}_{i+1} := \mathrm{H}(\mathsf{c}_i, \mathsf{m}_i)$. The first chaining value c_0 is a predefined constant set to an initial value

Table 8.1 – Meaning of standard symbols.

Symbol	Meaning
$\oplus$	Bitwise exclusive or
$+$	Modular addition
$\boxplus$	Modular addition, word-wise modular addition
$-$	Modular subtraction
$\vee$	Bitwise logical or
$\wedge$	Bitwise logical and
$\neg$	Bitwise complementation
$\circlearrowleft r$	Bit rotation by r to the left
$\circlearrowright r$	Bit rotation by r to the right

Table 8.2 – Meaning of the bit difference symbols, for a symbol located on $\mathcal{A}_t[i]$. The same symbols are also used for $\mathcal{W}$.

Symbol	Condition on $(\mathcal{A},\widetilde{\mathcal{A}})$
○	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i]$
●	$\mathcal{A}_t[i] \neq \widetilde{\mathcal{A}}_t[i]$
▲	$\mathcal{A}_t[i] = 0, \quad \widetilde{\mathcal{A}}_t[i] = 1$
▼	$\mathcal{A}_t[i] = 1, \quad \widetilde{\mathcal{A}}_t[i] = 0$
▽	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] = 0$
△	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] = 1$
☆	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] = A_{t-1}[i]$
★	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] \neq A_{t-1}[i]$
◇	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] = (A_{t-1} \circlearrowright 2)[i]$
◆	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] \neq (A_{t-1} \circlearrowright 2)[i]$
□	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] = (A_{t-2} \circlearrowright 2)[i]$
■	$\mathcal{A}_t[i] = \widetilde{\mathcal{A}}_t[i] \neq (A_{t-2} \circlearrowright 2)[i]$
∗	No condition on $\mathcal{A}_t[i]$, $\widetilde{\mathcal{A}}_t[i]$

IV given in the specifications of the function, and the last value c_k is the final output of the hash function. The padding rule of SHA-1 is a straightforward application of Merkle-Damgård strengthening; it is at least 65-bit long and is made of one “1” bit followed by a possibly zero number of “0” bits, and the length of the message to be hashed (without the padding) in bits as a 64-bit integer. The value of the IV is given in Table 8.3 as five 32-bit words; each of these words initialises one of the five internal registers of the compression function, described below. Note that neither the padding nor the IV actually play any role in our attacks.

Similarly to other members of the MD-SHA family, the compression function H

Table 8.3 – The initial value (IV) of SHA-1.

$\mathcal{A}_0$:`0x67452301`	$\mathcal{B}_0$:`0xefcfab89`	$\mathcal{C}_0$:`0x98badcfe`	$\mathcal{D}_0$:`0x10325476`	$\mathcal{E}_0$:`0xc3d2e1f0`

is built around an *ad hoc* block cipher E used in a Davies-Meyer construction. The block cipher itself is a five-branch generalised Feistel network using an Add-Rotate-XOR "ARX" step function with the addition of two non-linear Boolean functions, see Table 8.4. In its full version, the step function is iterated 80 times, divided in four rounds of 20 steps.

The internal state of E consists of five 32-bit registers $(\mathcal{A}_i, \mathcal{B}_i, \mathcal{C}_i, \mathcal{D}_i, \mathcal{E}_i)$; at each step, a 32-bit *expanded* message word $\mathcal{W}_i$ derived from a message block m is used to update the five registers:

$$\begin{cases} \mathcal{A}_{i+1} & = & (\mathcal{A}_i \circlearrowleft 5) + \varphi_i(\mathcal{B}_i, \mathcal{C}_i, \mathcal{D}_i) + \mathcal{E}_i + \mathcal{K}_i + \mathcal{W}_i \\ \mathcal{B}_{i+1} & = & \mathcal{A}_i \\ \mathcal{C}_{i+1} & = & \mathcal{B}_i \circlearrowright 2 \\ \mathcal{D}_{i+1} & = & \mathcal{C}_i \\ \mathcal{E}_{i+1} & = & \mathcal{D}_i \end{cases},$$

with $\mathcal{K}_i$ a constant and φ_i one of three possible bitwise Boolean functions, see Table 8.4 for their specifications. We give a graphical representation of this step function in Figure 8.1. From this figure and the definition of the function, one can notice that all updated registers except $\mathcal{A}_{i+1}$ are just rotated copies of another; thus it is possible to equivalently express SHA-1's step function in a recursive way, using only past values of the register $\mathcal{A}$. The definition then becomes:

$$\mathcal{A}_{i+1} = (\mathcal{A}_i \circlearrowleft 5) + \varphi_i(\mathcal{A}_{i-1}, \mathcal{A}_{i-2} \circlearrowright 2, \mathcal{A}_{i-3} \circlearrowright 2) + (\mathcal{A}_{i-4} \circlearrowright 2) + \mathcal{K}_i + \mathcal{W}_i. \tag{8.1}$$

With this notation, the output of one application of the compression function is made of the possibly rotated last five state words, $\mathcal{A}_{76} \ldots \mathcal{A}_{80}$.

Table 8.4 – Boolean functions and constants of SHA-1.

round	step i	$\varphi_i(x, y, z)$	$\mathcal{K}_i$
1	$0 \leq i < 20$	$\varphi_{\text{IF}} = (x \wedge y) \vee (\neg x \wedge z)$	`0x5a827999`
2	$20 \leq i < 40$	$\varphi_{\text{XOR}} = x \oplus y \oplus z$	`0x6ed6eba1`
3	$40 \leq i < 60$	$\varphi_{\text{MAJ}} = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$	`0x8fabbcdc`
4	$60 \leq i < 80$	$\varphi_{\text{XOR}} = x \oplus y \oplus z$	`0xca62c1d6`

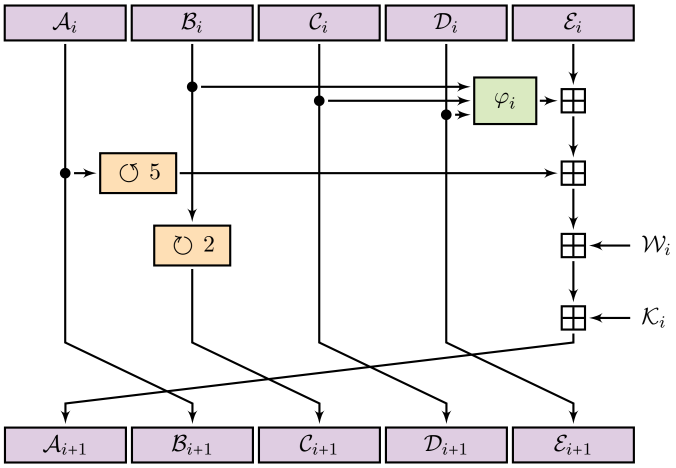


Figure 8.1 – One step of SHA-1. Figure adapted from [Jea].

Finally, the expanded message words $\mathcal{W}_i$ are computed from the 512-bit message block m. This message is first expressed as sixteen 32-bit words $\mathcal{M}_0, \ldots, \mathcal{M}_{15}$, which are then used to recursively define the eighty 32-bit words $\mathcal{W}_i$:

$$\mathcal{W}_i = \begin{cases} \mathcal{M}_i, & \text{for } 0 \le i \le 15 \\ (\mathcal{W}_{i-3} \oplus \mathcal{W}_{i-8} \oplus \mathcal{W}_{i-14} \oplus \mathcal{W}_{i-16}) \circlearrowleft 1, & \text{for } 16 \le i \le 79 \end{cases} . \tag{8.2}$$

The step function and the message expansion can both easily be inverted as follows:

$$\mathcal{A}_i = (\mathcal{A}_{i+5} - \mathcal{W}_{i+4} - \mathcal{K}_{i+4} - \varphi_{i+4}(\mathcal{A}_{i+3}, \mathcal{A}_{i+2} \circlearrowright 2, \mathcal{A}_{i+1} \circlearrowright 2) - (\mathcal{A}_{i+4} \circlearrowleft 5)) \circlearrowleft 2, \tag{8.3}$$

$$\mathcal{W}_i = (\mathcal{W}_{i+16} \circlearrowright 1) \oplus \mathcal{W}_{i+13} \oplus \mathcal{W}_{i+8} \oplus \mathcal{W}_{i+2}. \tag{8.4}$$

Chapter 9

A brief history of collision attacks on SHA-1

In this chapter we give a background on the literature of collision attacks on SHA-1, that was initiated by the major work of Wang, Yin and Yu from CRYPTO 2005 [WYY05a]. Some of the techniques used to attack SHA-1 had been previously introduced to attack other hash functions, and in particular SHA-0, SHA-1's close predecessor. Consequently, we start by discussing some of these earlier work. We then review some of the more recent developments of the original attacks.

None of the material presented in this chapter is new, and it may be safely skipped by an experienced reader. We believe however that it may be of some use to a public less familiar with the matter.

All the attacks presented in this chapter are differential in nature. In its simplest form, the idea of such attacks is to find a good *message difference* $\Delta(\mathcal{W},\widetilde{\mathcal{W}})$ (or $\Delta\,\mathcal{W}$) and associated state differential path $\Delta(\mathcal{A},\widetilde{\mathcal{A}})$ (or $\Delta\,\mathcal{A}$) such that a pair of states following the differential path $\Delta\,\mathcal{A}$ results in a collision, and finding a pair of messages of difference $\Delta\,\mathcal{W}$ such that the corresponding pair of states follows $\Delta\,\mathcal{A}$ is efficient; in particular, this means that searching for a collision in that way is faster than searching for one by brute force. In the case of SHA-1, the difference is actually imposed on the non-expanded message words $\mathcal{M}_i$. However, the message expansion being linear, we will usually rather consider the implied probability-one difference on the expanded message $\mathcal{W}_i$.

1 Preliminaries: collision attacks on SHA-0

In the initial SHA standard of 1993, there was no rotation by one to the left in the message expansion of the compression function [NIS93]; the corresponding original algorithm was retrospectively named SHA-0, to distinguish it from the updated standard SHA-1, first published in 1995 [NIS95]. At CRYPTO 1998, Chabaud and Joux presented a theoretical collision attack on SHA-0, with an estimated complexity of searching through 2^{61} message pairs [CJ98], equivalent to 2^{58} calls to the compression function of

SHA-0 [BCJ15]. However, the modified message expansion of SHA-1 prevents a straightforward application of the same approach from leading to an attack better than brute force.

There are four main components in the original attack on SHA-0 and its first improvements [BC04, BCJ⁺05, BCJ15], all of which found their way to the attacks on SHA-1, either *as is* or in a modified form:

1. *Local collisions* in the step function as a springboard for a collision for the full function (Section 2).

2. Using *signed differences* and a fine analysis of difference conditions in the Boolean functions (Section 3).

3. Regrouping local collisions along a *disturbance vector* (Section 4).

4. Efficient implementation of the probabilistic search for colliding messages (Section 5).

We will only briefly mention the multi-block techniques as used for SHA-0 [BCJ⁺05, BCJ15], as these are not directly relevant to the best attacks on SHA-1. The improvements by Wang *et al.* used to attack the full SHA-1 are presented next in Section 6.

2 Local collisions for a few steps of SHA

An instructive starting point when searching for collisions on SHA is to first consider a *linearised* variant (over $\mathbf{F}_2$) of the step function, obtained by replacing the Boolean functions φ_{IF} and φ_{MAJ} by φ_{XOR} and the additions in $\mathbf{Z}/2^{32}\mathbf{Z}$ by additions in $\mathbf{F}_2^{32}$,*i.e.* XORs. Although collisions for this simple variant named SHI-1 by Chabaud and Joux [CJ98] are trivial to find, SHI-1 is useful as a simple model to build the main structure of the attack, in particular the differential paths $\Delta\mathcal{W}$ and $\Delta\mathcal{A}$. A main element at the basis of this paths is the concept of *local collisions* for the step function.

It is easy to see for instance from Equation 8.1 that only five consecutive state words are used in the step function of SHA to determine the value of the next or previous word. Consequently, if we could introduce a difference in the message, such that after some steps the five state words of $\mathcal{A}$ and $\widetilde{\mathcal{A}}$ were equal, then the final hash values for the two computations would form a collision, as long as no more differences were present in the remainder of the message. Of course, the latter condition is hard to meet in general, but meeting the former seems to be quite easy as long as some limited control on the message words is available. It is also a good first objective, as it locally achieves the result that we wish to get at the end of the computation.

Let us assume that we have full control over six consecutive expanded message words: $\mathcal{W}_{i\ldots i+5}$ and $\widetilde{\mathcal{W}}_{i\ldots i+5}$ used in the computation of two related SHA states $\mathcal{A}$, $\widetilde{\mathcal{A}}$ and that we have $\mathcal{W}_j = \widetilde{\mathcal{W}}_j$ for $j < i$.

The first step of a local collision is to introduce a difference, for instance in one bit, so that the two messages are not consistently equal, as we are not interested in any trivial equality. For the linear variant SHI-1, the exact index where this difference is introduced does not matter much; let us assume w.l.o.g. that $\mathcal{W}_i$ and $\widetilde{\mathcal{W}_i}$ are different exactly on bit 7, *i.e.*

$$\Delta(\mathcal{W}_i, \widetilde{\mathcal{W}_i}) = \text{○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○○} \quad .$$

From Equation 8.1, we see that this introduces a difference between $\mathcal{A}_{i+1}$ and $\widetilde{\mathcal{A}}_{i+1}$ in the same position, *i.e.* on bit 7. Our goal is now to ensure that this difference does not propagate further, and that there are no differences between $\mathcal{A}_{i+2\ldots i+6}$ and $\widetilde{\mathcal{A}}_{i+2\ldots i+6}$:

- At $(\mathcal{A}_{i+2}, \widetilde{\mathcal{A}}_{i+2})$, we must cancel the difference coming from $(\mathcal{A}_{(i+2)-1}, \widetilde{\mathcal{A}}_{(i+2)-1}) \circlearrowleft 5 = (\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1}) \circlearrowleft 5$; this is done by inserting a difference in bit 12 of $\mathcal{W}_{i+1}$:

$$\Delta(\mathcal{W}_{i+1}, \widetilde{\mathcal{W}}_{i+1}) = \text{○○○○○○○○○○○○○○○○○○○●○○○○○○○○○○○○} \quad .$$

- At $(\mathcal{A}_{i+3}, \widetilde{\mathcal{A}}_{i+3})$, we must cancel the difference coming from $(\mathcal{A}_{(i+3)-2}, \widetilde{\mathcal{A}}_{(i+3)-2}) = (\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1})$; this is done by inserting a difference in bit 7 of $\mathcal{W}_{i+2}$:

$$\Delta(\mathcal{W}_{i+2}, \widetilde{\mathcal{W}}_{i+2}) = \text{○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○○} \quad .$$

- At $(\mathcal{A}_{i+4}, \widetilde{\mathcal{A}}_{i+4})$, we must cancel the difference coming from $(\mathcal{A}_{(i+4)-3}, \widetilde{\mathcal{A}}_{(i+4)-3}) \circlearrowright 2 = (\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1}) \circlearrowright 2$; this is done by inserting a difference in bit 5 of $\mathcal{W}_{i+3}$:

$$\Delta(\mathcal{W}_{i+3}, \widetilde{\mathcal{W}}_{i+3}) = \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \quad .$$

- At $(\mathcal{A}_{i+5}, \widetilde{\mathcal{A}}_{i+5})$, we must cancel the difference coming from $(\mathcal{A}_{(i+5)-4}, \widetilde{\mathcal{A}}_{(i+5)-4}) \circlearrowright 2 = (\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1}) \circlearrowright 2$; this is done by inserting a difference in bit 5 of $\mathcal{W}_{i+4}$:

$$\Delta(\mathcal{W}_{i+4}, \widetilde{\mathcal{W}}_{i+4}) = \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \quad .$$

- At $(\mathcal{A}_{i+6}, \widetilde{\mathcal{A}}_{i+6})$, we must cancel the difference coming from $(\mathcal{A}_{(i+6)-5}, \widetilde{\mathcal{A}}_{(i+6)-5}) \circlearrowright 2 = (\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1}) \circlearrowright 2$; this is done by inserting a difference in bit 5 of $\mathcal{W}_{i+5}$:

$$\Delta(\mathcal{W}_{i+5}, \widetilde{\mathcal{W}}_{i+5}) = \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \quad .$$

At this point, we have reached our goal of having no differences in a pair of five consecutive state words.

The pattern formed by the successive message differences of a local collision is commonly seen in various stages of attacks on SHA, and as such it deserves to be shown in its entirety:

$$\Delta(\mathcal{W}_{i\dots i+5}, \widetilde{\mathcal{W}}_{i\dots i+5}) = \begin{array}{l} \text{○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○○} \\ \text{○○○○○○○○○○○○○○○○○○○●○○○○○○○○○○○○} \\ \text{○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○○} \\ \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \\ \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \\ \text{○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○} \end{array}.$$

In the case of SHI-1, the probability of obtaining a local collision when following the above pattern is equal to one. However, this is not the case anymore when the true SHA step function is used. For instance, there is a probability 2^{-1} that the introduction of the difference in $(\mathcal{A}_{i+1}, \widetilde{\mathcal{A}}_{i+1})$ with modular addition rather than XOR leads to a difference in more than one bit because of different behaviours in the propagation of the carry in the two states, on which we do not assume to have any direct control. Overall, the probability of obtaining a successful local collision depends on several factors, including which Boolean function is used and whether several collisions are chained together or not. We partially address this matter next.

3 Difference analysis for impure ARX

We now move away from SHI-1 and start to analyse the behaviour of local collisions for the true SHA function. There are two main points in this analysis: what conditions *at the bit level* ensure the highest probability of success for the different types of single local collisions, and under optimal conditions, what is the probability of a chain of local collisions? We focus on the first question here, and defer the answer to the second to Section 9.

In the case of SHA, and more generally ARX primitives, the way of expressing differences between messages is less obvious than for *e.g.* bit or byte-oriented primitives. It is indeed natural to consider both "XOR differences" over $\mathbf{F}_2^{32}$ and "modular differences" over $\mathbf{Z}/2^{32}\mathbf{Z}$, as both operations are used in the function. In practice, the literature on SHA uses several hybrid representations of differences based on *signed XOR differences*. In its most basic form, such a difference is similar to an XOR difference with the additional information of the value of the differing bits and of bits equal to each other, which is a "sign" for the difference.

This is an important information when one works with modular addition as the sign impacts the propagation of carries in the addition of two differences. Let us for instance consider the two pairs of words $a = 11011000001_2$, $\widetilde{a} = 11011000000_2$ and $b = 10100111000_2$, $\widetilde{b} = 10100111001_2$; the XOR differences $(a \oplus \widetilde{a})$ and $(b \oplus \widetilde{b})$ are both 00000000001_2, *i.e.* ○○○○○○○○○○●, meaning that $(a \oplus b) = (\widetilde{a} \oplus \widetilde{b})$. On the other hand, the signed XOR difference between a and $\widetilde{a}$ may be written ○○○○○○○○○○▼ to convey the fact that they are different on their lowest bit *and* that the value of this bit is 1 for a and thence 0 for $\widetilde{a}$, *i.e.* $\widetilde{a} = a - 1$; similarly, the signed difference between b and $\widetilde{b}$ may be written ○○○○○○○○○○▲, which is a difference in the same position but of a different sign, *i.e.* $\widetilde{b} = b + 1$. From these differences, we can deduce that $(a + b) = (\widetilde{a} + \widetilde{b})$ because differences of different signs cancel while differences of the same sign do not; if we were to swap the values b and $\widetilde{b}$, both differences on a and b would have the same sign and indeed we have $(a + \widetilde{b}) \neq (\widetilde{a} + b)$ though $(a \oplus \widetilde{b})$ and $(\widetilde{a} \oplus b)$ remain equal. In the case of bits with no difference, we may similarly want to use different notations to express the fact that two bits are equal to zero ($\triangledown$) or equal to one ($\vartriangle$).

It is possible to extend signed differences to account for more generic combinations of possible values for each message bit; this was for instance done by De Cannière and Rechberger to aid in the automatic search of differential paths [DR06]. Another possible extension is to consider relations between various bits of different possibly rotated state words; this allows to efficiently keep track of the propagation of differences through the step function. Such differences are for instance used by Stevens [Ste13], and also later in this work.

Using signed differences, we can express a first simple necessary condition for a local collision to happen: because the initial difference in $\Delta\mathcal{A}_{i+1}$ has to be canceled in $\Delta\mathcal{A}_{i+2}$ through the modular addition of $\Delta\mathcal{W}_{i+1}$, we know that these differences have to be of different sign. As we have some control on the message, we can ensure that this is always the case for successful introductions of the difference on $\Delta\mathcal{A}_{i+1}$, *i.e.* ones that do not result in different carry propagations for $\mathcal{A}_{i+1}$ and $\widetilde{\mathcal{A}}_{i+1}$, by analysing what may happen in the four possible cases of an introduction (depending on the signs of the involved differences), at the level of one bit:

1. $\blacktriangle$ (the difference in $\Delta\mathcal{W}_i$, introducing the perturbation) + $\triangledown$ (the "difference" in the same position of the partial sum used to compute $\Delta\mathcal{A}_{i+1}$) = $\blacktriangle$, with no different carry propagation in the remainder of $\Delta\mathcal{A}_{i+1}$).

2. $\blacktriangle + \vartriangle = \blacktriangledown$, with different carry propagation.

3. $\blacktriangledown + \triangledown\!= \blacktriangledown$, with no different carry propagation.

4. $\blacktriangledown + \vartriangle = \blacktriangle$, with different carry propagation.

Of these four cases, (1) and (3) are always favourable to a local collision; (2) and (4) are not considered to be favourable here, except if the differences are on the most significant bit of the message and state words, as in this case the carry propagation is absorbed by the modular reduction; in that unique case, unsigned differences may be used safely. We will also later see in Section 9 that in some cases, a difference in the carry propagation does not actually always result in the absence of a local collision. We ignore this fact in the current analysis. Thus, except when the difference is introduced on the MSB, we see that a successful introduction on $\Delta\mathcal{A}_{i+1}$ preserves the sign of the difference $\Delta\mathcal{W}_i$, and thence we must always choose a different sign for the difference on $\Delta\mathcal{W}_{i+1}$ if we want the two to cancel.

We can analyse the rest of the conditions for a successful local collision in a similar fashion; it is helpful at this point to consider the possible behaviours of the Boolean functions of SHA for their different signed inputs. We follow here the approach by

Joux [Jou09, Chapter 5] and start by analysing the propagation of signed differences ▿, ▵, ▾, ▴ through φ_{IF} (Table 9.1), φ_{XOR} (Table 9.2) and φ_{MAJ} (Table 9.2), before considering what happens for each remaining correction of the local collision in $\Delta\mathcal{A}_{i+3\ldots i+6}$. An essentially identical analysis can for instance be found in [Pey08, BCJ15].

We should note that for reasons that will be made clear in Section 4, the corrections used to obtain a local collision must work with every possible Boolean function[1]. A consequence is that we cannot use the fact that the non-linear functions φ_{IF} and φ_{MAJ} may absorb a single difference, for instance as in φ_{IF}(▿, ▴, ▿), as there is never such a behaviour with φ_{XOR}; thus, there is always a correction introduced in every message, true to the local collision pattern of SHI-1 of Section 2. In general, the following things may then happen in the Boolean functions: the difference is absorbed (this never happens with φ_{XOR}); the difference is preserved, but its sign is changed (this never happens with φ_{MAJ}); the difference is preserved and its sign is not changed. As we already mentioned, knowing the sign of a difference is important if we want to cancel it with another one; thus, the possibility of unpredictable changes of signs decreases the success probability of a local collision. Let us see in general how this latter is impacted by the behaviours of the different Boolean functions.

- $\Delta\mathcal{A}_{i+3}$: the difference is on the first input (x) of the Boolean function. With φ_{IF}, there is a probability 2^{-1} that it is absorbed and 2^{-2} that the sign is changed otherwise, for a total success probability of 2^{-2} (2^{-1} on the MSB, where a change of sign has no consequence). With φ_{XOR}, there is a probability 2^{-1} that the sign is changed (total success of 2^{-1}, 1 on the MSB). With φ_{MAJ}, there is a probability 2^{-1} that it is absorbed (total success of 2^{-1}).

- $\Delta\mathcal{A}_{i+4}$: the difference is on the second input (y) of the Boolean function. With φ_{IF}, there is a probability 2^{-1} that it is absorbed (total success of 2^{-1}). With φ_{XOR} there is a probability 2^{-1} that the sign is changed (total success of 2^{-1}, 1 on the MSB). With φ_{MAJ} there is a probability 2^{-1} that it is absorbed (total success of 2^{-1}).

- $\Delta\mathcal{A}_{i+5}$: the difference is on the third input (z) of the Boolean function. This case is the same as for $\Delta\mathcal{A}i + 4$.

- $\Delta\mathcal{A}_{i+6}$: the difference is on a modular addition. If it is on the MSB, nothing needs to be done. Otherwise, it needs to be of sign opposite the one of the introductory difference to ensure a correction, which will then happen with probability 1. This condition can be enforced for free by properly choosing the signed message difference $\Delta\mathcal{W}$.

This analysis may be useful in several ways. First, it gives some necessary conditions on the signs of the corrections, possibly conditioned on the Boolean function of the round being considered; more generally, it may actually be used as a nearly exhaustive list of inputs resulting in local collisions, but this is less useful as we do not have control

[1] This is not true if our goal is to build boomerangs, which is something that we will consider in Section 5.

on the values of $\Delta\mathcal{A}$ in general. Second, it helps us to position the local collisions in an “optimal” way, by ensuring that many corrections involve bits that are at the most significant position, as we have seen that these may have higher success probabilities. Finally, it may be used to check in advance if a given local collision is going to happen or not, without necessarily computing the two states $\mathcal{A}$ and $\widetilde{\mathcal{A}}$. For instance, in the first round, assuming a perturbation on bit j of $\Delta\mathcal{A}_{i+1}$, we can predict that there will be no local collision if the bits $j-2$ of $\Delta\mathcal{A}_i$ and $\Delta\mathcal{A}_{i-1}$ are equal, even if the perturbation is properly introduced. Indeed, none of $\varphi_{\mathrm{IF}}(\blacktriangle,\triangledown,\triangledown)$, $\varphi_{\mathrm{IF}}(\blacktriangle,\vartriangle,\vartriangle)$, $\varphi_{\mathrm{IF}}(\blacktriangledown,\triangledown,\triangledown)$, $\varphi_{\mathrm{IF}}(\blacktriangledown,\vartriangle,\vartriangle)$ results in a difference, which is a direct consequence of the very definition of the IF function. This means that there will be no difference in the Boolean function computation at $\Delta\mathcal{A}_{i+3}$ and thus the tentative correction in $\Delta\mathcal{W}_{i+2}$ will actually introduce a difference.

Table 9.1 – Signed difference analysis of $\varphi_{\mathrm{IF}}(x,y,z)$.

$x =$ ▽	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ △	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▽	▽	▽	▽	$z =$ ▽	▽	△	▲	▼
$z =$ △	△	△	△	△	$z =$ △	▽	△	▲	▼
$z =$ ▲	▲	▲	▲	▲	$z =$ ▲	▽	△	▲	▼
$z =$ ▼	▼	▼	▼	▼	$z =$ ▼	▽	△	▲	▼
$x =$ ▲	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ ▼	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▽	▲	▲	▽	$z =$ ▽	▽	▼	▽	▼
$z =$ △	▼	△	△	▼	$z =$ △	▲	△	▲	△
$z =$ ▲	▽	▲	▲	▽	$z =$ ▲	▲	△	▲	△
$z =$ ▼	▼	△	△	▼	$z =$ ▼	▽	▼	▽	▼

4 Disturbance vectors

We have seen how using a local collision allows one to create a pair of different messages which may lead to a pair of SHA states that are identical at some point during the computation of their respective digests. Given enough control on the message, which implies control on the state, one may obtain such an equality with probability one, and it is quite easy to derive sufficient conditions for a single local collision to happen. This makes it possible to derive the success probability of uncontrolled independent local collisions.

In order to mount a complete attack and obtain a collision on the actual digest, we need to ensure that the last five state words are free of differences: this means that no local collision must have been started after step 75. Of course, we also need the colliding

Table 9.2 – Signed difference analysis of $\varphi_{\mathrm{XOR}}(x, y, z)$.

$x =$ ▽	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ △	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▽	△	▲	▼	$z =$ ▽	△	▽	▼	▲
$z =$ △	△	▽	▲	▼	$z =$ △	▽	△	▼	▲
$z =$ ▲	▲	▼	▽	△	$z =$ ▲	▼	▲	△	▽
$z =$ ▼	▼	▲	△	▽	$z =$ ▼	▲	▼	▽	△

$x =$ ▲	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ ▼	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▲	▼	▽	△	$z =$ ▽	▼	▲	△	▽
$z =$ △	▼	▲	△	▽	$z =$ △	▲	▼	▽	△
$z =$ ▲	▽	△	▲	▼	$z =$ ▲	△	▽	▼	▲
$z =$ ▼	△	▽	▼	▲	$z =$ ▼	▽	△	▲	▼

Table 9.3 – Signed difference analysis of $\varphi_{\mathrm{MAJ}}(x, y, z)$.

$x =$ ▽	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ △	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▽	▽	▽	▽	$z =$ ▽	▽	△	▲	▼
$z =$ △	▽	△	▲	▼	$z =$ △	△	△	△	△
$z =$ ▲	▽	▲	▲	▽	$z =$ ▲	▲	△	▲	△
$z =$ ▼	▽	▼	▽	▼	$z =$ ▼	▼	△	△	▼

$x =$ ▲	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼	$x =$ ▼	$y =$ ▽	$y =$ △	$y =$ ▲	$y =$ ▼
$z =$ ▽	▽	▲	▲	▽	$z =$ ▽	▽	▼	▽	▼
$z =$ △	▲	△	▲	△	$z =$ △	▼	△	△	▼
$z =$ ▲	▲	▲	▲	▲	$z =$ ▲	▽	△	▲	▼
$z =$ ▼	▽	△	▲	▼	$z =$ ▼	▼	▼	▼	▼

messages to be valid expanded messages, and this is where local collisions become really useful: the idea is to interleave a series of local collisions together so that the resulting message difference follows the message expansion and the joint probability of all the local collisions being successful is high; in particular, an attack is obtained if it is higher than $\approx 2^{-80}$.

The first condition is actually easy to meet by defining a *disturbance vector* (DV). This is a vector of eighty 32-bit words, whose "1" bits define the positions of the initial perturbations of the series of local collisions. Now it suffices to remark that because the message expansion is linear, a sum of expanded messages is an expanded message itself; then if the disturbance vector is a valid expanded message word, so is the complete message difference, including the corrections of the local collisions. For this to be correct, however, two conditions need to be met: first, the disturbance vector must have no differences in its first five "negative" words $\mathcal{W}_{-5,\ldots,-1}$, obtained through the backward message expansion. Indeed, remembering Section 2, the corrections of the local collisions will be obtained from possibly rotated shifted copies of the pattern of initial perturbations introduced by the DV, up to five positions down. If we want these copies to be valid expanded message words, they must be equal to the DV with its last (up to five) words removed and with a few (up to five) words added at the beginning, which would be obtained from the backward message expansion. As the added words must be zero, lest they create new perturbations that would not be corrected, we obtain the aforementioned condition. Second, all corrections need to be performed in the same way, so that they globally conform to the message expansion; this is why one cannot exploit the absorption properties of some Boolean functions as noticed in Section 3, as it is impossible to do so consistently across all the rounds.

This simple characterisation of series of local collisions allows to define efficient search strategies to find vectors that achieve a high joint probability. In the case of SHA-0, the message expansion can easily be defined at the bit level, as the bit i of an expanded message word $\mathcal{W}_j$, $j > 15$ is entirely determined by the bit i of the message words $\mathcal{W}_{j-16\ldots j-1}$. This means that any expanded message, and in particular a disturbance vector, can be seen as the union of 32 "one-bit expanded messages" that do not interact with each other. Thus, a search for good disturbance vectors can focus solely on such one-bit messages. As any consecutive window of sixteen message words entirely defines the remainder of an expanded message word through the (backward) message expansion, one can see that there is only a small number of 2^{16} one-bit disturbance vectors to consider, which can then be shifted laterally to start on any of the 32 bits of a message word.

Of the 2^{16} candidate disturbance vectors, many do not meet the condition of being zero on their first five negative positions and their last five positions. Overall, these requirements give "10 bits of conditions", and indeed only 2^{6} vectors meet all of them, one of them being the all-zero vector [Jou09, Chapter 5]. It now remains to determine which of these lead to the best attacks.

A first rather crude way of estimating the cost of a collision associated with a certain DV is simply to count the number of local collisions it introduces, *i.e.* to consider the Hamming weight of the vector and to multiply this by the probability of a local collision being successful. This estimate can immediately be enhanced by discounting the cost of collisions appearing in the first sixteen state words, as the attacker has a full control on the corresponding message words and can then fulfill all their associated conditions deterministically. Additionally, recalling Section 3, we know that the success probability of a local collision increases if some of the bit differences are located on the MSB. Thus, the probability of a DV is not invariant by lateral shift, and each vector should be considered on its best positions only; given the pattern of corrections of a local collision, inserting the perturbations on bit one is a good choice.

There is one problem remaining with this first cost function, which is that it assumes that all local collisions are independent. This is actually not the case, and a more detailed analysis of the success probability of non-independent local collisions is necessary to get an accurate estimate of the overall cost of an attack. We do not detail such an analysis in the present case of SHA-0, and instead refer to [Jou09, Chapter 5], which shows possible interactions between local collisions in a case-by-case study. The main result of this is that some interactions introduce contradictions that cannot be resolved and which thence entirely disqualify some *DVs*. This may happen in particular because of the absorption capabilities of φ_{IF}; in that case, it disqualifies *DVs* with two consecutive perturbations in the first round, the “IF” round. Alternatively, the probability of some other interactions being successful can be increased by properly choosing the sign of some specific collisions. We will later discuss the issue further for SHA-1 in Section 9.

With this refined cost function in mind, two good disturbance vectors were found for the original attack on SHA-0 [CJ98]; we show the first of them in Figure 9.1, using an unsigned differences notation.

○○○●○○○○○○○●○○●○○○○○○○○●○○○○●●○●●○●●●●●●○
●●○●○○●○○○○●○●○●○○●○●○●○○○●○●●●○○●●○○○○○

Figure 9.1 – A good (one bit) disturbance vector for SHA-0, with a basic cost of 2^{68}. Bit zero is shown on the top left.

5 Accelerating techniques for collision search

A suitable disturbance vector such as the one of Figure 9.1 defines an explicit attack procedure in a straightforward way: one uses the available freedom in the first sixteen message words to deterministically fulfill as many conditions for local collisions, while keeping enough free bits to expect being able to fulfill the remaining conditions probabilistically.

This was the approach taken in the original attack on SHA-0, and it results in an attack of complexity 2^{67} (measured in the number of message pairs to be tested for a collision) for the vector of Figure 9.1, and 2^{61} for another good disturbance vector [CJ98].

Although this is already an attack that is significantly faster than a brute-force approach, it remains fairly expensive, and no explicit collision for SHA-0 was computed at the time. Even nearly twenty years later, such an attack requires considerable resources; it would roughly be comparable in cost with the full free-start collision for SHA-1 which is the main topic of this part. A series of techniques were later developed to make such attacks considerably more efficient, which ultimately led to the first explicit collision on SHA-0 [BCJ+05]. These improvements were of two kinds: chaining multiple blocks to enable the use of better *DVs*, and using *neutral bits* to make the probabilistic phase of the attack more efficient. The first improvement as originally used for SHA-0 was later superseded by the two-block attack structure of Wang *et al.*, used both for improved attacks on SHA-0 and for the first full theoretical attack on SHA-1 [WYY05b, WYY05a], and we will not describe it here. The second improvement, originally introduced by Biham and Chen [BC04] was much more long-lived, and it is still useful in current attacks, including ours.

The aim of the neutral bits technique is to make a better use of the available freedom in the first sixteen message words, in order to speed up the attack. The idea is to identify bits of messages that with good probability do not interact with any local collision necessary conditions, say up to step n. Thus, if one found a message pair fulfilling all conditions up to step n, called a *partial solution*, flipping a neutral bit will lead to another message pair similarly fulfilling all conditions up to n with good probability. Using neutral bits then allows to amortise the cost of finding good message pairs, which makes the attack faster. To make things a bit more formal, we may give the following:

Definition 9.1 (Neutral bits). A bit b of $\mathcal{W}_{0\le i<16}$ is a *neutral bit* for a disturbance vector $\mathcal{V}$ at step n with probability p if the probability, taken over the message space, that it does not interact with any necessary condition for $\mathcal{V}$ up to step n is equal to p.

This definition can be generalised by considering groups of bits that need to be flipped together. This may be done either because it makes these bits being more effective, or because not doing so would change the sign of some bits of local collisions later in the expanded message.

In general, one can distinguish two not mutually excluding approaches to finding neutral bits: either running a random search across many partial solutions; or using the propagation properties of the step function to identify good candidates. In particular, the latter approach found a powerful expression with the technique of auxiliary differential paths, or *boomerangs*, first developed for SHA-1 [JP07] and which led to the currently fastest attacks on SHA-0 [MP08].

A boomerang in that context is a small set of neutral bits that are particularly efficient when activated together, *i.e.* of the first kind mentioned above. The reason of this efficiency is that they define a local collisions (or sometimes a few of them) for which all necessary conditions have been pre-satisfied. Flipping the bits of the boomerang then only introduces a single local perturbation that is immediately corrected and thus does not propagate. Yet, because the local collision defined by the neutral bits is not chained along a disturbance vector, uncorrected perturbations will eventually be introduced by the message expansion and result in interactions with necessary conditions; this will however happen much later than with “standard” neutral bits. One can notice that because the local collisions of a boomerang do not need to be chained, they do not need to be of a form suitable for every Boolean function. For instance, as it was hinted earlier in Section 3, one can obtain better boomerangs by using the absorption properties of the IF function to define local collisions with fewer necessary corrections.

We conclude this short summary on SHA-0 by giving an illustration of the different phases of a one-block attack on SHA-0 in Figure 9.2. The two main rectangles represent the state and expanded messages of a computation of SHA-0. The diagonal lines (▧, ▨) represent areas of the state and messages that are fixed for an attack; any state condition within this zone is deterministically fulfilled. The non-diagonal lines in the message represent bits that are used to generate many message pairs in the hope that one leads to a collision; horizontal lines (▤) do so purely probabilistically, and vertical lines (▥) represent neutral bits which are useful to amortise the cost of finding partial solutions; the (completely determined) remainder of the expanded message is left blank. Finally, dots (⊡) represent conditions that need to be fulfilled probabilistically.

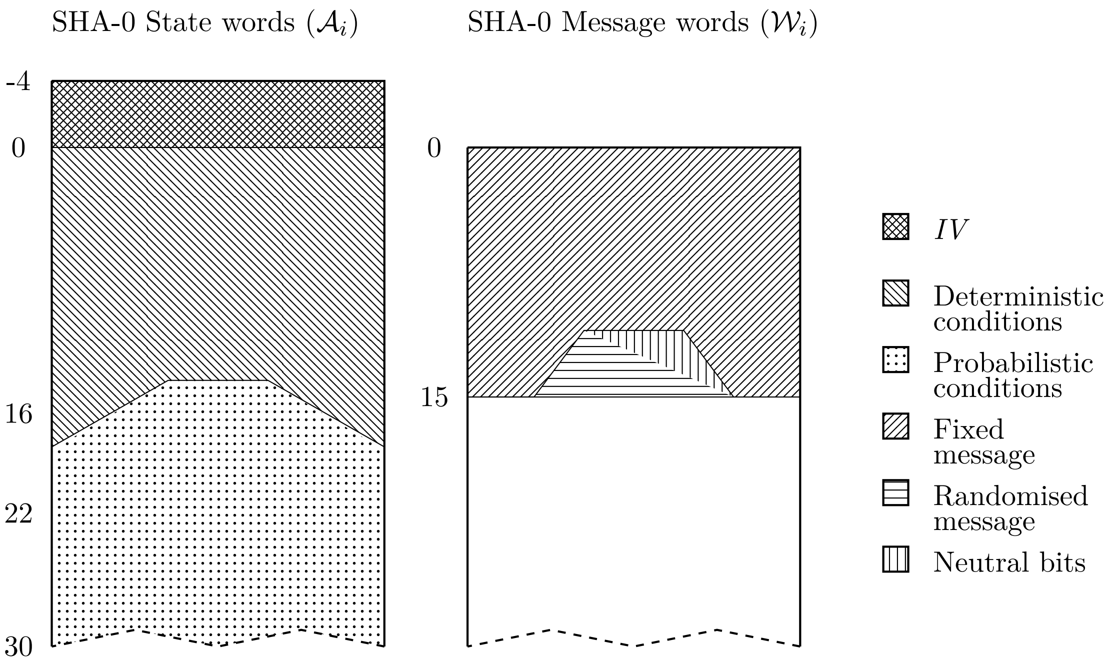


Figure 9.2 – The structure of a one-block attack on SHA-0.

6 Collision attacks on the full SHA-1

Having discussed the original attacks on SHA-0, we now turn our attention to SHA-1 for the rest of this chapter. Although both functions are very similar, their different message expansion makes a direct application of attacks on SHA-0 in the form discussed above inapplicable to the full SHA-1. We now discuss the new techniques that were developed for that purpose and some of their subsequent improvements:

1. The use of *non-linear* differential paths and of a two-block attack structure is the main improvement that makes full attacks on SHA-1 possible (Section 7).
2. The different message expansion of SHA-1 makes the search for good disturbance vectors more complex (Section 8).
3. Better cost functions were defined for the disturbance vectors (Section 9).

It is important to note that although these improvements were somehow necessary to attack SHA-1 and led to the first theoretical attack on this function [WYY05a], the same techniques can, and were, applied to improve the existing attacks on SHA-0 as well [WYY05b], and also other functions of the MD-SHA family.

Let us also mention for completeness that the original attacks of Wang *et al.* used an accelerating technique named *message modification* instead of neutral bits. Such a technique works by identifying changes in message words controlled by the attacker that only impact, say, a single necessary condition for a local collision located at a point where no direct control is possible. This can be used during the probabilistic phase of the attack to carefully fulfill some of the conditions independently of each other, which increases the probability of success. However, it is not clear whether this technique has a significant advantage over neutral bits, especially when including boomerangs, and it is somehow harder to implement efficiently, especially on a parallel architecture. Thus, we will not detail it further in this chapter.

7 The two-block structure and non-linear paths

With the benefit of hindsight, looking back on the case of SHA-0, we can identify two points in particular where the original attack does not seem to be optimal. Firstly, the original structure with a single block imposes the DV to be zero in its last five bits, which may disqualify some otherwise good DV*s*. A way to get around this restriction is to use "multi-block" attacks which culminated in using exactly two blocks.

A second point is that using a purely linearised model for the propagation of differences in the round function similarly imposes the condition that the first five negative bits of the DV have to be zero, and that there are no consecutive bits during the IF round. A way to remove these conditions is to switch to a non-linear propagation model for part of the first round. We now discuss these two improvements in more detail.

7.1 The two-block structure

In a multi-block attack, one uses DV*s* which only result in *near collisions*, *i.e.* final states still containing a few differences, only in controlled locations. The idea is then to chain such DV*s* together in a way that eventually produces a collision, by cancelling in the last block the few differences in the IV coming from a near collision in the previous block through the Merkle-Damgård domain extension with appropriate differences in the final state, thanks to the feed-forward of the Davies-Meyer construction.

This was first done by Biham *et al.* to obtain the first explicit collision on SHA-0, using four blocks of different DV*s* [BCJ+05], and it was improved by Wang *et al.* in a more systematic fashion using two blocks with the same DV, with the second block using a negated version of the DV of the first block *i.e.* with the differences being changed to their opposite, *e.g.* ▴ becomes ▾, ▵ becomes ▿. This results in a structure with a first near-collision that ends with a difference $+\Delta$, followed by a second block with a final state of negated difference $-\Delta$. In fact, following this exact structure is not strictly necessary, as there are a few admissible differences $\{+\widetilde{\Delta}\}$ for the end of the first block that can all lead to a collision in the second block.

The main reason why this structure may be used is that a non-linear propagation model is used in the beginning of the attack. Such a model in itself also allows to use better DVs.

7.2 Non-linear model for the propagation of local collisions

We have mentioned some limitations of using a purely linear model to establish the state differential path $\Delta\mathcal{A}$. In a non-linear model, we do not try to systematically avoid differences in the carry propagation of $\mathcal{A}$ and $\widetilde{\mathcal{A}}$, which also implies that not every local collision will be systematically corrected.

Two advantages of this approach over a purely linear model were already mentioned. The first is that there is no need to ensure the absence of local collisions in the beginning of the negative message anymore. The second is that one can keep successive local collisions in the IF round; this point was in fact already partially resolved by Biham *et al.* for SHA-0 [BCJ+05].

This allows to select better disturbance vectors than what would otherwise be possible, provided that one is able to find a suitable state difference, using a non-linear propagation model, that is compatible with the disturbance vector. It is however not essential, at least for a basic attack, for the path entailed by the “non-linear” state difference to be of high probability, as it will be located in part of the first round only where one has entire control of the message, as there is no obstacle to using a linear path for the remainder of the function. This amount of freedom is enough to find many solutions to the non-linear part of the differential path, even when keeping in mind that these need to leave some bits unspecified so that enough message candidates can later be generated to obtain a collision.

The other main advantage of using a non-linear path is that, as mentioned above, it is the chief reason why an efficient two-block structure can be used for an attack. Indeed, in the same way as it removes the conditions in the early message differences, it allows to disconnect the presence (for the second block) or absence (for the first) of incoming IV differences with the choice of the remaining linear part of the differential path. Thus, the same DV can be used for two blocks, and choosing opposite signs for the two is enough to lead to a collision. This new ability of using only one disturbance vector is in particular a consequence of the fact that many non-linear paths are possible for a fixed DV, even when the same IV differences are used, whereas a path following a linear behaviour is unique.

We show the simplified structure of a two-block attack using non-linear paths in Figure 9.3. The first block (on the left) takes a zero (0) IV difference, a message difference $\Delta\mathcal{W}$, and starts with a non-linear differential path NL 1 for which it is easy to find solutions in a deterministic way (🟩); this is followed by a linear path L which is satisfied probabilistically, first using accelerating techniques such as neutral bits (🟦), and then purely randomly (🟥). This results in a state difference $\Delta\mathcal{A}$ which is passed to a second block. This block uses a different non-linear path NL 2 to connect to the negated linear path $-L$ that is obtained by using an oppositely signed message difference $-\Delta\mathcal{W}$; following this second path leads to a collision.

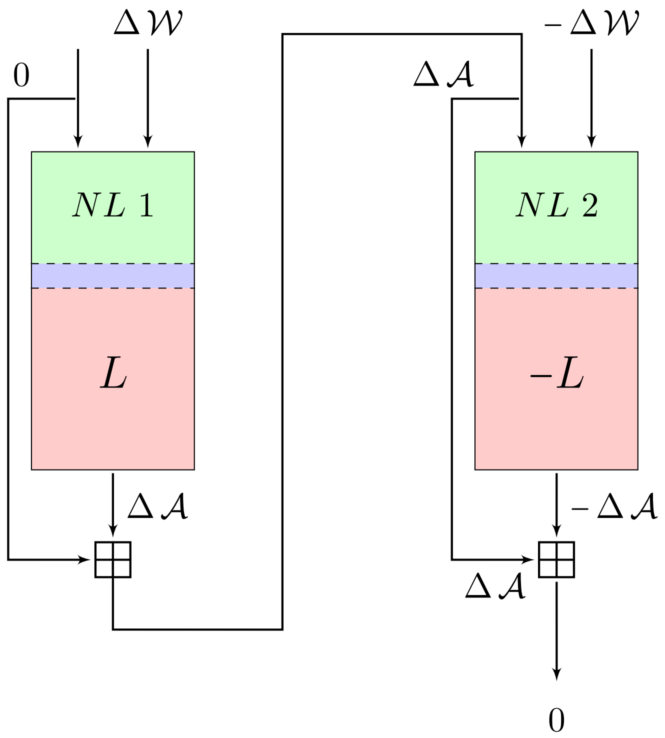


Figure 9.3 – The structure of a two-block collision attack for SHA. Figure adapted from [Jea]

The main difficulty in using non-linear paths is to find the paths themselves, as there is a large number of possible behaviours to take into account. The search was initially done by hand for the first attack on SHA-1, but automated tools were later developed to make this process much more efficient. One can for instance cite the work of De Cannière and Rechberger [DR06], who used a guess-and-determine approach to find new non-linear paths, leading in particular to an explicit two-block collision for SHA-1 reduced to the at-the-time record number of 64 steps.

The idea of the guess-and-determine method is to define a set of *constraints* that encode a differential path, along with an efficient constraint-propagation algorithm. One then starts from an underdefined initial path with many unconstrained differences but with the conditions for the IV and for the connecting linear path already set and iteratively chooses an unconstrained difference at random, assigns it a value, and propagates the consequences of this choice. A backtracking strategy is also used to escape situations where no more valid choices are possible. A path is found when every constraint is either a signed difference or a possibly signed equality.

One can also mention the alternative "meet-in-the-middle" approach for the construction of these paths, which was used by Yajima *et al.* [YSN+07], and later Stevens [Ste12]. This method works by defining two partial differential paths, one expanded forward *e.g.* starting from the IV and one expanded backward *e.g.* starting from the purely linear part of the path, that are then connected on a few consecutive steps.

The advantages of an automatic search of the non-linear part of the differential path over a manual one are twofold: first, it is much faster to create new attack instances, which allows for example to experiment quickly with several DVs. This is particularly useful if one wants both to mount attacks for the full function (even without running the attack completely) and attacks for a high number of reduced steps, the best DVs in each case being likely different. Second, the ability to generate many non-linear paths allows to search for ones that have few constraints (for instance leading to more available neutral bits) or that can incorporate more preset constraints that may for instance aid in the use of boomerang neutral bits.

8 Classes of disturbance vectors for SHA-1

We recall from Section 4 that the search for disturbance vectors for SHA-0 naturally reduces to a search among only 2^{16} candidates. Once a cost function is chosen, it is simple to evaluate it on every potential DV and one just needs to keep the best of them. Unfortunately, the message expansion of SHA-1 can no longer be seen as thirty-two smaller independent message expansions, making the search space for potential DVs significantly larger.

In the original attack from Wang *et al.*, the DV was found when searching through a reduced space of size 2^{38}, using the Hamming weight of the resulting DVs as the primary cost function. Subsequently, a significant amount of work focused on finding alternate disturbance vectors in the hope of decreasing the cost of the probabilistic phase of the attack. Manuel then noticed that all DVs suggested in the literature could actually be concisely described by two simple classes which lead to the best known vectors [Man11]; we now summarise these results.

We start by defining an *extended expanded message* for SHA-1 as follows:

Definition 9.2 (Extended expanded message). Let $\mathcal{M}$ be a SHA-1 message block made of sixteen 32-bit message words $\mathcal{M}_0, \ldots, \mathcal{M}_{15}$. The *extended expanded message* $\overline{\mathcal{W}}$ for $\mathcal{M}$ is made of 144 32-bit words $\overline{\mathcal{W}}_{-64}, \ldots, \overline{\mathcal{W}}_{79}$ defined by:

$$\overline{\mathcal{W}}_i = \begin{cases} \mathcal{M}_i, & \text{for } 0 \leq i \leq 15 \\ (\overline{\mathcal{W}}_{i-3} \oplus \overline{\mathcal{W}}_{i-8} \oplus \overline{\mathcal{W}}_{i-14} \oplus \overline{\mathcal{W}}_{i-16}) \circlearrowleft 1, & \text{for } 16 \leq i \leq 79 \\ \overline{\mathcal{W}}_i = (\overline{\mathcal{W}}_{i+16} \circlearrowright 1) \oplus \overline{\mathcal{W}}_{i+13} \oplus \overline{\mathcal{W}}_{i+8} \oplus \overline{\mathcal{W}}_{i+2}, & \text{for} - 64 \leq i \leq -1 \end{cases} . \tag{9.1}$$

In other words, an extended expanded message expands an initial message both forwards (using Equation 8.2) by 64 words, but also backwards (using Equation 8.4) by a similar amount. By definition, every consecutive 80 words $\overline{\mathcal{W}}_i, \ldots, \overline{\mathcal{W}}_{i+79}$, $i \in [-64, \ldots, 0]$

of $\overline{\mathcal{W}}$ form a valid expanded message "$(\overline{\mathcal{W}}_i)$" for SHA-1. Furthermore, it is easy to check that these 65 expanded messages are exactly the 65 possible such messages for which sixteen consecutive words are equal to $\mathcal{M}$.

We also note the following fact:

Fact 9.1 (The message expansion of SHA-1 is a quasi-cyclic code). *If $\mathcal{W} = \mathcal{W}_0, \dots, \mathcal{W}_{79}$ is a valid expanded message for SHA-1, then for every $i \in [0, 31]$, $\mathcal{W}^{\circlearrowleft i} := \mathcal{W}_0^{\circlearrowleft i}, \dots, \mathcal{W}_{79}^{\circlearrowleft i}$ is a valid expanded message for SHA-1.*

This fact, together with the notion of extended expanded message allows to define equivalence classes for expanded messages:

Definition 9.3 (Equivalence class for SHA-1 expanded messages [Man11]). Two SHA-1 expanded messages $\mathcal{W}$ and $\mathcal{W}'$ are equivalent if there are two pairs (i, j), (i', j') in $[-64, 0] \times [0, 31]$ and an extended expanded message $\overline{\mathcal{W}}$ such that $\mathcal{W} = (\overline{\mathcal{W}}_i)^{\circlearrowleft j}$ and $\mathcal{W}' = (\overline{\mathcal{W}}_{i'})^{\circlearrowleft j'}$.

In other words, two expanded messages are equivalent if they can be generated from the same, possibly rotated, extended expanded message. It should be noted however that a message necessarily belongs to many distinct such equivalence classes.

As the disturbance vectors are expanded messages themselves, one can then use equivalence classes as a natural way to segment the search space for good *DVs*. For instance, Manuel searched for candidates among all classes of extended expanded messages defined from windows of Hamming weight up to four, and for some classes defined by windows of weight five and six. It followed from this search that all good *DVs*, including all the ones described in previous work, come from two equivalence classes. The 16-word messages generating the extended expanded messages of the two classes (up to rotation) are shown in Figure 9.4. The messages in this figure are given using an unsigned difference notation. In Definition 9.2, we gave the definition using a "normal" message $\in \{\{0, 1\}^{32}\}^{16}$. As a disturbance vector is eventually used to define the difference between two messages, we think that using such a notation is appropriate in this case. Following this observation, Manuel termed $\mathrm{I}(i, j)$ and $\mathrm{II}(i, j)$ the disturbance vectors $(\overline{\mathcal{W}}_{-i})^{\circlearrowleft j}$ where $\overline{\mathcal{W}}$ is generated from the messages of type I and II of Figure 9.4 respectively.

9 Exact cost functions for disturbance vectors

We have already mentioned the role played by the cost functions when choosing a disturbance vector, both in the case of SHA-0 in Section 4 and in the case of SHA-1 in the previous Section 8. A first basic such function is simply to take the Hamming weight of a vector, *i.e.* to count the number of local collisions, over the steps where we expect the attack to be purely probabilistic, *e.g.* starting from step 20. Even in the case of SHA-0, we have seen that some additional interactions between local collisions need to be taken into account to make this function more accurate. The same sort of interactions is also

```
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ●○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ●○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○●  ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○●
```

Figure 9.4 – The messages defining the class of type I (left) and type II (right) disturbance vectors, given as sixteen 32-bit words $\mathcal{M}_0, \ldots, \mathcal{M}_{15}$, with $\mathcal{M}_0$ on top.

present in the case of SHA-1, and some new ones may appear as well, especially because several local collisions may now be started on the same step.

9.1 Bit compression

A first new kind of interaction between local collisions that is favourable to the cryptanalyst is the effect used in the technique of *bit compression* introduced by Wang *et al.* [WYY05a] as a *special counting rule* and later named as such by Yajima *et al.* [YIN⁺08]. Under certain conditions, this technique allows to significantly improve the joint probability of two or more neighbouring local collisions being successful by making it as high as for a single one. In a nutshell, the idea is to introduce the initial perturbations using a chain of differences all having the same sign, except for the last one, and to let the carry propagate from the first perturbation to the other ones instead of preventing such a propagation every time. Let us see how this may work on an example with three neighbouring differences.

Example 9.1 (Compression of three differences). Consider a chain of differences of the form … ○ ▲ ▼ ▼ ○ … added to a state with no difference … ○ ○ ○ ○ ○ …. It is most useful in this case to first consider the differences as modular ones. If we call $x, \widetilde{x}$ and $y, \widetilde{y}$ the two states that are added together, we have $\widetilde{x} = x - 2^i - 2^{i+1} + 2^{i+2}$ for some value i. Now we want to determine what are the probabilities of some of the possible resulting XOR differences between $x + y$ and $\widetilde{x} + \widetilde{y}$.

Using a traditional view and treating all differences separately, which is what would happen if these were perturbations of local collisions seen independently, we would like to have no carry propagation at any of the three positions, to obtain an XOR difference on the same three positions where differences were introduced. It is easy to see that this imposes three conditions on $x+y$, as it means that we want $\widetilde{x}+\widetilde{y} = (x+y) \oplus 2^i \oplus 2^{i+1} \oplus 2^{i+2}$, which translates to bits i, $i+1$ and $i+2$ of $x+y$ being 1, 1 and 0 respectively. The probability of this happening is thus 2^{-3}.

However, as we have $2^{i+2} - 2^{i+1} - 2^i = 2^i$, the alternative XOR difference $\widetilde{x} + \widetilde{y} = (x + y) \oplus 2^i$ only imposes one condition on $x + y$, namely that bit i must be 0; this difference may then happen with probability 2^{-1}.

We can use the effect showed in Example 9.1 to increase the probability of a successful introduction of perturbations in series of local collisions; this is however at the condition that the "compressed" XOR difference obtained as a result is compatible with the following corrections, which are still located on multiple bits. Fortunately, this condition is always fulfilled provided that the single difference is not absorbed or flipped in a Boolean function —the usual condition for a successful correction— and that the chain of consecutive differences is not broken through the bit rotations —a "hard" condition that determines if a series of neighbouring local collisions can indeed be compressed. We illustrate this by continuing our previous example.

Example 9.2 (Correction of compressed differences)**.** Let us use modular differences again. Consider the case of Example 9.1 and assume that the introduction of the perturbation resulted in the modular and XOR difference $\widetilde{x} + \widetilde{y} = (x + y) + 2^i = (x + y) \oplus 2^i$. Assume that this difference is preserved through the step function, excluding the addition of the message, resulting in a partial state $z, \widetilde{z}$ with $\widetilde{z} = z + 2^i$. This partial state is rotated to the left by $r \in \{0, 5, 30\}$ in the computation of the new state, and the differences in the message at this point are at positions $\alpha = i + r \mod 32$, $\beta = i + 1 + r \mod 32$, $\gamma = i + 2 + r \mod 32$ and of sign opposite the ones of the initial perturbation, *i.e.* we have $m, \widetilde{m}$ with $\widetilde{m} = m + 2^\alpha + 2^\beta - 2^\gamma$. If $\alpha < \beta < \gamma$, we then have $\widetilde{m} = m - 2^\alpha$. In that case, $(\widetilde{z} \circlearrowleft r) + \widetilde{m} = (z \circlearrowleft r) + 2^\alpha + m - 2^\alpha = (z \circlearrowleft r) + m$, and the correction is indeed successful.

It is worth noting that because only a single state difference needs to be preserved through the Boolean functions during the correction, the probability of all corrections of compressed local collisions being successful is also much higher, compared to the uncompressed case.

To summarise, local collisions starting at the same step and at neighbouring bit positions that remain consecutive through left rotations by 5 and 30 (*i.e.* the considered bit positions do not include bit 1 or 26 and bits to their left) can be compressed by choosing a proper signing for the initial perturbation. The probability of the resulting compressed collision to be successful is the same as the success probability of a single local collision.

Finally, let us note that compressing local collisions does not actually hinder in any way their chance of being successful in the "traditional" independent way. For instance, two local collisions in no particular position during an XOR round have a "theoretical" joint success probability of 2^{-8} when considered independent and of 2^{-4} if they are compressed as per Section 3. The two successful events being independent, the total theoretical success probability of these collisions is thence $2^{-3.91}$. It may thus appear that compressed local collisions actually have a *higher* probability than single ones. This is actually not the case, as we explain next.

9.2 Bit decompression

We now use the insight gained from the analysis of the bit compression technique to come closer to computing the exact success probability of local collisions. The observation we make here is that in the same way as neighbouring XOR bit differences of appropriate sign can be compressed into a single modular difference, a single XOR difference can be "decompressed" into a series of multiple modular ones. Similarly as for bit compression, if this difference is the perturbation of a local collision, the corrections may still be effective if they can themselves be decompressed.

In other words, it is not necessary that the introduction of the perturbation of a local collision does not trigger a difference in carry propagations; even if this is the case, the local collision may still be successful if the resulting state difference is preserved by the Boolean functions and if the carry chain is not broken by rotations during the corrections. Thus, the success probability of a single local collision not on a rotation boundary is strictly higher than the probability obtained from the analysis of Section 3.

We may rephrase this in a slightly more formal way as follows:

Consider a local collision started by an initial perturbation on m_s and $\widetilde{m_s}$ of positive sign at position $i < 31$, *i.e.* with a signed bit difference $\ldots \circ \blacktriangle \circ \ldots$. This corresponds to a modular difference $\widetilde{m_s} = m_s + 2^i$. Call x, $\widetilde{x}$ and y, $\widetilde{y}$ the state to which the message m_s, $\widetilde{m_s}$ is added and the result of this addition respectively. The probability (over the values of x) of having a signed difference $\ldots \circ \blacktriangle \circ \ldots$ for y, $\widetilde{y}$ is 2^{-1}. However, we also have $\widetilde{m_s} = m_s - \sum_{j=i}^{k-1} 2^j + 2^k$ for any $k < 32$. Thus we can write $\widetilde{y} = \widetilde{x} + \widetilde{m_s} = x + m_s - \sum_{j=i}^{k-1} 2^j + 2^k$. The probability of having a difference $\ldots \circ \blacktriangle \blacktriangledown \circ \ldots$ between $\widetilde{y}$ and y is thus 2^{-2}; more generally, the probability of having a difference $\ldots \circ \blacktriangle \underbrace{\blacktriangledown \ldots \blacktriangledown}_{u \text{ times}} \circ \ldots$ between $\widetilde{y}$ and y (with $i + u + 1 < 32$) is 2^{-u-1}.

For an initial difference of weight $u + 1$, the corrections on subsequent message words m_{s+o}, $\widetilde{m_{s+o}}$ are of the form $\widetilde{m_{s+o}} = m_{s+o} - 2^{i+r \mod 32}$, $r \in \{0, 5, 30\}$. An initial perturbation that resulted in a difference on y, $\widetilde{y}$ of weight $u + 1$ can thus be corrected at every step only if $i + r \mod 32 \leq i + u + r \mod 32$, because the equality $\widetilde{m_{s+o}} = m_{s+o} + \sum_{j=i+r \mod 32}^{i+r+u-1 \mod 32} 2^j - 2^{i+r+u \mod 32}$ must hold.

Now assume that the maximal weight of an initial perturbation that can be corrected is v, and that for the sake of simplicity all induced perturbations are in an XOR round in no particular position, meaning that no correction is on the MSB, then the success probability of having a local collision is $\sum_{i=1}^{v} 2^{-4v}$, which is higher than the probability 2^{-4} obtained by considering only the signed difference $\ldots \circ \blacktriangle \circ \ldots$.

A complete analysis of the impact of carry propagation on the success probability of a single local collision was done by Mendel *et al.* for all Boolean functions and positions of the perturbation [MPRR06]. Manuel also performed experiments validating this analysis [Man11]; in particular, these results seem to show that for a single local collision, no additional effect contributes to the success probability.

To conclude, we have seen that the interaction of XOR and modular differences in SHA leads to a slight *differential* effect for the disturbance vector. A single message difference actually defines several, not mutually exclusive, local collision patterns. Even though one of these patterns is much more likely than the others, *i.e.* the one with all possible compressions effectively done and no decompression, the contribution of the remaining ones is not nil.

9.3 Joint local collision analysis

We have just considered how the propagation of carries may influence the probability of a single local collision and of neighbouring local collisions started on the same step. We now consider how to account for similar effects that impact the joint success probability of local collisions sharing some common steps. We have already mentioned in Section 4 that an analysis in the spirit of Section 3 may be done on closely interacting local collisions, for instance on local collisions that share some correction bits or that have some corrections interacting together through the Boolean functions. However, this does not take into account the effects of carries, and a more precise study is thus possible.

We now summarise the *joint local collision analysis* (JLCA) approach of Stevens [Ste12, Ste13], which allows to compute the exact best probability of a disturbance vector by considering all interactions between non-disjoint local collisions. This results in a very good "exact" cost function for *DVs* of SHA-1.

The objective here is actually slightly more generic: for a given DV, there is some liberty in the choice of the actual message differences, *e.g.* by specifying their sign, and we have already seen that this may impact the success probability of local collisions. These variations should be taken into account when comparing *DVs* together, and only the message differences resulting in the best probability should be considered. Going further, for a given range of steps, we may want to determine which initial and final state differences yield the best probability. Thus we may reformulate our objective as wanting to find the maximum success probability for a given DV and a prescribed number of steps over the choice of compatible signed message differences and initial and final state differences.

To fulfill this objective, we may simply try for a given configuration to enumerate all differential paths and sum their probabilities. However, although this was feasible for a single local collision, cf. [MPRR06], the amount of paths to consider makes this task computationally intractable in general. The idea of Stevens to get around this limitation is to use a notion of *reduced* paths together with equivalence classes of message differences. A reduced path is essentially obtained from its non-reduced analogue by removing differences not interacting with the initial and final differences. Such paths can easily be enumerated, and most importantly it is possible to compute their associated "cumulative probabilities" which is, for a given reduced path and a given message difference, the sum of the probabilities of all possible complements to the reduced path that result in a valid overall path. Although the number of possible message differences to consider may be big, Stevens also shows how to find equivalence classes yielding the same probability for all their members. It is then enough to perform the computation for one representative of every class.

The above strategy allows to exactly compute the best achievable probability for a given DV over a given range of steps. However, one should keep in mind that the DV with the best such probability for the range of steps one wishes to consider is not necessarily the one most suited to an attack. Indeed, somehow in the same way as *DVs* were disqualified in the original SHA-0 attacks because of incompatibilities in the IF round for the model used at the time, a DV with high associated probability may be a worse choice than another with a lower probability if the former makes it harder to find a good non-linear part for the first round than the latter, or similarly if it makes it harder to use accelerating techniques.

These last criteria are much harder to capture into a cost function, and there was no attempt to do so in the literature. Ultimately, the complexity of the non-probabilistic phase of an attack can only be precisely determined by evaluating the speed at which an efficient implementation produces partial solutions up to a step where no freedom remains. The cost functions as presented in this entire chapter are then a very useful tool to precisely extrapolate the complexity of a full attack from this point on.

Chapter 10

Freestart collision attacks for SHA-1

1 A framework for freestart collisions for SHA-1

This section presents in detail our framework for freestart collision attacks on SHA-1. We start by presenting our general approach to such attacks in Section 1.1, and the addition and improvements to existing techniques that were developed to make the attacks possible in Section 1.2.

1.1 Faster collisions by exploiting more freedom

The main interest of freestart collisions is that they consist in an attack against a meaningful, though weakened security notion, while granting more freedom to the attacker by providing him with an additional input. It is thus expected that such attacks should be more efficient than ones targeting stronger notions *e.g.* hash function collisions, not unlike attacks targeting a reduced version of a function, using fewer steps.

However, it is not necessarily clear in general how to efficiently exploit the additional input in a freestart attack. We describe below our strategy in that respect in the case of SHA-1. Note that unlike hash function attacks *à la* Wang, our freestart collisions use only one block, which has some consequences when building the attack.

The basic idea underlying the structure of our freestart collisions is to start the initialisation of the hash function from a "middle" state rather than from the IV, and to similarly initialise the message with an offset. As both of the step function and the message expansion of SHA-1 can easily be inverted, any computation started from the middle can be equivalently defined as starting from the beginning. We call *main block offset* the offset in the initialisation of the message, giving us the technical definition:

Definition 10.1 (Main block offset). A freestart collision attack on SHA-1 uses a *main block offset* of $i \in \{0, \ldots, 64\}$ if the candidate expanded messages $\mathcal{W}$ tested for a collision are defined through the message expansion of the words $\mathcal{W}_i, \ldots, \mathcal{W}_{i+15}$.

The desired effect of using a non-zero main block offset is to move down the part where freedom is available, enabling to satisfy conditions up to a later point in the attack, either deterministically or with accelerating techniques. Being able to do so would mechanically make the attack more efficient, at the condition that this gain is not entirely offset by the cost of satisfying potential conditions when computing backward to the corresponding IV, this computation being eventually necessary to fully determine the colliding inputs.

In the case of a freestart attack, the entire freedom of the IV implies that no *a priori* constraints are set in the middle initialisation. This is in contrast to a hash function attack, where the necessity for the backward-computed IV to be of a specific value makes the approach less powerful, although potentially still useful.

It should be noted that this kind of broad approach has already been used successfully in the past in various hash function attacks, see *e.g.* [Dob96, LP13, MRS14]. We conclude this discussion by considering more closely how we applied this technique to SHA-1.

1.1.1 Different choices for the disturbance vectors

We already mentioned at length the importance of the disturbance vectors in attacks on SHA-1. When shifting the window where actual freedom is used, it seems natural to also shift the disturbance vector. This is in part a consequence of the fact that good DVs tend to naturally include a series of steps for which they impose a lot of conditions. These can easily be resolved provided that enough freedom is available, but they would heavily impact the complexity of an attack, were they to be satisfied probabilistically. It is thus important that this part remains in the window of available freedom.

As the equivalence classes of Manuel already include vertical shifts in their definitions, the two known good classes are already suitable to the search for good DVs for freestart collisions. We simply expect to use DVs with different shift values than the ones usually considered.

The structure of the attack also imposes additional conditions on the DV, compared to a usual two-block hash function attack. Both conditions come from the aforementioned fact that bit conditions which ensure that the message pair follows a proper differential path may need to be satisfied probabilistically when computing the IV backward from the values of the middle initialisation. To make this phase efficient, we would like to have as few such bit conditions as possible. This translates to the two high-level conditions:

1. The DV should not imply too many differences in the last five steps of the attack. This is because we aim for a one-block attack, thus corresponding differences will need to be inserted in the backward-computed IV for the pair to define a collision.

2. The DV itself should not include too many differences in its early words, that is the ones used in the backward computation, as we observed that being dense on these positions makes it harder to connect to the IV with few conditions. Additionally, it may decrease the overall probability of the computation.

1.1.2 New constraints for the accelerating techniques

The necessity of satisfying bit conditions in a backward computation also imposes some constraints on the accelerating techniques that can be used. These all have in common that they work by selectively changing the value of some bits of the message. As the structure of our attacks implies shifting down the free window of the message, changing a bit in this window may also impact the value of the first few message words that are computed with the backward message expansion. This may potentially result in unwanted interactions with the ability to satisfy the backward bit conditions.

In short, the selection of accelerating techniques, *e.g.* neutral bits, must take into account their "back effects", which may disqualify some otherwise good candidates.

1.2 New techniques for freestart collisions

There is a certain number of points that need to be specified to make the description of our framework complete. In particular, we will discuss here:

1. The construction of non-linear paths adapted to the shifted initialisation and to the requirements of high-probability backward computations, and improvements to existing methods (Section 1.2.1).

2. The accelerating techniques used in our attacks (Section 1.2.2).

3. The modified attack process adapted to the shifted initialisation (Section 1.2.3).

1.2.1 Differential paths for freestart attacks

In our freestart attacks, we have the requirements that the computation from the initialised state back to the IV be of high probability. Ideally, we would even like to be able to satisfy deterministically all the conditions associated with this computation, while still performing a shifted initialisation. A direct consequence of this is that the differential path should be sparse in the steps involved in the backward computation, *i.e.* include few conditions and especially few differences.

The known methods for the construction of non-linear differential paths for SHA do not provide good guarantees that the path will be sparse in *a priori* specified state words. However, it is not hard to search independently for a sparse prefix for the differential path and then only to search for paths that are extensions of this prefix. We implemented this strategy by first greedily searching for prefixes of various lengths and with few differences, and then by searching for good paths extending these prefixes. We show in Figure 10.1 the starting point that was eventually used in the search for the path of the 76-step attack, using a representation suitable for a guess-and-determine path search. Note that in order to ensure a collision through the Davies-Meyer feed-forward, a sign has been imposed in the differences in the IV.

Through the course of the preparation of the attacks, we tried both methods for constructing the extension of the non-linear path, *i.e.* a guess-and-determine and a meet-in-the-middle approach. We have found that the meet-in-the-middle approach

i	$\mathcal{A}_i$	$\mathcal{M}_i$
-4	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-3	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-2	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○▲○	
-1	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○▲	
0	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○
1	○○○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○	○○○○○●○○○○○○○○○○○○○○○○○○○○○●●●○○
2	○○○○○●○○○○○○○○○○○○○○○○●○○○○○○●○○	●●○○●●○○○○○○○○○○○○○○○○○○○○○●○●○○
3	○○○○○●○○○○○○○○○○○●○○○○○○●○○○○●○○	○○○○●●○○○○○○○○○○○○○○○○○○○○○○○○●○
4	○●○○○○○○○○○○●○○○○○○●○○○○●○○○○○●○	●●○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○
5	○●○○○○○○○○○○○○●○○○○●○○○○○○●○●○○○	●○●●○●○○○○○○○○○○○○○○○○○○○○○●●●○○
6	********************************	○○●●●●○○○○○○○○○○○○○○○○○○○○○○○●○○
7	********************************	●○●●●●○○○○○○○○○○○○○○○○○○○○○●●○●○
8	********************************	○○●○○○○○○○○○○○○○○○○○○○○○○○○●○○○○
9	********************************	○○●○○●○○○○○○○○○○○○○○○○○○○○○●●●○○
10	********************************	

Figure 10.1 – A sparse prefix for the non-linear differential path of a freestart attack. The symbol “$*$” denotes the absence of conditions.

allowed to find paths with fewer conditions, and that it allowed a better control of the position where some of these conditions were located. A slight downside of our meet-in-the-middle implementation, however, is that it did not give strict guarantees that the paths it produced were completely valid, as they may include contradictory conditions, for instance in the high-density part of the path where many differences are located. However, this issue could usually be resolved by considering slight variations of the impossible path, which we were able to do efficiently with a guess-and-determine method.

The evaluation and eventual selection of the *DVs* used in our attacks was done using joint local collision analysis. We modified the original implementation of Stevens [Ste13] to cover *all* steps, including the non-linear part, and to produce the entire set of sufficient state conditions and message bit relations as used by collision attacks. These are the conditions placed on the expanded messages $\mathcal{W}$ that ensure that the highest probability that is reachable for a given DV is indeed attained. More specifically, we improved JLCA in the following ways:

1. Originally JLCA considered the entire set of differential paths that conform to the disturbance vector only over the linear part. This was done by considering sets $\{\Delta\mathcal{A}_i\}$ of allowed state differences for each state word $\mathcal{A}_i$ given the disturbance

vector, including carries. We extended this to aid in the non-linear path search by defining sets $\{\Delta \mathcal{A}_i\}$ for the non-linear part as the state difference given by the previously constructed differential path of the non-linear part. Here one actually has a few options: only consider exact state difference of the non-linear path or also consider changes in carries or signs, as well as include state differences conforming to the disturbance vector. We found that allowing changes in carries or signs for the state differences given by the non-linear path made JLCA impractical, yet including state differences conforming to the disturbance vector was practical and had a positive effect on the overall probability of the full differential path.

2. In the original JLCA, the probability computation took into account the possible carry propagations in local collisions, as this improves the overall probability, the probability of variants of paths adding up. However, until a certain step, say 25, a typical attack implementation requires the partial solutions to strictly follow the main differential path, and thus variants induced by carry propagations are actually not allowed to happen in an attack, and thus should not be counted as improving the probability. We thus corrected JLCA by not adding up the probability of paths over the first 25 steps, but only by taking the maximum probability. In our implementation, this can be simply done by replacing the addition of two probabilities by taking their maximum, conditionally on the current SHA-1 step.

3. Originally JLCA only gave as output starting differences, ending differences, message bit relations and the optimal success probability. We extended it so that it can also reconstruct the set of differential paths over steps, say, $[0, 25]$ and determine minimal sets of sufficient conditions and message bit relations. This can be made possible by keeping the intermediate sets of reduced differential paths $\mathcal{R}_{[t_b,t_e]}$ which were constructed backwards starting at a zero-difference intermediate state of SHA-1. Then one can iteratively construct sets $\mathcal{O}_{[0,i[}$ of optimal differential paths over steps $0, \ldots, i-1$, *i.e.*, differential paths compatible with some combination of the optimal starting differences, ending differences and message bit relations such that the optimal success probability can be achieved. One starts with the set $\mathcal{O}_{[0,0]}$ determined by the optimal starting differences. Given $\mathcal{O}_{[0,i[}$ one can compute $\mathcal{O}_{[0,i+1[}$ by considering all possible extensions of every differential path in $\mathcal{O}_{[0,i[}$ with step i. From all those paths, one only stores in $\mathcal{O}_{[0,i+1[}$ those that can be complemented by a reduced differential path over steps $i+1, \ldots, t_e$ from $\mathcal{R}_{[i+1,t_e]}$ such that the optimal success probability is achieved over steps $0, \ldots, t_e$.

   Now given, say, $\mathcal{O}_{[0,26[}$, we can select any path and determine its necessary and sufficient conditions for steps $0, \ldots, 25$ and the optimal set of message bit relations that goes with it. Although we use only one path, having the entire set $\mathcal{O}_{[0,26[}$ opens even more avenues for future work. For instance, one might consider an entire subclass of 2^k differential paths from $\mathcal{O}_{[0,26[}$ that can be described by state conditions linear in message bits and a set of linear message bit relations. This would provide k bits more in degrees of freedom that can be exploited by speed up techniques.

We thus proposed several extensions to the original JLCA that allowed us to determine sufficient state conditions and message bit relations optimised for collision attacks, *i.e.* minimal set of conditions for paths attaining the highest success probability.

1.2.2 Instantiating accelerating techniques

Generally speaking, the accelerating technique used in our attacks are neutral bits. Only "single-bit" neutral bits were used in the attack on 76 steps, but the 80-step attack also used "boomerang" neutral bits where three (and in one occasion, four) carefully chosen bits are flipped at once to compute the pair of related messages. Additionally, in both attacks, some additional changes in the message may be necessary, depending on which neutral bits are activated, to ensure that no message bit relation becomes violated. In the following, we use the term *neutral bit* to mean either of the "single-bit" or "boomerang" neutral bits.

The main difference in the selection of neutral bits compared to a usual, non-freestart attack is the presence of the main block offset, which determines the offset of the message freedom window used during the attack. We have selected a main block offset of 6 (resp. 5) in the 76-step (resp. 80-step) case, as this led to the best distribution of usable neutral bits and boomerangs. This means that all the neutral bits, including potential boomerangs, directly lead to changes in the state from steps 6 (resp. 5) up to 21 (resp. 20) and that these changes propagate to steps 5 (resp. 4) down to 0 backwards and steps 22 (resp. 21) up to 79 forwards.

We describe below the selection process more specifically in the case of the 80-step attack; the approach was similar in the 76-step case, with the omission of the boomerangs neutral bits which were not used.

The search for both single and boomerang neutral bits requires to evaluate the probability that a candidate does not interact badly with path conditions up to a certain point. This probability is estimated experimentally by observing the effect of activating potential neutral bits on many partial solutions for the differential path. Because the dense area of the attack conditions may implicitly force certain other bits to specific values, resulting in hidden conditions, we used more than 4000 partial solutions over steps 1 up to 16 in the analysis. The 16 steps fully determine the message block, and also verify the sufficient conditions in the IV and in the dense non-linear differential path of the first round. It should be noted that for this step it is important to generate every sample independently. Indeed using *e.g.* message modification techniques to generate many samples from a single one would result in a biased distribution where many samples would only differ in the last few steps.

Searching for Boomerangs. We analyse potential boomerangs from the framework of Joux and Peyrin [JP07], which initially flip a single state bit together with 2 or more message bits. Each boomerang should be orthogonal to the attack conditions, *i.e.*, the state bit where a difference is introduced should be free of sufficient conditions, while flipping the message bits should not break any of the message bit relations either directly or through the message expansion. Let $t \in [6, 16], b \in [0, 31]$ be such that the state bit $\mathcal{A}_t[b]$ has no sufficient condition.

First, we determine the best usable boomerang on $\mathcal{A}_t[b]$ as follows. For every sampled solution, we flip that state bit and compute the signed bit differences between the resulting and the unaltered message words $\mathcal{W}_5, \ldots, \mathcal{W}_{20}$. We verify that the boomerang is usable by checking that flipping its constituting bits breaks none of the message bit relations. We normalise these signed bit differences by negating them all when the state bit is flipped from 1 to 0. In this manner we obtain a set of usable boomerangs for $\mathcal{A}_t[b]$. We determine the additional conditions on message and state bits associated with the boomerangs that ensure that the initial local collision corrections are successful with probability one, and only keep the best usable boomerang that has the fewest such conditions.

Secondly, we analyse the behaviour of the boomerang over the backward steps. For every sampled solution, we simulate the application of the boomerang by flipping all of its bits. We then recompute steps 4 to 0 backwards and verify if any sufficient condition on these steps is broken. Any boomerang that breaks any sufficient conditions on the early steps with probability higher than 0.1 is dismissed.

Thirdly, we analyse the behaviour of the boomerang over the forward steps. For every sampled solution, we simulate the application of the boomerang by flipping its constituting bits. We then recompute steps 21 up to 79 forwards and keep track of any sufficient condition for the differential path that becomes violated. A boomerang will be used at step i in our attack if it does not break any sufficient condition up to step $i - 1$ with probability more than 0.1.

This process is iterated as long as good boomerangs are found.

Searching for single neutral bits. The neutral bit analysis uses the same overall approach as the boomerangs, with the following changes. After boomerangs are determined, their conditions are added to the previous attack conditions and used to generate a new set of solution samples. Usable neutral bits consist of a set of one or more message bits that are flipped simultaneously. However, unlike for boomerangs, the reason for flipping more than one bit is to preserve message bit relations, and not to control the propagation of a state difference. Let $t \in [5, 20], b \in [0, 31]$ be a candidate neutral bit; flipping $\mathcal{W}_t[b]$ may possibly break certain message bit relations. We express each message bit relation over $\mathcal{W}_5, \ldots, \mathcal{W}_{20}$ using linear algebra, and use Gaussian elimination to ensure that each of them has a unique last message bit $\mathcal{W}_i[j]$, *i.e.* where $i * 32 + j$ is maximal. For each relation involving $\mathcal{W}_t[b]$, let $\mathcal{W}_i[j]$ be its last message bit. If (i, j) equals (t, b) then this neutral bit is not usable; indeed, this would mean that its value is fully determined by earlier message bits. Otherwise we add bit $\mathcal{W}_i[j]$ to be flipped together with $\mathcal{W}_t[b]$ as part of the neutral bit. Similarly to boomerangs, we dismiss any neutral bit that breaks sufficient conditions backwards with probability higher than 0.1. The step i in which the neutral bit is used is determined in the same way as for the boomerangs.

1.2.3 The shifted initialisation and the attack process

In the implementations of our attacks, we only apply neutral bits on fully determined message blocks. In other words, neutral bits are used to try to extend to more steps partial solutions for the differential path which cannot be extended by using more freedom in the message. We call *base solutions* such partial solutions; each of them consists of an expanded message and an IV such that 21 consecutive state words (one for each free message word plus the IV) follow the differential path:

Definition 10.2 (Base solution). A *base solution* for a differential path $\mathcal{P}$ of offset $i \in \{0, \dots, 64\}$ is a pair $(\mathcal{W}, I_j(\mathcal{A}) \coloneqq (\mathcal{A}_j, \dots, \mathcal{A}_{j+4}; j \in \{-4+i, \dots 11+i\})$ such that the state words $\mathcal{A}_i, \dots, \mathcal{A}_{i+20}$ entailed by $\mathcal{W}$ and the "IV" $I_j(\mathcal{A})$ satisfy all necessary conditions imposed on them by $\mathcal{P}$, *i.e.* is a partial solution for $\mathcal{P}$.

Although we may define base solutions with an offset as above, one should note that it is not directly related to the main block offset; in particular, both offsets do not need to be equal, and they are indeed different in our attacks. The reason of that difference is that the two offsets represent different things: the base solution offset defines which state words have their conditions pre-satisfied before any neutral bit is applied, while the main block offset defines which state words are modified through the action of the neutral bits. Thus we can see that we typically want the base solution to have an offset only if there are no path conditions on the lower state words that become ignored by the initialisation as a result. If this were not the case, the conditions would need to be satisfied probabilistically, likely with no accelerating technique to make that process efficient. On the other hand, the main block offset is chosen to be the one yielding the best neutral bits, and it is chiefly limited by the magnitude of the backward interactions it entails.

There is also some subtlety in how to specify the "IV" of a base solution, as it potentially includes two offsets. The first offset of an IV naturally comes from the one of the base solution it is part of, as the IV needs to be entirely included in the twenty-one words of the solution. The second offset then comes from the freedom one has, to choose which five contiguous words from the twenty-one of the base solution are to be initialised.

Let us make this more concrete by discussing the specific cases of our attacks.

The first state condition in both attacks appears on $\mathcal{A}_{-3}$, *i.e.* the second word of the original IV. Consequently, we chose a base solution offset of one in both cases. Thus, a base solution consists of state words $\mathcal{A}_{-3}, \dots, \mathcal{A}_{17}$ specified through a five-word IV and an expanded message such that all path conditions are verified by this partially computed state.

The search for a base solution consists in specifying a suitable IV $I_j(\mathcal{A})$ and then finding a message producing a valid partial solution. For instance, in the case of the 76-step attack, this process is instantiated by first choosing an initial solution $I_8(\mathcal{A})$ over $\mathcal{A}_8, \dots, \mathcal{A}_{12}$ and then extending it backward using message words $\mathcal{W}_{11}, \dots, \mathcal{W}_1$ and forward using words $\mathcal{W}_{12}, \dots, \mathcal{W}_{16}$.

For both attacks, the implementation of this search directly uses the path conditions, such as they are computed by JLCA at the end of the generation of the differential path.

Attack process. We already mentioned that the main block offset is 6 (resp. 5) for the 76-step (resp. 80-step) attack. Let us detail here the effect of this offset on the base solution and on the attack process in a bit more details, considering the 76-step case, without losing much generality.

In the course of the attack, the window of freedom for the message is made of $\mathcal{W}_6, \dots, \mathcal{W}_{21}$, and all state conditions are known to be satisfied for steps −4 to 17. The value of the message words cannot be changed entirely, because they must conform to the base solution. It is thus not possible to *e.g.* fix the value of $\mathcal{W}_{17}$ so that all conditions for $\mathcal{A}_{18}$ become satisfied. However, a certain number of neutral bits have been determined for this message window. The attack then consists in searching for good patterns of active and inactive neutral bits that allow to satisfy path conditions in state words past $\mathcal{A}_{17}$. These neutral bits were chosen so that they do not interact badly with the already fulfilled conditions, first of the base solution, and then of the further state words that are iteratively added to the partial solution, all the while allowing to try many different messages. The best of these neutral bits does not interact w.h.p. with any condition before $\mathcal{A}_{26}$ excluded; in general, for each previous step starting from $\mathcal{A}_{18}$, some fresh neutral bits are available. Past $\mathcal{A}_{26}$, the attack switches to a purely probabilistic phase.

We show in Figure 10.2 the starting point for the attack process in the 76-step case. This figure shows the state words for which the conditions were satisfied in the base solution (⊟ ◫), the message words where neutral bits are located (▩) and the remaining (untouched) free message window (▧), and the state words where the cost of satisfying the path conditions is amortised through the action of the neutral bits (⚃).

2 A framework for efficient GPU implementations of collision attacks

In the previous section, we have described the framework used to mount freestart collision attacks on SHA-1. We now turn to the matter of concrete implementation of the attack procedure. Specifically, we describe implementations on *graphics processing units* (GPUs).

The use of GPUs is attractive for computation-intensive cryptanalysis, as they offer much more raw computational power than similarly-priced general-purpose processors (CPUs). The availability of efficient frameworks for general-purpose GPU programming such as CUDA [NVIb] allows for potentially complex code to be conveniently deployed on GPUs. However, the differences in architecture between CPUs and GPUs as highlighted below need to be taken into account, and not every attack may be suitable for a GPU implementation.

GPUs have already been used with some success in heavy-computation cryptography, notably to aid in integer factorisation or in finding discrete logarithms [BK12, MBKL14, BGGM15, Jel15], and also for collision attacks on reduced SHA-1 [GA11]. Even if the

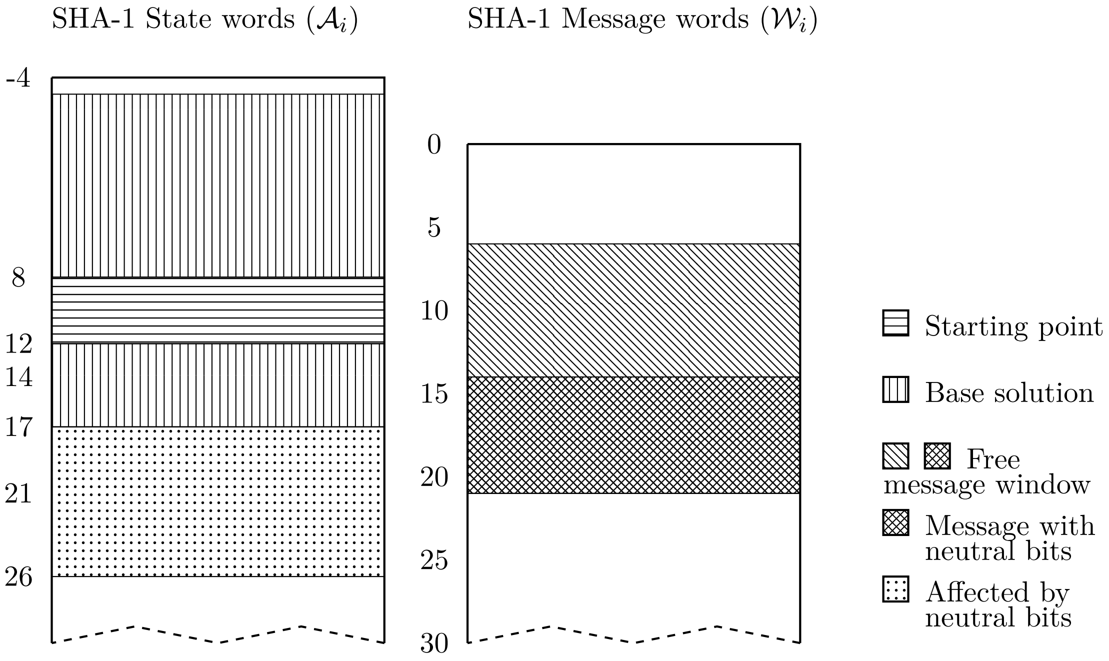

SHA-1 State words ($\mathcal{A}_i$)
SHA-1 Message words ($\mathcal{W}_i$)
-4
8
12
14
17
21
26
0
5
10
15
20
25
30
Starting point
Base solution
Free message window
Message with neutral bits
Affected by neutral bits

Figure 10.2 – The freestart attack layout made of a base solution and a message with offset.

latter case in particular is essentially identical to our own, we nonetheless developed our GPU framework for implementing collision attacks from scratch.

2.1 GPU architecture and programming model

We start by first recalling a few important points about current GPUs that will help understanding our design decisions. We specifically discuss these points for Nvidia GPUs of the *Maxwell* generation such as the GTX 970 used in our attacks.

A modern GPU can feature more than a thousand small cores, that are packed together in a small number of larger "multiprocessor" execution units. Taking the example of the Nvidia GTX 970 for concreteness, there are 13 multiprocessors of 128 cores each, making 1664 cores in total [NVIc]. The fastest instructions such as for instance 32-bit bitwise logical operations or modular addition have a throughput of 1 per core, which means that in ideal conditions 1664 instructions may be simultaneously processed by such a GPU in one clock cycle [NVIa].

Yet, so many instructions cannot be emitted independently, or to put it in another way, one cannot run an independent thread of computation for every core. In fact, threads are grouped together by 32 forming a *warp*, and only warps may be scheduled independently. Threads within a warp may have a diverging control flow, for instance by taking a different path upon encountering a conditional statement, but their execution in this case is serialised. At an even higher level, warps executing the same code can be grouped together as *blocks*.

Each multiprocessor can host a maximum of 2048 threads regrouped in at least 2 and at most 32 blocks [NVIa]. If every multiprocessor of the GPU hosts 2048 threads, we say that we have reached *full occupancy*. While a multiprocessor can only physically run one thread per core, *i.e.* 128, at a given time, a higher number of resident threads is beneficial to hide computation and memory latencies. These can have a significant impact on the performance as a single waiting thread causes its entire warp of 32 to wait with him; it is thus important in this case for the multiprocessor to be able to schedule another warp in the meantime.

Achieving full occupancy is not however an absolute objective as it may or may not result in optimal performance depending on the resources needed by every thread. Important factors in that respect are the average amount of memory and the number of registers needed by a single thread, both being resources shared among the threads. In our implementation, the threads need to run rather heavy functions and full occupancy is typically not desirable. One reason why it is so is that we need to allocate 64 registers per thread in order to prevent register spilling in some of the most expensive functions; a multiprocessor having “only” 2^{16} registers, this limits the number of threads to 1024. As a result, we use a layout of 26 blocks of 512 threads each, every multiprocessor being then able to host 2 such blocks.

In the same way as they feature many execution units, GPUs also provide memory of a generous size, *e.g.* 4 GB for the GTX 970, which must however be shared among the threads. The amount of memory available to a single thread is therefore much less than what is typically available on a CPU; though it of course highly depends on the number of running threads, it can be lower than 1 MB. This, together with the facts that threads of a same warp do not actually execute independently of each other and that threads of a same block run the same code makes it enticing to organise the memory structure of a program at the block level. Fortunately, this is made rather easy by the fact that many efficient synchronisation functions are available for the threads, both at the warp and at the block level.

2.2 High-level structure of the framework

The implementation of our attacks can be broadly decomposed into two phases. The first step consists in computing a certain number of base solutions and in storing them on disk. Because the total number of base solutions necessary to find a collision is rather small (about 2^{25} in the 76-step case, for instance) and because they can be computed quickly, this can be done efficiently in an offline fashion using CPUs.

The second phase then consists in trying to extend probabilistically the base solutions to satisfy path conditions up to a further point by trying many neutral bit combinations, in the hope of eventually finding a collision. This is an intensely parallel task that is well suited to running on GPUs. However, as it was emphasised above, GPUs are most efficient when there is a high coherency between the execution of many threads. For that reason, we must avoid having idle threads that are waiting because their candidate solutions failed to follow the differential paths, while others keep on verifying a more successful one. Our approach to this is to fragment the verification into many small functions or *snippets* that are chosen in a way which ensures that coherency is maintained for every thread of a warp when executing a single snippet, except in only a few small points. This is achieved through a series of intermediary buffers that store inputs and outputs for the snippets. A warp then only executes a given snippet if enough inputs are available for every of its threads. One should note that there is no need to entirely decompose the second step of the attack into snippets, and that a final part can again be run in a more serial fashion, typically on CPU. Indeed, if inputs to such a part are scarce, there is no real advantage in verifying them in a highly parallel way.

The sort of decomposition used for the GPU phase of our attack as described above is in no way constrained by the specific context of a SHA-1 collision search. In fact, it is quite general, and we believe that it can be successfully applied to many an implementation of (symmetric) cryptographic attacks. We conclude this section by giving more details of the application of this approach to the case of SHA-1.

As we mentioned in Section 1, the attack process consists in trying many combinations of neutral bits, with each step in a small window adding new neutral bits to be tested. It is thus quite natural to reflect this process in the choice of the snippets: we use intermediary buffers to store partial solutions up to $\mathcal{A}_{17}$ (*i.e.* base solutions), $\mathcal{A}_{18}$, etc. Then for each step the corresponding snippet consists in loading one partial solution per thread of a warp and applying every possible combination of neutral bits for this step. Each combination is tried by every thread at the same time on its own partial solution, thereby maintaining coherency. Then, each thread assesses if the current combination yields a valid extension by one step of its own partial solution, and writes the result to an output buffer for the snippet, which is the input buffer for the next snippet, if this is the case; this conditional write is the only part of the code where threads may briefly diverge.

For the later steps when no neutral bits can be used anymore, the snippets regroup the computation of several steps together. Eventually the verification that partial solutions up to step 56 (in the 76-step case; 60 in the 80-step one) result in valid collisions is done on a CPU. This is partly because the amount of available memory makes it hard to use step-by-step snippets until the end, but also because such partial solutions are only produced very slowly. For instance, a single GTX 970 produces partial solutions up to step 56 of a 76-step collision at a speed of about 0.017 solution per second, that is about 1 per minute; waiting for enough partial solutions to feed a single complete warp would in this case take a completely unreasonable half hour.

A complete implementation of an attack mostly consists in the snippets and supporting functions, such as buffer management. Connecting the snippets together is straightforward. Every warp tries to work with partial solutions that are up to the latest step for which enough solutions are available. This means that it visits the buffer of partial solutions in order from the top, stopping at the first that is able to feed it entirely. In the worst case where none of the buffers are full enough, it simply resorts to using base solutions.

In practice, warps spend most of their time feeding on partial solutions that are valid up to a rather late step. For instance, in the 76-step attack, about 90% of the time is spent on $\mathcal{A}_{24}$ or higher, which is at most two steps away from the latest one where neutral bits are available. Thus, work on early steps and in particular on base solutions is done only intermittently.

We conclude this high-level description by giving a simplified flow chart of the GPU part of the 76-step attack in Figure 10.3, made slightly incorrect for the sake of clarity, notably omitting the fact that further verification is still done on GPU up to steps 56.

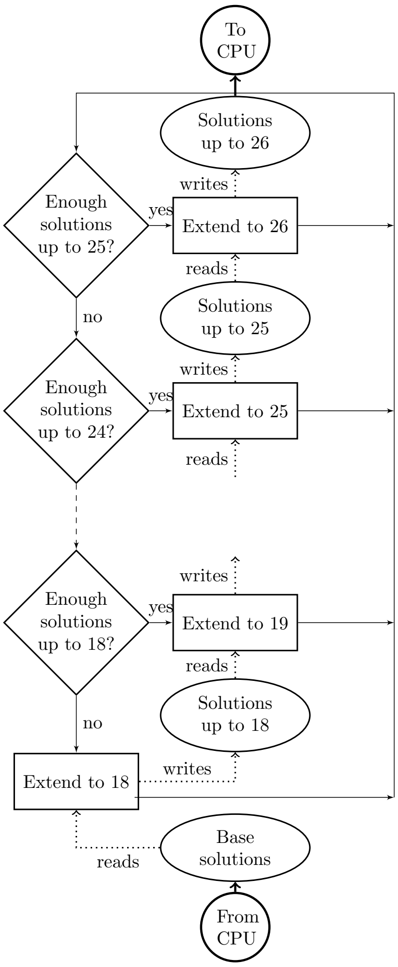


Figure 10.3 – Simplified flow chart for the GPU part of the attack. The start of this infinite loop is in the top left corner. Rectangles "□" represent snippets, ellipses "⬭" represent shared buffers, plain lines "➚" represent control flow, and dotted lines "⇡" represent data flow.

2.3 Implementation details

We now give more details about the implementations of the attacks. In particular, we discuss how partial solutions are represented at various steps in Section 2.3.1 and we comment the code of a snippet function in Section 2.3.2. We very briefly discuss how to tune GPU settings to use them more efficiently in Section 2.3.3.

2.3.1 Representation of partial solutions

As the basis of our framework is to store, load and extend many partial solutions for the differential path, we need to be able to work with such objects in an efficient way. One thus needs to define good representations for partial solutions, such that processing them is computationally simple, and managing them in memory causes as little overhead as possible.

The representation we use is based on two types of buffers: some are holding enough information to define a SHA computation in its entirety, *i.e.* it contains the value of at least five (resp. sixteen) consecutive state (resp. message) words, while others only contain the necessary information to express how the associated partial solution differs from a completely-defined one. Typically, the first kind of buffer may be used to store base solutions, while the second is used to keep track of which neutral bits are active in a partial solution.

It is quite straightforward to define the structure of a base solution buffer, as there is little doubt about the necessary information they need to include and the way to represent it. For instance, the 76-step base solution buffer contains the value of state words $\mathcal{A}_{13},\ldots,\mathcal{A}_{17}$ and of the message words $\mathcal{W}_6,\ldots,\mathcal{W}_{21}$. Additionally, it includes the value of $\mathcal{A}_{12}$; while this information is not strictly necessary, it is useful to speed-up the computation of the activation of some neutral bits. Similar buffers without an extra state word are used to hold partial solutions up to $\mathcal{A}_{36}$ and $\mathcal{A}_{56}$ (resp. $\mathcal{A}_{40}$ and $\mathcal{A}_{60}$ in the 80-step case). The choice of these boundaries to define the late partial solutions where no more control is possible simply comes from the fact that a pair of messages following the differential path of either attack is not expected to have any state differences at these steps. It is thus particularly efficient to filter solutions that are not valid at these points.

The structures of the other buffers are also quite simple, but their instantiations may require some care to make them especially efficient. In a nutshell, we just want such a buffer to keep a reference to a base solution and to remember the values of the currently active neutral bits. To make this efficient and convenient to use, we would like to share a similar structure for the input and output buffers of a snippet; this would allow to extend a partial solution that possibly already has some bits active by simply adding the newly activated bits for this step. This is the approach we followed in our implementations, with an added refinement.

We have already mentioned in the past section that *e.g.* the 76-step attack contains neutral bits acting on a wide range of steps, from $\mathcal{A}_{18}$ to $\mathcal{A}_{26}$; the range for the 80-step attack is even wider, due to the use of boomerangs. Additionally, the neutral bits themselves are located on many message words. Thus, it would seem wasteful to recompute the action of past neutral bits on message words as low as $\mathcal{W}_{14}$ while in the snippet corresponding *e.g.* to $\mathcal{A}_{24}$. Consequently, we have a strong incentive to store intermediary partial solutions in order to save some of this recomputation.

In the 76-step case, there is a natural location that may be used to define what we call *extended base solutions*. As we will detail in Section 3, neutral bits located on words $\mathcal{W}_{14}$ to $\mathcal{W}_{18}$ are only used up to $\mathcal{A}_{21}$, and the ones located on words $\mathcal{W}_{19}$ to $\mathcal{W}_{21}$ are only used in later steps. Thus, it would make sense that once the active bits up to $\mathcal{W}_{18}$ have all been determined, only the modified message words and the corresponding value for the state should be stored. There is however no need to keep again sixteen message words in such an extended base solution, as most of them are identical to the ones of the corresponding base solution, and as a base solution is in general extended into many distinct extended base solutions, it would not make sense to *e.g.* add enough message words to the latter and erase the former. One subtlety in the definition of this extended base solution is that it also includes the message word $\mathcal{W}_{20}$. Although no neutral bits on this word have been activated at the point where the solution is formed, some of its bits may need to be flipped depending on the use of neutral bits on words $\mathcal{W}_{15}$ and $\mathcal{W}_{16}$, so as to preserve message bit relations. A convenient way to remember this information is simply to preemptively add the possible contributions of the neutral bits to $\mathcal{W}_{20}$ and to store this modified word in the extended base solution. All in all, the buffer of extended base solution of the 76-step attack is made of twelve words: five state words $\mathcal{A}_{17}$ to $\mathcal{A}_{21}$, six message words $\mathcal{W}_{14}$ to $\mathcal{W}_{18}$ and $\mathcal{W}_{20}$, and one word holding an identifier for the base solution from which it is extended.

The inter-snippet buffers that refer to changes from a base or extended base solution are only made of two words consisting of concatenated segments of the message words containing neutral bits and of a reference to the associated solution.

The 80-step attack uses similar representations but with a few variations. As we will show in Section 4, there are more possible corrections on the messages to be performed in the 80-step attack to preserve message bit relations. Consequently, some of the precomputations of individual neutral bit contributions are stored alongside the actual neutral bits. It is also slightly less immediate to determine where to start defining an extended base solution, as there is no natural separation between the location of various neutral bits as there was in the 76-step case. This is not a major issue, however, as the location of the neutral bits of a same word shared between base and extended base solutions are not overlapping inside the word itself; splitting their representation over multiple buffers thus does not result in significant overhead. Finally, the additional use of boomerang neutral bits, or somehow equivalently the use of more neutral bits than in the 76-step attack, implies that the last inter-snippet buffers contain one more word compared to the ones for early snippets and all such buffers in the 76-step case, *i.e.* three in total.

In Section 3 and Section 4, we will describe the full content of most of the inter-snippet buffers.

We conclude this part by discussing some implementation aspects of the various buffers.

All of the buffers are cyclic and hold 2^{20} elements, regardless of their sizes, except the buffers of partial solutions extended up to $\mathcal{A}_{36} \sim \mathcal{A}_{40}$ and $\mathcal{A}_{56} \sim \mathcal{A}_{60}$ which only have 2^{10} elements as they see a lower production rate due to their purely probabilistic nature.

With the exception of the buffers holding the base solutions and the collision candidates formed by partial solutions up to $\mathcal{A}_{56} \sim \mathcal{A}_{60}$, *i.e.* the buffers that are written or read by a CPU, there is one instance of every buffer per block, *i.e.* 26 buffers per GPU. This allows to use block-wise instead of global synchronisation mechanisms when updating the buffers' content, thence reducing the overhead inherent to the use of such shared data structures. Taken together, the buffers thus use a significant portion of the 4 GB memory available on the GTX 970, needing in the neighbourhood of 3 GB.

We carefully took into account the presence of a limited amount of very fast multiprocessor-specific shared memory. While the 96 KB available per multiprocessor is hardly enough to store the whole buffers themselves, we take advantage of it by dissociating the storage of the buffers and of the meta-data used for their control logic, the latter being held in shared memory. This improves the overall latency of buffer manipulations, especially in case of heavy contention between different warps. This local shared memory is also very useful to buffer the writes to the buffers themselves. Indeed, only a fraction of the threads of a warp often as low as 2^{-3} have a valid solution to write after having tested a single candidate, and the more unsuccessful threads need to wait while the former write their solution to global memory. It is therefore beneficial to first write the solutions to a small local warp-specific buffer and to flush it to the main block-wise buffer as soon as it holds 32 solutions or more, thence significantly reducing the number of accesses to the slower global memory.

2.3.2 An example of snippet function

We now illustrate the discussion of this section by providing a commented snippet from the 76-step attack given in Figure 10.4 written in CUDA C/C++, namely a function that is taking partial solutions up to $\mathcal{A}_{22}$ and that is trying to extend them up to $\mathcal{A}_{23}$ using neutral bits on $\mathcal{W}_{19}$. Its structure is a straightforward application of the framework, and is fairly representative of most of the code of both of our attacks although some snippets may at first seem more complex due to their use of neutral bits located on several distinct message words.

```
__device__ void stepQ23(uint32_t thread_rd_idx)
{
        uint32_t base_idx = Q22SOLBUF.get<11>(thread_rd_idx);
        uint32_t q17 = Q22SOLBUF.get<0>(thread_rd_idx);
        uint32_t q18 = Q22SOLBUF.get<1>(thread_rd_idx);
        uint32_t q19 = Q22SOLBUF.get<2>(thread_rd_idx);
        uint32_t q20 = Q22SOLBUF.get<3>(thread_rd_idx);
        uint32_t q21 = Q22SOLBUF.get<4>(thread_rd_idx);
        uint32_t m6  = BASESOLBUF.get<6>(base_idx);
        uint32_t m8  = BASESOLBUF.get<8>(base_idx);
        uint32_t m19 = BASESOLBUF.get<19>(base_idx);
        uint32_t m21 = BASESOLBUF.get<21>(base_idx);
        uint32_t m14 = Q22SOLBUF.get<5>(thread_rd_idx);
        uint32_t m22;

        uint32_t q22  = sha1_round2(q21, q20, q19, q18, q17, m21);

        uint32_t w19_q23_nb = 0;
        for (unsigned i = 0; i < 32; i++)
        {
                NEXT_NB(w19_q23_nb, W19NBQ23M);

                m19 &= ~W19NBQ23M;
                m19 |= w19_q23_nb;
                m22 = sha1_mess(m19, m14, m8, m6);

                uint32_t newq20 = q20 + w19_q23_nb;
                uint32_t newq21 = q21 + rotate_left(w19_q23_nb, 5);
                uint32_t newq22 = sha1_round2(newq21, newq20, q19, q18,
 ↪  q17, m21);
                uint32_t newq23 = sha1_round2(newq22, newq21, newq20,
 ↪  q19, q18, m22);

                uint32_t q23nessies = Qset1mask[QOFF + 23] ^ (Qprevrmask
 ↪  [QOFF + 23] & rotate_left(newq22, 30));
                bool valid_sol = (0 == ((newq21 ^ q21) & Qcondmask[QOFF
 ↪  + 21]));
                valid_sol &= (0 == ((newq22 ^ q22) & Qcondmask[QOFF +
 ↪  22]));
                valid_sol &= (0 == ((newq23 ^ q23nessies) &
 ↪  Qcondmask[QOFF + 23]));

                uint32_t sol_val_0 = pack_update_q23_0(m19);
                uint32_t sol_val_1 = pack_update_q23_1(thread_rd_idx);

                WARP_TMP_BUF.write2(valid_sol, sol_val_0, sol_val_1,
 ↪  Q23SOLBUF, Q23SOLCTL);
        }
        WARP_TMP_BUF.flush2(Q23SOLBUF, Q23SOLCTL);
}
```

Figure 10.4 – The *stepQ23* function from the 76-step attack.

This function takes as argument a thread-dependent identifier thread_rd_idx for a partial solution, which is essentially an index for the buffer Q22SOLBUF of solutions up to $\mathcal{A}_{22}$.

The partial solution is loaded from the buffer and reconstructed from lines 3 to 16. This buffer being the first one holding an extended base solution, there is no need to reapply neutral bits and most of the work simply consists in loading the appropriate state and message words either directly from the extended base solution buffer, or from the matching solution of the base solution buffer BASESOLBUF, the index of which is recovered on line 3. The only recomputation performed here is the one of q22 on line 16, using the SHA-1 round function. This would in fact not be necessary in this function, as the value could have been included in the extended base solution altogether. This was not done because recomputing q22 is necessary in the snippets of the following steps and causes only minimal overhead in this one, while saving a 32-bit word from the Q22SOLBUF buffer.

The loop from lines 18 to 41, *i.e.* the remainder of the function, applies every combination of the five neutral bits for this step, all of which are located on $\mathcal{W}_{19}$. In more details, line 21 sets the register w19_q23_nb to one of the 32 possible combinations. Lines 23 and 24 clear the message word m19 of the previous neutral bit combination and applies the new one, and line 25 computes the expanded message word m22 based on the new value for m19, using SHA-1's message expansion. Note that this computation could actually be optimised, *e.g.* by precomputing the contribution of m14, m8 and m6, which are fixed in this function. Lines 27 to 30 compute the impact of the current neutral bit combination, first by partially recomputing the state words newq20 and newq21, then fully recomputing newq22, and finally computing the as yet unknown word newq23. Line 32 computes a mask of sufficient conditions for the value of newq23 based on the value of newq22. Lines 33 to 35 determine if the current combination of neutral bits lead to a valid partial solution for $\mathcal{A}_{23}$, first by comparing the values of q21 and q22 which we know to fulfill the conditions with the updated values newq21 and newq22 (lines 33 and 34), and then checking newq23 for the previously computed conditions. Lines 37 and 38 prepare the description of the partial solution. Finally, line 40 writes the partial solution to the buffer Q23SOLBUF, at the condition that it is indeed valid. As mentioned at the end of Section 2.3.1, this is done through a warp-specific temporary buffer, which is flushed to the actual buffer on line 42 to ensure that every valid solution is indeed eventually copied to Q23SOLBUF.

2.3.3 GPU tuning

After our initial implementation of the 76-step attack, we did some fine tuning of the GPU BIOS settings in order to try improving the performance. One first objective was to ensure that the GPU fans work at 100% during the attack, as this was strangely not the case initially for our particular boards, and was obviously not ideal for cooling. We also experimented with various temperature limits that define when the GPU will start to throttle and both over-clocking and under-volting.

Taken together, these variations had a significant impact on the overall performance of the program. For a single GPU, the initial setting resulted in an estimated time of 4.94 days to produce one 76-step freestart collision. We were able to eventually reduce this to 4.16 days. We also set-up machines with four GPUs, and observed significant performance variations across the GPUs. This was likely due to uneven heat dissipation, as was shown by the different temperatures reached by every board. All in all, the average expected time to obtain a collision on a single 4-GPU machine was 4.42 days per GPU, and thus about 1.1 day per machine. As the cooling was not optimal in our set-up, reaching a performance closer to the one-GPU setting is likely to be possible.

We did not make any major changes for the 80-step attack. However, due to the higher computational cost, we ran the attack only once. The collision was found after ten days, with the attack being run on sixteen 4-GPU machines.

2.4 Efficiency of the framework

We conclude this section by evaluating the performance of our framework, and in particular by assessing the relative efficiency of a GPU-based attack compared to a more traditional CPU implementation. We provide this analysis for the 76-step attack, but the results apply entirely to the 80-step attack as well and to would-be SHA-1 collision attacks implemented with our framework in general.

Our GPU implementation of SHA-1 can compute about $2^{31.8}$ full SHA-1 compression functions per second on a GTX 970. Comparing this with the expected time to find a collision of 4.16 days, this means that the 76-step attack has a complexity equivalent to $2^{50.25}$ calls to the compression function for the best-performing GPU; this increases slightly to $2^{50.34}$ when considering the 4-GPU average of 4.42 days.

Comparatively, on a Haswell Core-i5 running at 3.2 GHz, the OpenSSL implementation of SHA-1 can compute $2^{23.47}$ compression functions per second on one core. A CPU implementation of our attack on the same processor leads to an expected time to collision of 606.12 core-days, which translates to a complexity of $2^{49.1}$ compression function calls, though this could probably be improved by vectorizing part of the CPU implementation. This means that a single GTX 970 is worth 322 such CPU cores when computing the SHA-1 compression function, and 138 cores when running our attack program; this increases to 146 for the best-performing GPU. While this drop in relative efficiency was to be expected, it is somehow surprisingly small given the complexity of our implementation and *e.g.* the intensive use of large shared data structures. Our careful implementation thus gives a much better value for the GPUs when compared to previous attempts at running collision attacks on such a platform: in their work, Grechnikov and Adinetz estimated a GPU to be worth 39 CPU cores [GA11]. This comparison should however be modulated by considering possibly uneven progress in GPUs and CPUs since 2011 and different hardware quality. We believe that the gap between these and our results is nonetheless significant.

3 Freestart collisions for 76-step SHA-1

This section gives the attack parameters for the 76-step freestart collision attack on SHA-1.

3.1 The differential path for most of the first two rounds

We give a graphical representation of the differential path used in our attack up to step 29 in Figure 10.5. The meanings of the various symbols are defined in Table 8.2. The remainder of the path up to step 76 can easily be determined by linerisation of the step function, given the differences in the message.

The message bit relations used in the attack for message words past $\mathcal{W}_{35}$ are given in Figure 10.6, together with a graphical representation in Figure 10.7.

3.2 The neutral bits

We give here the list of the neutral bits used in the attack. There are fifty-one of them over the eight message words $\mathcal{W}_{14}$ to $\mathcal{W}_{21}$, distributed as follows:

- $\mathcal{W}_{14}$: 9 neutral bits at bit positions (starting with the least significant bit (*LSB*) at zero) 5,6,7,8,9,10,11,12,13
- $\mathcal{W}_{15}$: 11 neutral bits at positions 5,6,7,8,9,10,11,12,13,14,16
- $\mathcal{W}_{16}$: 8 neutral bits at positions 6,7,8,9,10,11,13,16
- $\mathcal{W}_{17}$: 5 neutral bits at positions 10,11,12,13,19
- $\mathcal{W}_{18}$: 2 neutral bits at positions 15,16
- $\mathcal{W}_{19}$: 8 neutral bits at positions 6,7,8,9,10,11,12,14
- $\mathcal{W}_{20}$: 5 neutral bits at positions 0,6,11,12,13
- $\mathcal{W}_{21}$: 3 neutral bits at positions 11,16,17

We give a graphical representation of the repartition of these neutral bits in Figure 10.8.

Not all of the neutral bits located on the same word (say $\mathcal{W}_{14}$) are neutral up to the same state word. Their repartition in that respect is as follows

- Bits neutral up to step 18 (excluded): $\mathcal{W}_{14}$[9,10,11,12,13], $\mathcal{W}_{15}$[14,16]
- Bits neutral up to step 19 (excluded): $\mathcal{W}_{14}$[5,6,7,8], $\mathcal{W}_{15}$[8,9,10,11,12,13], $\mathcal{W}_{16}$[13,16], $\mathcal{W}_{17}$[19]
- Bits neutral up to step 20 (excluded): $\mathcal{W}_{15}$[5,6,7], $\mathcal{W}_{16}$[9,10,11]
- Bits neutral up to step 21 (excluded): $\mathcal{W}_{16}$[6,7,8], $\mathcal{W}_{17}$[10,11,12,13], $\mathcal{W}_{18}$[15,16]

i	$\mathcal{A}_i$	$\mathcal{M}_i$
-4	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-3	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-2	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○☆▲○	
-1	▽○○○○○○○○○○○○○○○○○○○○○○○○△○○○○○▲	
0	△▽○▽○○○○○○○○○○○○○○○○○○○○○▽○▽○○○○	○○○○○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
1	○▽○△○○○○○○○○○○○○○○○○☆○▽○○○○▲○○○○	○○○○○▼○○○○○○○○○○○○○○○○○○○○○▼▲▲○○
2	○○○☆○▼○○○○○○○○○☆○△○○○○▲○○○○△○▼○△	▼▲○○▼▲○○○○○○○○○○○○○○○○○○○○○▲○▼○○
3	○○○○○▼○△○○☆○△○○○○▲○○○○△○▼○○○☆▼○▽	○○○○▲▼○○○○○○○○○○○○○○○○○○○○○○○○▲○
4	○▼○△○○○△△○▽○▲○○○○▽▽▼△△○○▼○☆○○▽▲△	▲▼○○○○○○○○○○○○○○○○○○○○○○○○○▼○○○○
5	△▲○▽○☆○▽▽☆△△△○▼○△▽△▼▽▽○△○○▲○▲△○▽	▲○▲▲○▼○○○○○○○○○○○○○○○○○○○○○▼▲▼○○
6	▽▽○▼☆▼△▼△▼▼▲▲▲▲☆▽△▲▽▼○▼▼○△△○△○▽▼	○○▲▼▲▲○○○○○○○○○○○○○○○○○○○○○○○▲○○
7	△▲○▼☆▲▼▽▼▲△▽▲▼▽☆▲▲▼▲△☆△○▽▲○○▽○△△	▲○▲▼▼▼○○○○○○○○○○○○○○○○○○○○○▲▼○▼○
8	▲▼○△○○▲▼▼▼▼▼▼▼▼▼▼▼▼▼△▼▲▲△△○▼○○▲▽	○○▼○○○○○○○○○○○○○○○○○○○○○○○○▼○○○○
9	▼▼▲▽○△○▽△▽○△▽▽▽○△▽▲▼○▽○△▽▽▼△○△▽○	○○▲○○▲○○○○○○○○○○○○○○○○○○○○○▼▲▲○○
10	▼△▽○△○○○△▽▽▽△△△▽△△▽▽△○○△△△○○△▽○○	▼▼▲○▲▼○○○○○○○○○○○○○○○○○○○○○▼○▲○○
11	△○○▼△○○○○○○○○○○○○○○○△▽○○○○○○△○☆○	○○○○▼▲○○○○○○○○○○○○○○○○○○○○○○○○▼○
12	▲○▲○○△○○○○○○○○○○○○○○○○○○☆○○○○○○○	▼▲○○○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
13	▽○▲○△▽○○○○○○○○○○○○○○○○○○○○▼○○○○▽	▲○▼▼○▼○○○○○○○○○○○○○○○○○○○○○▲▼▼○○
14	○○▲○○○○○○○○○○○○○○○○○○○○○○○○▽○○△	○○▼○▼▲○○○○○○○○○○○○○○○○○○○○○○○▼○○
15	△▼○○○○○○○○○○○○○○○○○○○○○○○○○○△○★○	▲○▼▲▲▲○○○○○○○○○○○○○○○○○○○○○▲▼○○○
16	▲○○△▽○○○○○○○○○○○○○○○○○○○○○○○○○☆○	▼○▼▲○○○○○○○○○○○○○○○○○○○○○○○▼○○○○
17	▲○▼△○○○○○○○○○○○○○○○○○○○○○○○○○○★○	○○○○○○○○○○○○○○○○○○○○○○○○○○○○▼▲○○
18	▲○○○△○○○○○○○○○○○○○○○○○○○○○○○○○○☆	▲○▲▼▲○○○○○○○○○○○○○○○○○○○○○○▼○○○○
19	○▼○○◆○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○▼○○○○○○○○○○○○○○○○○○○○○○▲▼○○○
20	▼○◆■○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○▲▲▲▲○○○○○○○○○○○○○○○○○○○○○○▲○○○○
21	▼○▼◆○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○▼○○○○○○○○○○○○○○○○○○○○○○▲○▲○○
22	▼○○○□○○○○○○○○○○○○○○○○○○○○○○○○○○○	○▼▲▲○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
23	▼○▲○◇○○○○○○○○○○○○○○○○○○○○○○○○○○○	▼○▲▼▲○○○○○○○○○○○○○○○○○○○○○○▲▼▲○○
24	○○○○■○○○○○○○○○○○○○○○○○○○○○○○○○○○	▼▼▲○▲○○○○○○○○○○○○○○○○○○○○○○○○○○○
25	○○▼○◆○○○○○○○○○○○○○○○○○○○○○○○○○○○	▼○▲▲○○○○○○○○○○○○○○○○○○○○○○○○○▲○○
26	▼○○○■○○○○○○○○○○○○○○○○○○○○○○○○○○☆	○▼○▲▼○○○○○○○○○○○○○○○○○○○○○○▲○○○○
27	○▼▼○◇○○○○○○○○○○○○○○○○○○○○○○○○○○○	▲○▲▼○○○○○○○○○○○○○○○○○○○○○○○○▲▲○○
28	○○◇■■○○○○○○○○○○○○○○○○○○○○○○○○○○○	○▲○○▲○○○○○○○○○○○○○○○○○○○○○○○○○○○
29	○○○◇◇○○○○○○○○○○○○○○○○○○○○○○○○○○○	

Figure 10.5 – The differential path of the 76-step attack up to step 29.

$$\begin{array}{lll}
\mathcal{W}_{54}[29] \oplus \mathcal{W}_{55}[29] = 0 & \mathcal{W}_{53}[29] \oplus \mathcal{W}_{55}[29] = 0 & \mathcal{W}_{51}[4] \oplus \mathcal{W}_{55}[29] = 0 \\
\mathcal{W}_{49}[29] \oplus \mathcal{W}_{50}[29] = 0 & \mathcal{W}_{48}[29] \oplus \mathcal{W}_{50}[29] = 0 & \mathcal{W}_{47}[28] \oplus \mathcal{W}_{47}[29] = 1 \\
\mathcal{W}_{46}[4] \oplus \mathcal{W}_{50}[29] = 0 & \mathcal{W}_{46}[28] \oplus \mathcal{W}_{47}[28] = 0 & \mathcal{W}_{46}[29] \oplus \mathcal{W}_{47}[28] = 1 \\
\mathcal{W}_{45}[28] \oplus \mathcal{W}_{47}[28] = 0 & \mathcal{W}_{44}[29] \oplus \mathcal{W}_{44}[30] = 0 & \mathcal{W}_{43}[3] \oplus \mathcal{W}_{47}[28] = 0 \\
\mathcal{W}_{43}[4] \oplus \mathcal{W}_{47}[28] = 1 & \mathcal{W}_{42}[29] \oplus \mathcal{W}_{47}[28] = 1 & \mathcal{W}_{41}[4] \oplus \mathcal{W}_{47}[28] = 0 \\
\mathcal{W}_{40}[29] \oplus \mathcal{W}_{47}[28] = 0 & \mathcal{W}_{39}[4] \oplus \mathcal{W}_{47}[28] = 1 & \mathcal{W}_{37}[4] \oplus \mathcal{W}_{47}[28] = 0 \\
\mathcal{W}_{59}[4] \oplus \mathcal{W}_{63}[29] = 0 & \mathcal{W}_{57}[4] \oplus \mathcal{W}_{59}[29] = 0 & \mathcal{W}_{74}[0] = 1 \\
\mathcal{W}_{75}[5] = 0 & \mathcal{W}_{73}[1] \oplus \mathcal{W}_{74}[6] = 1 & \mathcal{W}_{71}[5] \oplus \mathcal{W}_{75}[30] = 0 \\
\mathcal{W}_{70}[0] \oplus \mathcal{W}_{75}[30] = 1 & &
\end{array}$$

Figure 10.6 – The message bit relations of the 76-step attack for message words $\mathcal{W}_{36}$ to $\mathcal{W}_{75}$.

- Bits neutral up to step 23 (excluded): $\mathcal{W}_{19}$[9,10,11,12,14]
- Bits neutral up to step 24 (excluded): $\mathcal{W}_{19}$[6,7], $\mathcal{W}_{20}$[11,12], $\mathcal{W}_{21}$[16,17]
- Bits neutral up to step 25 (excluded): $\mathcal{W}_{19}$[8], $\mathcal{W}_{20}$[6,13], $\mathcal{W}_{21}$[11]
- Bits neutral up to step 26 (excluded): $\mathcal{W}_{20}$[0]

We also give a graphical representation of this repartition in Figure 10.9.

Finally, on the implementation side, we show how the neutral bits are packed together inside the inter-snippet buffers from Section 2.3.1 in Figure 10.10 and Figure 10.11. These figures represent each of the two words of the buffers as thirty-two numbered coloured circles, one for each bit. The colours represent the original message word from which the bit comes from, and the number is the bit position in this original word. For instance, in Figure 10.10, ⑬ is the value of the thirteenth bit in some instance of $\mathcal{W}_{14}$, *i.e.* $\mathcal{W}_{14}[13]$; the fact that this circle is the leftmost one in the top sequence means that in the buffer, this information is stored as the MSB of the first word. One should note that not all consecutive bits of a message are neutral; non-neutral bits do not need to be stored, but it is nonetheless useful to maintain the relative distance between the actual neutral bits inside the packing. For instance, there is no neutral bit on $\mathcal{W}_{15}[15]$, thus this bit is not included *per se* in Figure 10.10, but a padding bit is added instead. This is shown as ⊙.

3.3 An example of colliding message pair

We give an example of collision in Table 10.1. This shows the two (message, IV) pairs with their identical resulting digest. The IVs and the digests' words are ordered similarly as in Table 8.3; the messages' words are ordered as $\mathcal{M}_0, \ldots, \mathcal{M}_{15}$ from top to bottom, left to right. Although the separations between the 32-bit words are not materialised, they should be taken into account, and the values should not be interpreted as binary strings.

```
W36: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W37: . . . . . . . . . . . . . . . . . . . . . . . . . . . a . . . .
W38: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W39: . . . . . . . . . . . . . . . . . . . . . . . . . . . A . . . .
W40: . . a . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W41: . . . . . . . . . . . . . . . . . . . . . . . . . . . a . . . .
W42: . . A . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W43: . . . . . . . . . . . . . . . . . . . . . . . . . . . A a . . .
W44: . b b . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W45: . . . a . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W46: . . A a . . . . . . . . . . . . . . . . . . . . . . . c . . . .
W47: . . A a . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W48: . . c . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W49: . . c . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W50: . . c . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W51: . . . . . . . . . . . . . . . . . . . . . . . . . . . d . . . .
W52: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W53: . . d . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W54: . . d . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W55: . . d . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W56: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W57: . . . . . . . . . . . . . . . . . . . . . . . . . . . e . . . .
W58: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W59: . . e . . . . . . . . . . . . . . . . . . . . . . . . f . . . .
W60: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W61: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W62: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W63: . . f . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W64: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W65: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W66: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W67: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W68: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W69: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W70: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . g
W71: . . . . . . . . . . . . . . . . . . . . . . . . . . G . . . . .
W72: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W73: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . h .
W74: . . . . . . . . . . . . . . . . . . . . . . . . . H . . . . . 1
W75: . G . . . . . . . . . . . . . . . . . . . . . . . . 0 . . . . .
```

Figure 10.7 – The message bit-relations used in the attack for words $\mathcal{W}_{36}$ to $\mathcal{W}_{75}$ (graphical representation). A dot ("`.`") means an absence of condition. A zero ("`0`") or one ("`1`") character represents a bit unconditionally set to 0 or 1. A pair of two identical letters x means that the two bits have the same value. A pair of two letters x and X means that the two bits have different values.

Note that unlike most inputs discussed in this chapter, the *IV* values are compatible with the *original* description of SHA-1, *i.e.* they are not (all) equal to the corresponding state values $\mathcal{A}_0, \ldots, \mathcal{A}_{-4}$; compared to $\mathcal{A}_{-2}, \ldots, \mathcal{A}_{-4}$, the last three *IV* words are rotated by two to the right.

3.4 Complexity of the attack

As we already mentioned in Section 2.3.3, the expected time to find a collision on a single average-performing GTX 970 is 4.42 days. This figure is obtained by considering the rate at which partial solutions up to $\mathcal{A}_{56}$ are produced together with the probability

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○●●●●●●●●●○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○●○●●●●●●●●●●○○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○●○○●○●●●●●●○○○○○○

$\mathcal{W}_{17}$: ○○○○○○○○○○○○●○○○○○●●●●○○○○○○○○○○

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○●●○○○○○○○○○○○○○○○

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○●○●●●●●●●○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○●●●○○○○●○○○○○●

$\mathcal{W}_{21}$: ○○○○○○○○○○○○○○●●○○○○●○○○○○○○○○○○

Figure 10.8 – The fifty-one neutral bits of the 76-step attack, using (with some abuse) a "difference" notation. A "○" (resp. "●") symbol means the absence (resp. presence) of a neutral bit on the corresponding bit. The message words are (as usual) written left to right from MSB to LSB.

Table 10.1 – A freestart collision for 76-step SHA-1. Message and IV bytes with differences are highlighted with $\boxed{\text{coloured boxes}}$.

	Message 1
IV	81 bf 23 06 41 b8 3b 5c 03 e9 a7 8f ba 50 28 d5 fc 50 87 88
$\mathcal{M}$	46 fa 5a 88 f4 f0 c7 f0 b8 de db ec 95 1e 25 88
	77 34 fd f5 4c 42 c4 97 52 d7 d8 f9 5f 14 52 ea
	b4 9e 93 b2 91 c2 30 71 c7 0f 35 9b 8a ba cf af
	b3 7f fb 27 3d fe 7f ad 7a de 56 95 20 fd 7c ea
H(IV, $\mathcal{M}$)	af 49 5d 10 52 82 35 03 e4 9e 46 78
	dc e7 f3 b3 d6 da a3 24

	Message 2
ΔIV	81 bf 23 06 41 b8 3b 5d 83 e9 a7 8f ba 50 28 d5 fc 50 87 88
$\Delta\mathcal{M}$	46 fa 5a 98 f0 f0 c7 ec 7a de db f8 99 1e 25 8a
	b7 34 fd e5 f8 42 c4 8b 6e d7 d8 fd e3 14 52 f0
	94 9e 93 a2 b5 c2 30 6d 2b 0f 35 8f 86 ba cf ad
	73 7f fb 37 89 fe 7f b1 56 de 56 91 9c fd 7c f2
H(ΔIV, $\Delta\mathcal{M}$)	af 49 5d 10 52 82 35 03 e4 9e 46 78
	dc e7 f3 b3 d6 da a3 24

$\mathcal{A}_{18}$

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○●●●●●○○○○○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○●○●○○○○○○○○○○○○○○

$\mathcal{A}_{19}$

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○○○○○○●●●●○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○○○○●●●●●●○○○○○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○●○○●○○○○○○○○○○○○○

$\mathcal{W}_{17}$: ○○○○○○○○○○○○●○○○○○○○○○○○○○○○○○○○

$\mathcal{A}_{20}$

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○○○○○○○○○○●●●○○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○○○○○○●●●○○○○○○○○○

$\mathcal{A}_{21}$

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○○○○○○○○○●●●○○○○○○

$\mathcal{W}_{17}$: ○○○○○○○○○○○○○○○○○○●●●●○○○○○○○○○○

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○●●○○○○○○○○○○○○○○○

$\mathcal{A}_{23}$

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○●○●●●●○○○○○○○○○

$\mathcal{A}_{24}$

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○○○○○○○○●●○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○○●●○○○○○○○○○○○

$\mathcal{W}_{21}$: ○○○○○○○○○○○○○○●●○○○○○○○○○○○○○○○○

$\mathcal{A}_{25}$

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○●○○○○○○●○○○○○○

$\mathcal{W}_{21}$: ○○○○○○○○○○○○○○○○○○○○●○○○○○○○○○○○

$\mathcal{A}_{26}$

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○●

Figure 10.9 – The fifty-one neutral bits regrouped by the first state where they start to interact. A “•” represents the presence of a neutral bit, and a “◦” the absence thereof. The LSB position is the rightmost one.

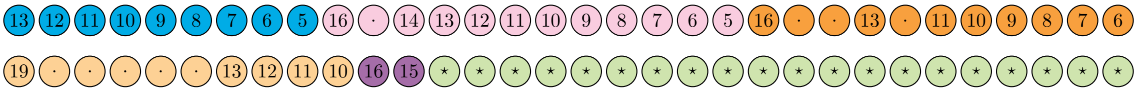


Figure 10.10 – The inter-snippet buffer for steps $\mathcal{A}_{18}$ to $\mathcal{A}_{21}$. From top to bottom, left to right, bright cerulean (●) refers to bits from $\mathcal{W}_{14}$; light rhodamine (●) refers to bits from $\mathcal{W}_{15}$; bright (burnt) orange (●) bits come from $\mathcal{W}_{16}$, while light (yellow) orange (●) means bits from $\mathcal{W}_{17}$. Finally, bright fuchsia (●) and light lime (●) refer to bits of $\mathcal{W}_{18}$ and bits holding the index of a base solution respectively.

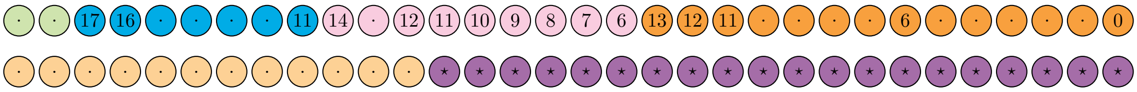


Figure 10.11 – The inter-snippet buffer for steps $\mathcal{A}_{23}$ to $\mathcal{A}_{26}$. From top to bottom, left to right, the two lime bits (●) are bits of padding that do not hold any meaningful data; bright cerulean (●) refers to bits from $\mathcal{W}_{21}$ and light rhodamine (●) to bits from $\mathcal{W}_{19}$; bright (burnt) orange (●) bits come from $\mathcal{W}_{20}$. Light (yellow) orange (●) bits are also padding bits, and bright fuchsia (●) bits finally hold the index of an extended base solution.

that such a partial solution leads to a collision. In our experiments, partial solutions were obtained at a rate of 0.0171 per second on average, while JLCA gives a probability of $2^{-12.67}$, leading to the above expected time to collision. As we mentioned in Section 2.4, this translates to a complexity of $2^{50.34}$ compression function calls on a GTX 970.

4 Freestart collisions for 80-step SHA-1

This section gives the attack parameters for the 80-step full freestart collision attack on SHA-1.

4.1 The differential path for part of the first two rounds

A graphical representation of the differential path used in our attack up to step 28 is given in Figure 10.12. It consists of sufficient conditions for the state, and of the associated message signed bit differences. The meaning of the bit condition symbols were defined in Table 8.2. Note that the signs of message bit differences are enforced through message bit relations. The message bit relations used in the attack past $\mathcal{W}_{28}$ are given in Figure 10.13, and a graphical representation thereof in Figure 10.14. The remainder of the path can easily be determined by linearisation of the step function given the differences in the message.

4.2 The neutral bits and boomerangs

We give here the list of the neutral bits used in the 80-step attack. There are sixty of them over the seven message words $\mathcal{W}_{14}$ to $\mathcal{W}_{20}$, distributed as follows:

- $\mathcal{W}_{14}$: 6 neutral bits at bit positions (starting with the least significant bit (*LSB*) at zero) 5,7,8,9,10,11
- $\mathcal{W}_{15}$: 11 neutral bits at positions 4,7,8,9,10,11,12,13,14,15,16
- $\mathcal{W}_{16}$: 9 neutral bits at positions 8,9,10,11,12,13,14,15,16
- $\mathcal{W}_{17}$: 10 neutral bits at positions 10,11,12,13,14,15,16,17,18,19
- $\mathcal{W}_{18}$: 11 neutral bits at positions 4,6,7,8,9,10,11,12,13,14,15
- $\mathcal{W}_{19}$: 8 neutral bits at positions 6,7,8,9,10,11,12,14
- $\mathcal{W}_{20}$: 5 neutral bits at positions 6,11,12,13,15

We give a graphical representation of the position of these neutral bits in Figure 10.15.

Not all of the neutral bits of the same word (say $\mathcal{W}_{14}$) are used at the same step during the attack. Their repartition in that respect is as follows

- Bits neutral up to step 18 (excluded): $\mathcal{W}_{14}$[8,9,10,11], $\mathcal{W}_{15}$[13,14,15,16]
- Bits neutral up to step 19 (excluded): $\mathcal{W}_{14}$[5,7], $\mathcal{W}_{15}$[8,9,10,11,12], $\mathcal{W}_{16}$[12,13,14,15,16]
- Bits neutral up to step 20 (excluded): $\mathcal{W}_{15}$[4,7,8,9], $\mathcal{W}_{16}$[8,9,10,11,12], $\mathcal{W}_{17}$[14,15,16, 17,18,19]
- Bits neutral up to step 21 (excluded): $\mathcal{W}_{17}$[10,11,12,13], $\mathcal{W}_{18}$[15]
- Bits neutral up to step 22 (excluded): $\mathcal{W}_{18}$[9,10,11,12,13,14], $\mathcal{W}_{19}$[10,14]
- Bits neutral up to step 23 (excluded): $\mathcal{W}_{18}$[4,6,7,8], $\mathcal{W}_{19}$[9,11,12], $\mathcal{W}_{20}$[15]
- Bits neutral up to step 24 (excluded): $\mathcal{W}_{19}$[6,7,8], $\mathcal{W}_{20}$[11,12,13]
- Bit neutral up to step 25 (excluded): $\mathcal{W}_{20}$[7]

One should note that this list only includes a single bit per neutral bit group. As we mentioned in the previous section, some additional flips may be needed in order to preserve message bit relations.

We also give a graphical representation of this repartition in Figure 10.17.

In addition to the “single” neutral bits, the 80-step attack also uses boomerangs. These are regrouped in two sets of two. The first one introduces a difference in the message at word $\mathcal{W}_{10}$; as it does not significantly impact conditions up to step 27, it is used to increase the number of partial solutions for $\mathcal{A}_{28}$. The second set introduces a difference at word $\mathcal{W}_{11}$, and is used on partial solutions up to $\mathcal{A}_{30}$. More precisely, the four boomerangs have their first differences on bits 7,8 of $\mathcal{W}_{10}$ and 8,9 of $\mathcal{W}_{11}$. In Figure 10.16, we give a graphical representation of the complete set of message bits to be flipped for each boomerang. One can see that these follow the pattern of a local collisions, with some "linear" corrections omitted thanks to the absorption properties of the φ_{IF} Boolean function.

We conclude by showing how the neutral bits are packed together with the index of an (extended) base solution in Figure 10.18 and Figure 10.18. Note that neutral bits on $\mathcal{W}_{17}$ are split between the buffers for steps 18–20 and 21–25. Furthermore, the packing of steps 21–25 also includes some "flip" values, which are partial sums of some selected neutral bits that aid in determining if additional bits need to be flipped so as to preserve message bit relations. The representation of the packing is done similarly as in Section 3.2.

4.3 An example of colliding message pair

We give an example of 80-step collision in Table 10.2. This shows the two (message, IV) pairs with their identical resulting digest. This table is formatted in the same way as the one of Section 3.3.

4.4 Complexity of the attack

To estimate the complexity of the attack, we may as in Section 3.4 consider the production rate of partial solutions up to $\mathcal{A}_{60}$ and the corresponding probability of extending these to a collision. However, the slower production rate due to the higher number of conditions would lead to a less reliable estimate than in the 76-step case. Thus, we choose instead to consider partial solutions up to $\mathcal{A}_{40}$. On a good-performing GTX 970, such solutions are produced at a rate of about 1030 per second. The probability that one of these results in a collision is given by JLCA to be $2^{-35.58}$, leading to an expected time to collision of 577.16 days. Recalling that one GTX 970 can compute $2^{31.8}$ SHA-1 compression functions per second, the complexity of the attack is thus of $2^{57.37}$ on such a GPU. However, it should be noted that as for the 76-step attack, the average performance of the GPUs on an actual cluster slightly degrades compared to a single one. In our attack, we used sixteen 4-GPU machines, which combined together led to an attack of complexity about $2^{57.5}$, which takes about ten days to run.

Table 10.2 – A freestart collision for 80-step SHA-1. Message and IV bytes with differences are highlighted with coloured boxes.

	Message 1
IV	50 6b 01 78 ff 6d 18 90 20 22 91 fd 3a de 38 71 b2 c6 65 ea
$\mathcal{M}$	9d 44 38 28 a5 ea 3d f0 86 ea a0 fa 77 83 a7 36
	33 24 48 4d af 70 2a aa a3 da b6 79 d8 a6 9e 2d
	54 38 20 ed a7 ff fb 52 d3 ff 49 3f c3 ff 55 1e
	fb ff d9 7f 55 fe ee f2 08 5a f3 12 08 86 88 a9
$\mathrm{H}(IV, \mathcal{M})$	f0 20 48 6f 07 1b f1 10 53 54 7a 86
	f4 a7 15 3b 3c 95 0f 4b

	Message 2
ΔIV	50 6b 01 78 ff 6d 18 91 a0 22 91 fd 3a de 38 71 b2 c6 65 ea
$\Delta \mathcal{M}$	3f 44 38 38 81 ea 3d ec a0 ea a0 ee 51 83 a7 2c
	33 24 48 5d ab 70 2a b6 6f da b6 6d d4 a6 9e 2f
	94 38 20 fd 13 ff fb 4e ef ff 49 3b 7f ff 55 04
	db ff d9 6f 71 fe ee ee e4 5a f3 06 04 86 88 ab
$\mathrm{H}(\Delta IV, \Delta \mathcal{M})$	f0 20 48 6f 07 1b f1 10 53 54 7a 86
	f4 a7 15 3b 3c 95 0f 4b

i	$\mathcal{A}_i$	$\mathcal{M}_i$
-4	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-3	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	
-2	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○☆▼○	
-1	△○○○△○○○○○○○○○○○○○○○○○○○○▽○○○○○▲	
0	▽△○○▽○○○○○○○○○○○○○○○○○○○○△○○○○○○	●○▲○○○▲○○○○○○○○○○○○○○○○○○○○▲○○○○
1	△△▲☆○○▲○○○○○○○○○○○○○☆○○○○○○▲○○○○	○○▼○○▼○○○○○○○○○○○○○○○○○○○○○▼▲▲○○
2	○○▼△△▼△○△○○○○○○☆○○○○○△▲△△▽○△○▽○○	○○▲○○▼▼○○○○○○○○○○○○○○○○○○○○▼○▲○○
3	○▽○▽▼▽▽△△○☆○△▽○○○▲▽△○▽△△△△☆▽○△○△	○○▼○○▼▼○○○○○○○○○○○○○○○○○○○○▼▲○▼○
4	○△○△△▲▼△▲☆☆☆▲△☆☆☆▽△△☆☆○▼▲▲▲▲▲▼○▲	○○○○○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
5	○▲○▲○▼▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲▲○▲▽▼△△△△	○○○○○▼○○○○○○○○○○○○○○○○○○○○○▲▲▲○○
6	○▽○▽○△○▽△△○△△△○△△△△▽▼▽△▽▽▼△○△▽▼▲	●▲○○▲▲○○○○○○○○○○○○○○○○○○○○○▼○▲○○
7	△▼○▲○△○▽△▽△▽▽▽△▽▽▽▽▽▽▽△△△▲○▼○▽○▲	○○○○▼▲○○○○○○○○○○○○○○○○○○○○○○○○▲○
8	▽▲○▽○▽○○○○○○○○○○○○○○○○▽○○▲○▼○▽○△	●▼○○○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
9	○▲○▽○▽○○○○○○○○○○○○○○○○○○○▽○▲○○○☆	●○▼▲○▼○○○○○○○○○○○○○○○○○○○○○▼▲▲○○
10	○▲○○○○○○○○○○○○○○○○○○○○○○○○○▲○▽○○	○○▼▲▲▲○○○○○○○○○○○○○○○○○○○○○○○▼○○
11	○○○▼○○○○○○○○○○○○○○○○○○○○○○○○○○○○	●○▲▲▲▲○○○○○○○○○○○○○○○○○○○○○▼▲○▲○
12	○○○▽○△○○○○○○○○○○○○○○○○○○○○○○○△○○	○○▼○○○○○○○○○○○○○○○○○○○○○○○○▼○○○○
13	○△○○○▽○○○○○○○○○○○○○○○○○○○○○○○○★☆	○○▲○○▲○○○○○○○○○○○○○○○○○○○○○▼▲▲○○
14	▲▼○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	●▲▲○▲▼○○○○○○○○○○○○○○○○○○○○○▼○▲○○
15	△○△▼○○○○○○○○○○○○○○○○○○○○○○○○○○★○	○○○○▲▼○○○○○○○○○○○○○○○○○○○○○○○○▲○
16	▲○△▽○△○○○○○○○○○○○○○○○○○○○○○○○○○○	●▲○○○○○○○○○○○○○○○○○○○○○○○○○▼○○○○
17	△○▼○○▽○○○○○○○○○○○○○○○○○○○○○○○○○☆	●○▲▲○▲○○○○○○○○○○○○○○○○○○○○○▲▼▼○○
18	○▲▼○▽○○○○○○○○○○○○○○○○○○○○○○○○○○★	○○▲○▼▼○○○○○○○○○○○○○○○○○○○○○○○▼○○
19	○▲○□○○○○○○○○○○○○○○○○○○○○○○○○○○○○	●○▲▼▼▼○○○○○○○○○○○○○○○○○○○○○▼▲○○○
20	▼○○○◆○○○○○○○○○○○○○○○○○○○○○○○○○○○	●○▲▲○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
21	▼○▲◆○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○○○○○○○○○○○○○○○○○○○○○○○○○▲▲○○
22	▼○○○■○○○○○○○○○○○○○○○○○○○○○○○○○○☆	●○▼▼▼○○○○○○○○○○○○○○○○○○○○○○▲○○○○
23	○▼○○◆○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○▼○○○○○○○○○○○○○○○○○○○○○○▲▼○○○
24	▼○◇□○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○▼▲▼▼○○○○○○○○○○○○○○○○○○○○○○▲○○○○
25	▼○▼◇○○○○○○○○○○○○○○○○○○○○○○○○○○○○	○○○○▲○○○○○○○○○○○○○○○○○○○○○○▲○▲○○
26	▼○○○□○○○○○○○○○○○○○○○○○○○○○○○○○○○	○▲▼▼○○○○○○○○○○○○○○○○○○○○○○○▲○○○○
27	▼○▼○◇○○○○○○○○○○○○○○○○○○○○○○○○○○○	●○▲▼▲○○○○○○○○○○○○○○○○○○○○○○▲▲▼○○
28	○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	●▲▼○▼○○○○○○○○○○○○○○○○○○○○○○○○○○○
29	○○▼○○○○○○○○○○○○○○○○○○○○○○○○○○○○○	

Figure 10.12 – The differential path used in the 80-step attack up to step 29.

$\mathcal{W}_{29}[2] = 0$	$\mathcal{W}_{29}[28] = 0$	$\mathcal{W}_{29}[29] = 0$
$\mathcal{W}_{30}[27] \oplus \mathcal{W}_{30}[28] = 1$	$\mathcal{W}_{30}[30] = 1$	$\mathcal{W}_{31}[2] = 0$
$\mathcal{W}_{31}[3] = 0$	$\mathcal{W}_{31}[28] = 0$	$\mathcal{W}_{31}[29] = 0$
$\mathcal{W}_{33}[28] \oplus \mathcal{W}_{33}[29] = 1$	$\mathcal{W}_{30}[4] \oplus \mathcal{W}_{34}[29] = 0$	$\mathcal{W}_{35}[27] = 0$
$\mathcal{W}_{35}[28] = 0$	$\mathcal{W}_{35}[4] \oplus \mathcal{W}_{39}[29] = 0$	$\mathcal{W}_{58}[29] \oplus \mathcal{W}_{59}[29] = 0$
$\mathcal{W}_{57}[29] \oplus \mathcal{W}_{59}[29] = 0$	$\mathcal{W}_{55}[4] \oplus \mathcal{W}_{59}[29] = 0$	$\mathcal{W}_{53}[29] \oplus \mathcal{W}_{54}[29] = 0$
$\mathcal{W}_{52}[29] \oplus \mathcal{W}_{54}[29] = 0$	$\mathcal{W}_{51}[28] \oplus \mathcal{W}_{51}[29] = 1$	$\mathcal{W}_{50}[4] \oplus \mathcal{W}_{54}[29] = 0$
$\mathcal{W}_{50}[28] \oplus \mathcal{W}_{51}[28] = 0$	$\mathcal{W}_{50}[29] \oplus \mathcal{W}_{51}[28] = 1$	$\mathcal{W}_{49}[28] \oplus \mathcal{W}_{51}[28] = 0$
$\mathcal{W}_{48}[29] \oplus \mathcal{W}_{48}[30] = 0$	$\mathcal{W}_{47}[3] \oplus \mathcal{W}_{51}[28] = 0$	$\mathcal{W}_{47}[4] \oplus \mathcal{W}_{51}[28] = 1$
$\mathcal{W}_{46}[29] \oplus \mathcal{W}_{51}[28] = 1$	$\mathcal{W}_{45}[4] \oplus \mathcal{W}_{51}[28] = 0$	$\mathcal{W}_{44}[29] \oplus \mathcal{W}_{51}[28] = 0$
$\mathcal{W}_{43}[4] \oplus \mathcal{W}_{51}[28] = 1$	$\mathcal{W}_{43}[29] \oplus \mathcal{W}_{51}[28] = 0$	$\mathcal{W}_{41}[4] \oplus \mathcal{W}_{51}[28] = 0$
$\mathcal{W}_{63}[4] \oplus \mathcal{W}_{67}[29] = 0$	$\mathcal{W}_{79}[5] = 0$	$\mathcal{W}_{78}[0] = 1$
$\mathcal{W}_{77}[1] \oplus \mathcal{W}_{78}[6] = 1$	$\mathcal{W}_{75}[5] \oplus \mathcal{W}_{79}[30] = 0$	$\mathcal{W}_{74}[0] \oplus \mathcal{W}_{79}[30] = 1$

Figure 10.13 – The message bit relations of the 80-step attack for message words $\mathcal{W}_{29}$ to $\mathcal{W}_{79}$.

```
W29:  . . 0 0 . . . . . . . . . . . . . . . . . . . . . . . . . 0 . .
W30:  . 1 . A a . . . . . . . . . . . . . . . . . . . . . . c . . . .
W31:  . . 0 0 . . . . . . . . . . . . . . . . . . . . . . . . 0 0 . .
W32:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W33:  . . B b . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W34:  . . c . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W35:  . . . 0 0 . . . . . . . . . . . . . . . . . . . . . . d . . . .
W36:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W37:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W38:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W39:  . . d . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W40:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W41:  . . . . . . . . . . . . . . . . . . . . . . . . . . . e . . . .
W42:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W43:  . . e . . . . . . . . . . . . . . . . . . . . . . . . E . . . .
W44:  . . e . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W45:  . . . . . . . . . . . . . . . . . . . . . . . . . . . e . . . .
W46:  . . E . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W47:  . . . . . . . . . . . . . . . . . . . . . . . . . . . E e . . .
W48:  . f f . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W49:  . . . e . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W50:  . . E e . . . . . . . . . . . . . . . . . . . . . . . g . . . .
W51:  . . E e . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W52:  . . g . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W53:  . . g . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W54:  . . g . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W55:  . . . . . . . . . . . . . . . . . . . . . . . . . . . h . . . .
W56:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W57:  . . h . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W58:  . . h . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W59:  . . h . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W60:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W61:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W62:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W63:  . . . . . . . . . . . . . . . . . . . . . . . . . . . i . . . .
W64:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W65:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W66:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W67:  . . i . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W68:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W69:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W70:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W71:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W72:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W73:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W74:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . j
W75:  . . . . . . . . . . . . . . . . . . . . . . . . . . J . . . . .
W76:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
W77:  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . k .
W78:  . . . . . . . . . . . . . . . . . . . . . . . . . K . . . . . 1
W79:  . J . . . . . . . . . . . . . . . . . . . . . . . . 0 . . . . .
```

Figure 10.14 – The message bit-relations used in the attack for words $\mathcal{W}_{29}$ to $\mathcal{W}_{79}$ (graphical representation). A dot ("`.`") means an absence of condition. A zero ("`0`") or a one ("`1`") character represents a bit unconditionally set to 0 or 1. A pair of two letters x means that the two bits have the same value. A pair of two letters x and X means that the two bits have different values.

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○○○●●●●●○●○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○●●●●●●●●●●○○●○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○●●●●●●●●●○○○○○○○○

$\mathcal{W}_{17}$: ○○○○○○○○○○○○●●●●●●●●●●○○○○○○○○○○

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○○●●●●●●●●●●○●○○○○

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○●○●●●●●●●○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○●○●●●○○○○●○○○○○○

Figure 10.15 – The sixty neutral bits of the 80-step attack, using (with some abuse) a "difference" notation. A "○" (resp. "●") symbol means the absence (resp. presence) of a neutral bit on the corresponding bit. The message words are (as usual) written left to right from MSB to LSB.

$\mathcal{W}_{10}$: ○○○○○○○○○○○○○○○○○○○○○○▲★○○○○○○○○

$\mathcal{W}_{11}$: ○○○○○○○○○○○○○○○○○△☆○○○○▼◆○○○○○○○

$\mathcal{W}_{12}$: ○○○○○○○○○○○○○○○○○○▽◇○○○○○○○○○○○○

$\mathcal{W}_{13}$: ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○○○○○○○○☆○○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○○○○○○○○○○△☆○○○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○○○○○○○○○○○▽◇○○○○○

Figure 10.16 – The local collision patterns for each of the four boomerangs, using "difference" symbols by an abuse of notation. The position of the first difference to be introduced is highlighted with a difference (black) symbol; the associated correcting differences (identified by the corresponding white symbols) must then have a sign different from this one. Note that boomerang "★" uses one more difference than the others.

$\mathcal{A}_{18}$

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○○○●●●●○○○○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○●●●●○○○○○○○○○○○○○

$\mathcal{A}_{19}$

$\mathcal{W}_{14}$: ○○○○○○○○○○○○○○○○○○○○○○○○●○●○○○○○

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○○○○○●●●●●○○○○○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○●●●●●○○○○○○○○○○○○

$\mathcal{A}_{20}$

$\mathcal{W}_{15}$: ○○○○○○○○○○○○○○○○○○○○○○○○●○○●○○○○

$\mathcal{W}_{16}$: ○○○○○○○○○○○○○○○○○○○○●●●●○○○○○○○○

$\mathcal{W}_{17}$: ○○○○○○○○○○○○●●●●●●○○○○○○○○○○○○○○

$\mathcal{A}_{21}$

$\mathcal{W}_{17}$: ○○○○○○○○○○○○○○○○○○●●●●○○○○○○○○○○

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○○●○○○○○○○○○○○○○○○

$\mathcal{A}_{22}$

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○○○●●●●●●○○○○○○○○○

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○●○○○●○○○○○○○○○○

$\mathcal{A}_{23}$

$\mathcal{W}_{18}$: ○○○○○○○○○○○○○○○○○○○○○○○●●●○●○○○○

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○○○●●○●○○○○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○●○○○○○○○○○○○○○○○

$\mathcal{A}_{24}$

$\mathcal{W}_{19}$: ○○○○○○○○○○○○○○○○○○○○○○○●●●○○○○○○

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○●●●○○○○○○○○○○○

$\mathcal{A}_{25}$

$\mathcal{W}_{20}$: ○○○○○○○○○○○○○○○○○○○○○○○○○●○○○○○○

Figure 10.17 – The sixty neutral bits regrouped by the first state where they start to interact. A "•" represents the presence of a neutral bit, and a "◦" the absence thereof. The LSB position is the rightmost one.

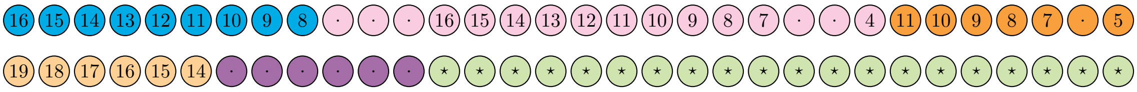


Figure 10.18 – The inter-snippet buffer for steps $\mathcal{A}_{18}$ to $\mathcal{A}_{20}$. From top to bottom, left to right, bright cerulean (●) refers to bits from $\mathcal{W}_{16}$; light rhodamine (●) refers to bits from $\mathcal{W}_{15}$ (with some additional padding); bright (burnt) orange (●) bits come from $\mathcal{W}_{14}$, while light (yellow) orange (●) means bits from $\mathcal{W}_{17}$. Finally, bright fuchsia (●) and light lime (●) refer to padding bits and bits holding the index of a base solution respectively.

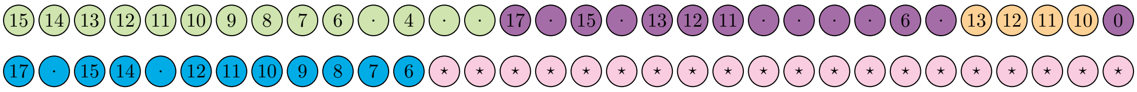


Figure 10.19 – The inter-snippet buffer for steps $\mathcal{A}_{21}$ to $\mathcal{A}_{25}$. From top to bottom, left to right, light lime bits (●) come from $\mathcal{W}_{18}$ (with a few trailing padding bits); bright fuchsia (●) refers to bits from $\mathcal{W}_{20}$, including two “flip” bits on positions 17 and 0, and yellow-tinted orange bits (●) are from $\mathcal{W}_{17}$. Bright cerulean means (●) bits from $\mathcal{W}_{19}$ (with two flip bits on 17 and 15), and finally light rhodamine (●) bits hold the index of an extended base solution.

5 Conclusion

The work described in this chapter culminated in the computation of an explicit freestart collision for the full SHA-1. Although it remains the case that no collision for the entire hash algorithm is known, the progress we have made does allow us to precisely estimate and update what would be the computational and financial cost to generate such a collision, using the latest cryptanalysis results [Ste13]; the computational cost required to generate such a collision is actually a recurrent debate in the academic community since the first theoretical attack from Wang *et al.* [WYY05a].

Schneier's projections [Sch12] on the cost of SHA-1 collisions, made in 2012, were that on Amazon EC2 the cost would be ≈700K US$ by 2015, ≈173K US$ by 2018 and ≈43K US$ by 2021. These computations were based on an early announcement of [Ste13]. These projections have been used to establish the timeline of migrating away from SHA-1-based signatures for secure Internet websites, resulting in a migration by January 2017 —one year before Schneier estimated that a SHA-1 collision would be within the resources of criminal syndicates.

This work demonstrated that GPUs are much faster for this type of attacks when compared to CPUs, and we could estimate that at the time where this work was done, in the autumn 2015, a full SHA-1 collision would not cost more than between 75K and 120K US$ by renting computational power on Amazon EC2. Our new GPU-based projections are more accurate and significantly below Schneier's estimations. What could be considered more worrying is that they were theoretically already within Schneier's estimated financial resources of criminal syndicates as of the end of 2015, almost two years earlier than previously expected, and one year before SHA-1 would start being marked as unsafe in modern Internet browsers. This led us to recommend that migration from SHA-1 to the secure SHA-2 or SHA-3 hash algorithms should be done sooner than previously planned.

It had previously been shown that a more advanced *chosen-prefix collision* attack on MD5 allowed the creation of a rogue Certification Authority undermining the security of all secure websites [SSA⁺09]. Collisions on SHA-1 can result in *e.g.* signature forgeries, but do not directly undermine the security of the Internet at large. Chosen-prefix collisions are significantly more threatening, but currently much costlier to mount for SHA-1. Yet, given the lessons learned with the MD5 full collision break, it is not advisable to wait until these become practically possible to move away from using SHA-1.

At the time of the finding of the 80-step freestart collision, in October 2015, we learned that in an ironic turn of events the CA/Browser Forum, which is the main association of industries regulating the use of digital certificates on the Internet, was planning to hold a ballot to decide whether to extend issuance of SHA-1 certificates through the year 2016 [For15a]. With our new cost projections in mind, we strongly recommended against this extension and the ballot was also subsequently withdrawn [For15b]. Further action was subsequently considered by major browser providers such as Microsoft [Mic15] and Mozilla [Moz15] to speed up the deprecation of SHA-1 certificates.

Chapter 11

Preimage attacks for SHA-1

1 Introduction

This short chapter presents new preimage attacks for the SHA-1 hash function. The results are obtained by extending the differential view of meet-in-the-middle preimage attacks with higher-order differentials.

The starting point of the work described here is the meet-in-the-middle technique, which was first used in cryptography by Diffie and Hellman in 1977 to attack double-encryption [DH77]. Its use for preimage attack is much more recent and is due to Aoki and Sasaki, who used it as a framework to attack various hash functions, including for instance SHA-0 and SHA-1 [AS09]. The basic principle behind a meet-in-the-middle technique is to exploit the fact that some intermediate value of a function's computation can be expressed in two different ways involving different parts of a secret, which can then be sampled independently of each other. In the case of hash function cryptanalysis, there is no actual secret to consider, but a similar technique can nonetheless be exploited in certain cases; we show in more details how to do so in the preliminaries of Section 2.

At CRYPTO 2012, Knellwolf and Khovratovich introduced a differential formulation of the meet-in-the-middle framework of Aoki and Sasaki, which they used to improve the best attacks on SHA-1 [KK12]. One of the main interests of their approach is that it simplifies the formulation of several advanced extensions of the meet-in-the-middle technique, and thereby facilitates the search for attack parameters, which in the case of meet-in-the-middle attacks roughly correspond to good partitions for the "secret".

In this chapter, we further exploit this differential formulation and generalise it to use higher-order differentials, which were introduced in cryptography by Lai in 1994 [Lai94]. The essence of this technique is to consider "standard" differential cryptanalysis as exploiting properties of the first-order derivative of the function one wishes to analyse; it is then somehow natural to generalise the idea and to consider higher-order derivatives as well. We illustrate this with a small example using XOR differences: consider a function f and assume the equality $\Delta_\alpha f(x) := f(x) \oplus f(x \oplus \alpha) = A$ holds with a good probability over the values of x; this is the same as saying that the derivative of f in α is biased towards A. In an extreme case, if f is linear, then $\Delta_\alpha f$ is constantly equal to $f(\alpha)$. Now if we consider the expression $\Delta_\alpha f(x) \oplus \Delta_\alpha f(x \oplus \beta) = f(x) \oplus f(x \oplus \alpha) \oplus f(x \oplus \beta) \oplus f(x \oplus \alpha \oplus \beta)$, this corresponds to taking the derivative of f twice, first in α, and then in β. A possible advantage of doing this is that the resulting function may be more biased than $\Delta_\alpha f$ was, for instance by being constant when $\Delta_\alpha f$ is linear. This process can be iterated at will, each time decreasing the algebraic degree of the resulting function until it reaches zero.

As higher-order differentials are obviously best formulated in differential form, they combine neatly with the differential view of the framework of Knellwolf and Khovratovich, whereas using a similar technique in a meet-in-the-middle attack independently of any differential formulation would probably prove to be much more difficult. As a final motivation for this generalisation, we show a small application to the analysis of the MD4 hash function [Riv90]. This does not improve the best known preimage attacks [GLRW10, ZL12], but gives a good illustration of the potential of the technique.

Higher-order differentials for the compression function of MD4. The inverse of the state update function inside the compression function of MD4 is of the form: $q_{i-4} \leftarrow (q_i \circlearrowleft s_i) - \varphi(q_{i-1}, q_{i-2}, q_{i-3}) - m_j$, with the subtraction being done modulo 2^{32}. Four consecutive steps of this inverse function can thus be written as:

$$q_3 \leftarrow (q_7 \circlearrowleft s_7) - \varphi(q_6, q_5, q_4) - m_7$$

$$q_2 \leftarrow (q_6 \circlearrowleft s_6) - \varphi(q_5, q_4, q_3) - m_6$$

$$q_1 \leftarrow (q_6 \circlearrowleft s_5) - \varphi(q_4, q_3, q_2) - m_5$$

$$q_0 \leftarrow (q_4 \circlearrowleft s_4) - \varphi(q_3, q_2, q_1) - m_4$$

If we consider order-2 differentials on m_6 and m_7, with additive differences modulo 2^{32} concentrated around the most significant bit, that is of the form ●●●●○○○○, computing the state update from above on these differences results in a state $q_{0 \ldots 3}$ with differences of the form:

$q_3^{\ddagger}$:	∗∗∗∗○○○○	$q_3^{\star}$:	○○○○○○○○	$q_3^{\dagger}$:	equal to $q_3^{\ddagger}$
$q_2^{\ddagger}$:	∗∗∗∗○○○○	$q_2^{\star}$:	∗∗∗∗○○○○	$q_2^{\dagger}$:	∗∗∗∗○○○○
$q_1^{\ddagger}$:	∗∗∗∗○○○○	$q_1^{\star}$:	∗∗∗∗○○○○	$q_1^{\dagger}$:	∗∗∗∗○○○○
$q_0^{\ddagger}$:	∗∗∗∗○○○○	$q_0^{\star}$:	∗∗∗∗○○○○	$q_0^{\dagger}$:	∗∗∗∗○○○○
For diff. on m_7		For diff. on m_6		For diff. on m_6 & m_7	

Thus there is no difference on the value $q_3^{\dagger} - q_3^{\star} - q_3^{\ddagger} + q_3$, and the differences on the remaining words also have a high probability. This good differential behaviour can then be exploited in a later attack.

It is worth noting that in the very case of MD4, one could also use very good local collisions from order-1 message differences, and higher-order differentials do not typically outperform these; we just gave them for illustration.

1.1 Previous and new results on SHA-1

The first preimage attacks on SHA-1 were due to De Cannière and Rechberger [DR08], who used a system-based approach that in particular allows to compute practical preimages for a non-trivial number of steps. In order to attack more steps, Aoki and Sasaki later used a meet-in-the-middle approach [AS09]. This was subsequently improved by Knellwolf and Khovratovich [KK12], who attacked the highest number of rounds so far. To be more precise, they attack reduced versions of the function up to 52 steps for *one-block preimages with padding*, 57 steps for *one-block preimages without padding*, and 60 steps for *one-block pseudo-preimages with padding*, *i.e.* freestart preimages with padding. The latter two attacks can be combined to give 57 steps *two-block preimages with padding*. In this work, we present *one-block preimages with padding* up to 56 steps, *one-block preimages without padding* up to 62 steps, *one-block pseudo preimages with padding* up to 64 steps, resulting in *two-block preimages with padding* up to 62 steps.

We give a summary of these results in Table 11.1.

Table 11.1 – Existing and new preimage attacks on SHA-1, with complexity given in base-2 logarithm.

Function	# blocks	# rounds	complexity	ref.
SHA-1	1	52	158.4	[KK12]
	1	52	156.7	Section 4.2
	1	56	159.4	Section 4.2
	2	57	158.8	[KK12]
	2	58	157.9	Section 4.3
	2	62	159.3	Section 4.3
SHA-1, without padding	1	57	158.7	[KK12]
	1	58	157.4	Section 4.1
	1	62	159	Section 4.1
SHA-1, pseudo-preimage	1	60	157.4	[KK12]
	1	61	156.4	Section 4.3
	1	64	158.7	Section 4.3

2 Meet-in-The-Middle Attacks and the Differential Framework from CRYPTO 2012

As a preliminary, we give a description of the meet-in-the-middle framework for preimage attacks on hash functions, and in particular of the differential formulation of Knellwolf and Khovratovich from CRYPTO 2012 [KK12].

The relevance of meet-in-the-middle for preimage attacks comes from the fact that many hash functions are built from a compression function which is an *ad hoc* block cipher used in one of the PGV modes [PGV93]. The SHA-1 function itself follows this design strategy and uses the particular Davies-Meyer mode, already described in Chapter 4. We recall that in this mode, the compression function $\mathrm{H} : \{0,1\}^v \times \{0,1\}^n \to \{0,1\}^v$ compressing a chaining value c with a message m to form the updated chaining value $c' := \mathrm{H}(c,m)$ is defined as $\mathrm{H}(c,m) = f(m,c) + c$, with $f : \{0,1\}^n \times \{0,1\}^v \to \{0,1\}^v$ a block cipher of key-length and message-length n and v. Given a compression function H, the problem of finding a preimage of t for H is then equivalent to finding a key m for f such that $f(m,p) = c$ for a pair (p,c), with $c = t - p$. Additional constraints can also be put on p, such as prescribing it to a fixed initialisation value IV.

In its most basic form, a meet-in-the-middle attack can speed-up the search for a preimage if the block cipher f can equivalently be described as the composition $f_2 \circ f_1$ of two block ciphers $f_1 : \mathcal{K}_1 \times \{0,1\}^v \to \{0,1\}^v$ and $f_2 : \mathcal{K}_2 \times \{0,1\}^v \to \{0,1\}^v$ with independent key spaces $\mathcal{K}_1, \mathcal{K}_2 \subset \{0,1\}^n$. Indeed, if such a decomposition is possible, an attacker can select a subset $\{k_i^1, i = 1 \dots N_1\}$ (resp. $\{k_j^2, j = 1 \dots N_2\}$) of keys of $\mathcal{K}_1$ (resp. $\mathcal{K}_2$), which together suggest $N := N_1 \cdot N_2$ candidate keys $k_{ij}^{12} := (k_i^1, k_j^2)$ for f by setting $f(k_{ij}^{12}, \cdot) = f_2(k_j^2, \cdot) \circ f_1(k_i^1, \cdot)$.

Since the two sets $\{f_1(k_i^1, p), i = 1 \dots N_1\}$ and $\{{f_2}^{-1}(k_j^2, c), j = 1 \dots N_2\}$ can be computed independently, the complexity of testing $f(k_{ij}^{12}, p) = c$ for N keys is only of $\mathcal{O}(\max(N_1, N_2))$ time and $\mathcal{O}(\min(N_1, N_2))$ memory, which is less than N and can be as low as $\sqrt{N}$ when $N_1 = N_2$.

2.1 Formalizing meet-in-the-middle attacks with related-key differentials

Let us denote by $(\alpha, \beta) \xrightarrow[p]{f} \gamma$ the fact that $\Pr_{(x,y)}\left[f(x \oplus \alpha, y \oplus \beta) = f(x,y) \oplus \gamma\right] = p$, meaning that (α, β) is a related-key differential for f that holds with probability p. The goal of an attacker is now to find two linear sub-spaces D_1 and D_2 of $\{0,1\}^m$ such that:

$$D_1 \cap D_2 = \{0\} \tag{11.1}$$

$$\forall \delta_1 \in D_1 \ \exists \ \Delta_1 \in \{0,1\}^v \text{ s.t. } (\delta_1, 0) \xrightarrow[1]{f_1} \Delta_1 \tag{11.2}$$

$$\forall \delta_2 \in D_2 \ \exists \ \Delta_2 \in \{0,1\}^v \text{ s.t. } (\delta_2, 0) \xrightarrow[1]{{f_2}^{-1}} \Delta_2. \tag{11.3}$$

Let d_1 and d_2 be the dimension of D_1 and D_2 respectively. Then for a set M of messages $\mu_i \in \{0,1\}^m$, or more precisely the quotient space of $\{0,1\}^m$ by $D_1 \oplus D_2$, one can define

$\#M$ distinct sets $\mu_i \oplus D_1 \oplus D_2$ of dimension $d_1 + d_2$ (and size $2^{d_1+d_2}$), which can be tested for a preimage with a complexity of only $\mathcal{O}(\max(2^{d_1}, 2^{d_2}))$ time and $\mathcal{O}(\min(2^{d_1}, 2^{d_2}))$ memory. We recall the procedure to do so in Algorithm 11.1.

Algorithm 11.1: Testing $\mu \oplus D_1 \oplus D_2$ for a preimage [KK12]

```
Input: $D_1, D_2 \subset \{0,1\}^m$, $\mu \in \{0,1\}^m$, $p$, $c$
Output: A preimage of $c + p$ if there is one in $\mu \oplus D_1 \oplus D_2$, $\perp$ otherwise
Data: Two lists $L_1, L_2$ indexed by $\delta_2, \delta_1$ respectively
forall $\delta_2 \in D_2$ do
    $L_1[\delta_2] \leftarrow f_1(\mu \oplus \delta_2, p) \oplus \Delta_2$
forall $\delta_1 \in D_1$ do
    $L_2[\delta_1] \leftarrow {f_2}^{-1}(\mu \oplus \delta_1, c) \oplus \Delta_1$
forall $(\delta_1, \delta_2) \in D_1 \times D_2$ do
    if $L_1[\delta_2] = L_2[\delta_1]$ then
        return $\mu \oplus \delta_1 \oplus \delta_2$
return $\perp$
```

2.1.1 Analysis of Algorithm 11.1

For the sake of simplicity we assume that d_1 and d_2 are equal to a certain value $d < \frac{v}{2}$. The running time of every loop of Algorithm 11.1 is therefore $\mathcal{O}(2^d)$, assuming efficient data structures and equality testing for the lists, and $\mathcal{O}(2^d)$ memory is necessary for storing L_1 and L_2. It is also clear that if the condition $L_1[\delta_2] = L_2[\delta_1]$ is met, then $\mu \oplus \delta_1 \oplus \delta_2$ is a preimage of $c + p$. Indeed, this translates to $f_1(\mu \oplus \delta_2, p) \oplus \Delta_2 = {f_2}^{-1}(\mu \oplus \delta_1, c) \oplus \Delta_1$, and using the differential properties of D_1 and D_2 for f_1 and f_2, we have that $f_1(\mu \oplus \delta_1 \oplus \delta_2, p) = f_1(\mu \oplus \delta_2, p) \oplus \Delta_1$ and ${f_2}^{-1}(\mu \oplus \delta_1 \oplus \delta_2, c) = {f_2}^{-1}(\mu \oplus \delta_1, c) \oplus \Delta_2$. Hence, $f_1(\mu \oplus \delta_1 \oplus \delta_2, p) = {f_2}^{-1}(\mu \oplus \delta_1 \oplus \delta_2)$, and $f(\mu \oplus \delta_1 \oplus \delta_2, p) = c$. This algorithm therefore allows to search through 2^{2d} candidate preimages with a complexity of $\mathcal{O}(2^d)$, and thus gives a speed-up of 2^d. The complexity of a full attack is hence $\mathcal{O}(2^{v-d})$.

2.1.2 Comparison with the basic meet-in-the-middle attack

When setting $\Delta_1 = \Delta_2 = 0$, this differential variant of the meet-in-the-middle technique becomes a special case of the general formulation of the basic technique given at the beginning of this section: the key spaces $\mathcal{K}_1$ and $\mathcal{K}_2$ now possess a structure of affine spaces. Although this is a restriction on the shape of the key partition, the additional structure is useful when one is intent on finding actual attacks.

The advantage of the differential view of the meet-in-the-middle attack then comes from the fact that it gives a practical way of searching for the key spaces, as differential path search is a well-studied area of symmetric cryptanalysis. Another major advantage is that it makes the formulation of several extensions to the basic attack very natural, without compromising the ease of the search for the key spaces. One such immediate extension is obviously to consider non-zero values for Δ_1 and Δ_2. As noted by Knellwolf and Khovratovich, this already corresponds to an advanced technique of *indirect matching* in the original framework of Aoki and Sasaki. Further extensions are detailed next.

2.2 Probabilistic truncated differential meet-in-the-middle

There are two natural ways to generalise the differential formulation of the meet-in-the-middle, that each correspond to relaxing one condition. First, one can consider differentials of probability less than one —although a high probability is still usually needed; second, one can consider truncated differentials by using an equivalence relation "$\equiv$" instead of the equality, with one usually taking $\equiv$ to be a truncated equality: $a \equiv b\ [m] \Leftrightarrow a \wedge m = b \wedge m$ for $a, b, m \in \{0,1\}^v$. One can then use the notation $(\alpha, \beta) \underset{p}{\overset{f}{\rightsquigarrow}} \gamma$ to denote the fact that $\Pr_{(x,y)}\left[f(x \oplus \alpha, y \oplus \beta) \equiv f(x,y) \oplus \gamma\right] = p$. Hence Equation 11.2 becomes:

$$\forall \delta_1 \in D_1\ \exists\ \Delta_1 \in \{0,1\}^v \text{ s.t. } (\delta_1, 0) \underset{p_1}{\overset{f_1}{\rightsquigarrow}} \Delta_1, \tag{11.4}$$

for some probability p_1 and relation $\equiv$, and the same changes apply to Equation 11.3.

Again, these generalisations correspond to advanced techniques of Aoki and Sasaki's framework, which find here a concise and efficient description.

The only change to Algorithm 11.1 needed to accommodate these extensions is to replace the equality by the appropriate equivalence relation on line 6. However, the fact that this equivalence holds no longer ensures that $x := \mu \oplus \delta_1 \oplus \delta_2$ is a preimage, which implies an increased complexity for the attack. This increase comes from two factors: first, even when x *is* a preimage, the relation on line 6 might not hold with probability $1 - p_1 p_2$, meaning that on average one needs to test $1/p_1 p_2$ times more candidates in order to account for the false negatives; secondly, if we denote by s the average size of the equivalence classes under $\equiv$ (when using truncation as above, this is equal to 2^{v-r} with r the Hamming weight of m), then one needs to check s potential preimages as returned on line 6 on average before finding a valid one, in order to account for the false positives. The total complexity of an attack with the modified algorithm is therefore $\mathcal{O}((2^{v-d} + s)/\tilde{p_1}\tilde{p_2})$, where $\tilde{p_1}$ and $\tilde{p_2}$ are the respective average probabilities for p_1 and p_2 over the spaces D_1 and D_2.

2.3 Splice-and-cut, initial structures and bicliques

The two techniques we present now are older than the framework of [KK12], but are fully compatible with its differential approach.

Splice-and-cut was introduced by Aoki and Sasaki in 2008 [AS08]. Its idea is to use the feedforward of the compression function so as to be able to start the computation of f_1 and ${f_2}^{-1}$ not from p and c but from an intermediate value from the middle of the computation of f. If one sets $f = f_3 \circ f_2 \circ f_1$ and calls s the intermediate value ${f_3}^{-1}(c)$ (or equivalently $f_2 \circ f_1(p)$), an attacker may now sample the functions $f_1(t - f_3(s))$ and

${f_2}^{-1}(s)$ on their respective key-spaces, which are as always taken to be independent, when searching a preimage for t. By giving more possible choices for the decomposition of f, one can hope for better attacks. This however comes at the cost that they are now pseudo-preimage attacks, as one does not control the value of the IV anymore, which becomes equal to $t - f_3(s)$.

A possible improvement to a splice-and-cut decomposition is the use of *initial structures* [SA09], which were later reformulated as *bicliques* [KRS12]. Instead of starting the computations in the middle from an intermediate value s, the idea is now to start from a set of multiple values possessing a special structure that spans several rounds. If the cost of constructing such sets is negligible w.r.t the rest of the computations, the rounds spanned by the structure actually come for free. In more details, a biclique, say for f_3 in the above decomposition of f, is a set $\{m, D_1, D_2, Q_1, Q_2\}$ where m is a message, D_1 and D_2 are linear spaces of dimension d, and Q_1 (resp. Q_2) is a list of 2^d values indexed by the differences δ_1 of D_1 (resp. D_2) s.t. $\forall(\delta_1, \delta_2) \in D_1 \times D_2 \quad Q_2[\delta_2] = f_3(m \oplus \delta_1 \oplus \delta_2, Q_1[\delta_1])$. This allows to search the message space $m \oplus D_1 \oplus D_2$ in $\mathcal{O}(2^d)$ with a meet-in-the-middle approach that does not need any call to f_3, essentially bypassing this part of the decomposition.

3 Higher-Order Differential Meet-in-The-Middle

We now describe how to modify the framework of Section 2 to use higher-order differentials. Let us denote by $(\{\alpha_1, \alpha_2\}, \{\beta_1, \beta_2\}) \xrightarrow[p]{f} \gamma$ the fact that $\Pr_{(x,y)}\big[f(x \oplus \alpha_1 \oplus \alpha_2, y \oplus \beta_1 \oplus \beta_2) \oplus f(x \oplus \alpha_1, y \oplus \beta_1) \oplus f(x \oplus \alpha_2, y \oplus \beta_2) = f(x, y) \oplus \gamma\big] = p$, meaning that $(\{\alpha_1, \alpha_2\}, \{\beta_1, \beta_2\})$ is a related-key order-2 differential for f that holds with probability p.

Similarly as in Section 2, the goal of the attacker is to find four linear subspaces $D_{1,1}, D_{1,2}, D_{2,1}, D_{2,2}$ of $\{0,1\}^m$ in direct sum such that:

$$\forall \delta_{1,1}, \delta_{1,2} \in D_{1,1} \times D_{1,2} \; \exists \; \Delta_1 \in \{0,1\}^v \text{ s.t. } (\{\delta_{1,1}, \delta_{1,2}\}, \{0,0\}) \xrightarrow[1]{f_1} \Delta_1 \tag{11.5}$$

$$\forall \delta_{2,1}, \delta_{2,2} \in D_{2,1} \times D_{2,2} \; \exists \; \Delta_2 \in \{0,1\}^v \text{ s.t. } (\{\delta_{2,1}, \delta_{2,2}\}, \{0,0\}) \xrightarrow[1]{{f_2}^{-1}} \Delta_2. \tag{11.6}$$

Then $M \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}$ is a *preimage* of $c + p$ if and only if $f_1(\mu \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}, c) = {f_2}^{-1}(\mu \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}, p)$ which is equivalent by the Equation 11.5 and Equation 11.6 to the equality:

$$\begin{array}{ccc} f_1(\mu \oplus \delta_{1,1} \oplus \delta_{2,1} \oplus \delta_{2,2}, p) \oplus & & {f_2}^{-1}(\mu \oplus \delta_{2,1}, \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus \\ f_1(\mu \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}, p) \oplus & = & {f_2}^{-1}(\mu \oplus \delta_{2,2}, \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus \\ f_1(\mu \oplus \delta_{2,1} \oplus \delta_{2,2}, p) \oplus \Delta_1 & & {f_2}^{-1}(\mu \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus \Delta_2. \end{array} \tag{11.7}$$

We denote by $d_{i,j}$ the dimension of the sub-space $D_{i,j}$ for $i, j = 1, 2$. Then for a set M of messages $\mu \in \{0,1\}^m$) one can define $\#M$ affine sub-sets $\mu_i \oplus D_{1,1} \oplus D_{1,2} \oplus D_{2,1} \oplus D_{2,2}$ of dimension $d_{1,1}+d_{1,2}+d_{2,1}+d_{2,2}$ (since the sub-spaces $D_{i,j}$ are in direct sum by hypothesis), which can be tested for a preimage using Equation 11.7. This can be done efficiently by a modification of Algorithm 11.1 into the following Algorithm 11.2.

Algorithm 11.2: Testing $\mu \oplus D_{1,1} \oplus D_{1,2} \oplus D_{2,1} \oplus D_{2,2}$ for a preimage

```
Input: $D_{1,1}, D_{1,2}, D_{2,1}, D_{2,2} \subset \{0,1\}^m$, $\mu \in \{0,1\}^m$, $p$, $c$
Output: A preimage of $c+p$ if there is one in $\mu \oplus D_{1,1} \oplus D_{1,2}, \oplus D_{2,1} \oplus D_{2,2}$, $\perp$
        otherwise
Data: Six lists:
$L_{1,1}$ indexed by $\delta_{1,2}, \delta_{2,1}, \delta_{2,2}$
$L_{1,2}$ indexed by $\delta_{1,1}, \delta_{2,1}, \delta_{2,2}$
$L_{2,1}$ indexed by $\delta_{1,1}, \delta_{1,2}, \delta_{2,2}$
$L_{2,2}$ indexed by $\delta_{1,1}, \delta_{1,2}, \delta_{2,1}$
$L_1$ indexed by $\delta_{2,2}, \delta_{2,1}$
$L_2$ indexed by $\delta_{1,1}, \delta_{1,2}$
1  forall $\delta_{1,2}, \delta_{2,1}, \delta_{2,2} \in D_{1,2} \times D_{2,1} \times D_{2,2}$ do
2      $L_{1,1}[\delta_{1,2}, \delta_{2,1}, \delta_{2,2}] \leftarrow {f_2}^{-1}(\mu \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}, c)$ ;
3  forall $\delta_{1,1}, \delta_{2,1}, \delta_{2,2} \in D_{1,1} \times D_{2,1} \times D_{2,2}$ do
4      $L_{1,2}[\delta_{1,1}, \delta_{2,1}, \delta_{2,2}] \leftarrow {f_2}^{-1}(\mu \oplus \delta_{1,1} \oplus \delta_{2,1} \oplus \delta_{2,2}, c)$ ;
5  forall $\delta_{1,1}, \delta_{1,2}, \delta_{2,2} \in D_{1,1} \times D_{1,2} \times D_{2,2} \times$ do
6      $L_{2,1}[\delta_{1,1}, \delta_{1,2}, \delta_{2,2}] \leftarrow f_1(\mu \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,2}, p)$ ;
7  forall $\delta_{1,1}, \delta_{1,2}, \delta_{2,1} \in D_{1,1} \times D_{1,2} \times D_{2,1}$ do
8      $L_{2,2}[\delta_{1,1}, \delta_{1,2}, \delta_{2,1}] \leftarrow f_1(\mu \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,1}, p)$ ;
9  forall $\delta_{1,1}, \delta_{1,2} \in D_{1,1} \times D_{1,2}$ do
10     $L_2[\delta_{1,1}, \delta_{1,2}] \leftarrow {f_2}^{-1}(\mu \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus \Delta_1$ ;
11 forall $\delta_{2,1}, \delta_{2,2} \in D_{2,1} \times D_{2,2}$ do
12     $L_1[\delta_{2,1}, \delta_{2,2}] \leftarrow f_1(\mu \oplus \delta_{2,1} \oplus \delta_{2,2}, p) \oplus \Delta_2$ ;
13 forall $\delta_{1,1}, \delta_{1,2}, \delta_{2,1}, \delta_{2,2} \in D_{1,1} \times D_{1,2} \times D_{2,1} \times D_{2,2}$ do
14     if $L_{1,1}[\delta_{1,2}, \delta_{2,1}, \delta_{2,2}] \oplus L_{1,2}[\delta_{1,1}, \delta_{2,1}, \delta_{2,2}] \oplus L_1[\delta_{2,1}, \delta_{2,2}] =$
         $L_{2,1}[\delta_{1,1}, \delta_{1,2}, \delta_{2,2}] \oplus L_{2,2}[\delta_{1,1}, \delta_{1,2}, \delta_{2,1}] \oplus L_2[\delta_{1,1}, \delta_{1,2}]$ then
15         return $\mu \oplus \delta_{1,1} \oplus \delta_{1,2} \oplus \delta_{2,1} \oplus \delta_{2,2}$
16 return $\perp$
```

3.1 Analysis of Algorithm 11.2

If we denote by Γ_1 and Γ_2 the cost of evaluating of f_1 and ${f_2}^{-1}$ and Γ_{match} the cost of the test on line 14, then the algorithm allows to test $2^{d_{1,1}+d_{1,2}+d_{2,1}+d_{2,2}}$ messages with a complexity of $2^{d_{1,2}+d_{2,1}+d_{2,2}}\Gamma_2+2^{d_{1,1}+d_{2,1}+d_{2,2}}\Gamma_2+2^{d_{1,1}+d_{1,2}+d_{2,1}}\Gamma_1+2^{d_{1,1}+d_{1,2}+d_{2,1}}\Gamma_1+2^{d_{1,1}+d_{1,2}}\Gamma_2+2^{d_{2,1}+d_{2,2}}\Gamma_1+\Gamma_{\mathrm{match}}$. The algorithm must then be run $2^{n-(d_{1,1}+d_{1,2}+d_{2,1}+d_{2,2})}$ times in order to test 2^n messages. In the special case where all the linear spaces have the same dimension d and if we consider that Γ_{match} is negligible with respect to the total complexity, the total complexity of an attack is then of : $2^{n-4d} \cdot (2^{3d} \cdot (2\Gamma_1 + 2\Gamma_2) + 2^{2d} \cdot (\Gamma_1 + \Gamma_2)) = 2^{n-d+1}\Gamma + 2^{n-2d}\Gamma = \mathcal{O}(2^{n-d})$ where Γ is the cost of the evaluation of the total compression function f. We think that the assumption on the cost of Γ_{match} is reasonable given the small size of d in actual attacks and the fact that performing a single match is much faster than computing f.

The factor that is gained from a brute-force search of complexity $\mathcal{O}(2^n)$ is hence of 2^d, which is the same as for Algorithm 11.1. However, one now needs four spaces of differences of size 2^d instead of only two, which might look like a setback. Indeed the real interest of this method does not lie in a simpler attack but in the fact that using higher-order differentials may now allow to attack functions for which no good-enough order-1 differentials are available.

3.2 Using probabilistic truncated differentials

Similarly as in Section 2, Algorithm 11.2 can be modified in order to use probabilistic truncated differentials instead of probability-1 differentials on the full state. The changes to the algorithm and the complexity evaluation are identical to the ones described in Section 2.2.

4 Applications to SHA-1

4.1 One-block preimages without padding

We can apply the framework of Section 3 to mount attacks on SHA-1 for *one-block preimages without padding*. These are rather direct applications of the framework, the only difference being the fact that we use *sets* of differentials instead of *linear spaces*. This has no impact on Algorithm 11.2, but makes the description of the attack parameters less compact.

As was noted in [KK12], the message expansion of SHA-1 being linear, it is possible to attack 15 steps both in the forward and backward direction, for a total of 30, without advanced matching techniques: it is sufficient to use a message difference in the kernel of the 15 first steps of the message expansion. When applying our framework to attack more steps, say 55 to 62, we have observed experimentally that splitting the forward and backward parts around steps 22 to 27 seems to give the best results. A similar behaviour was observed by Knellwolf and Khovratovich in their attacks, and this can be explained by the fact that the SHA-1 step function has a somewhat weaker diffusion when computed backward compared to forward.

We used Algorithm 11.3 to construct a suitable set of differences in the preparation of an attack. This algorithm was run on input differences of low Hamming weight; a difference was kept only when it resulted in output differences with truncation masks that are long enough and with good overall probabilities. The sampling parameter Q that we used was 2^{15}; the threshold value t was subjected to a tradeoff: the larger it is, the less bits are chosen in the truncation mask, but the better the probability of the resulting differential. In practice, we used values between 2 and 5, depending on the differential considered.

Algorithm 11.3: Computing a suitable output difference for a given input difference

Input: A chunk f_i of the compression function, $\delta_{i,1}, \delta_{i,2} \in \{0,1\}^m$, a threshold value t, a sample size Q, an internal state c.

Output: An output difference $\mathcal{S}$, and a mask $T_{\mathcal{S}}$ for the differential $((\delta_{i,1}, \delta_{i,2}), 0) \overset{f_i}{\rightsquigarrow} \mathcal{S}$

Data: An array d of n counters initially set to 0.

```
for $q = 0$ to $Q$ do
    Choose $\mu \in \{0,1\}^m$ at random;
    $\Delta \leftarrow f_i(\mu \oplus \delta_{i,1} \oplus \delta_{i,2}, c) \oplus f_i(\mu \oplus \delta_{i,1}, c) \oplus f_i(\mu \oplus \delta_{i,2}, c) \oplus f_i(\mu, c)$;
    for $i = 0$ to $n - 1$ do
        if the $i^{th}$ bit of $\Delta$ is 1 then
            $d[i] \leftarrow d[i] + 1$;
for $i = 0$ to $n - 1$ do
    if $d[i] \geq t$ then
        Set the $i$-th bit of the output difference $\mathcal{S}$ to 1;
```

Once input and output differences had been chosen, we used an adapted version of [KK12, Algorithm 2] given in Algorithm 11.4 to compute suitable truncation masks.

Algorithm 11.4: Find truncation mask T for matching

Input: $D_{1,1}, D_{1,2}, D_{2,1}, D_{2,2} \subset \{0,1\}^m$, a sample size Q, a mask size d.

Output: A truncation mask $T \in \{0,1\}^n$ of Hamming weight d.

Data: An array k of n counters initially set to 0.

```
for $q = 0$ to $Q$ do
    Choose $\mu \in \{0,1\}^m$ at random ;
    $c \leftarrow f(\mu, IV)$;
    Choose $(\delta_{1,1}, \delta_{1,2}, \delta_{2,1}, \delta_{2,2}) \in D_{1,1} \times D_{1,2} \times D_{2,1} \times D_{2,2}$ at random;
    $\Delta \leftarrow f_1(\mu \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus f_1(\mu \oplus \delta_{1,1}, c) \oplus f_1(\mu \oplus \delta_{1,2}, c)$;
    $\Delta \leftarrow \Delta \oplus f_2^{-1}(\mu \oplus \delta_{2,1} \oplus \delta_{2,2}, c) \oplus f_2^{-1}(\mu \oplus \delta_{2,2}, c) \oplus f_2^{-1}(\mu \oplus \delta_{2,2}, c)$;
    for $i = 0$ to $n - 1$ do
        if the $i^{th}$ bit of $\Delta$ is 1 then
            $k[i] \leftarrow k[i] + 1$;
Set the d bits of lowest counter value in $k$ to 1 in T.
```

The choice of the size of the truncation mask d in this algorithm offers a tradeoff between the probability one can hope to achieve for the resulting truncated differential and how efficient a filtering of “ bad ” messages it will offer. In our applications to SHA-1, we chose masks of size about $\min(\log_2(|D_{1,1}|), \log_2(|D_{1,2}|), \log_2(|D_{2,1}|), \log_2(|D_{2,2}|))$, which is consistent with taking masks of size the dimension of the affine spaces as is done in [KK12].

We similarly adapted [KK12, Algorithm 3] as Algorithm 11.5 in order to estimate the average false negative probability associated with the final truncated differential.

Algorithm 11.5: Estimate the average false negative probability

```
Input: $D_{1,1}, D_{1,2}, D_{2,1}, D_{2,2} \subset \{0,1\}^m, T \in \{0,1\}^n$, a sample size $Q$
Output: Average false negative probability $\alpha$.
Data: A counter $k$ initially set to 0.
for $q = 0$ to $Q$ do
    Choose $\mu \in \{0,1\}^m$ at random ;
    $c \leftarrow f(\mu, IV)$;
    Choose $(\delta_{1,1}, \delta_{1,2}, \delta_{2,1}, \delta_{2,2}) \in D_{1,1} \times D_{1,2} \times D_{2,1} \times D_{2,2}$ at random;
    $\Delta \leftarrow f_1(\mu \oplus \delta_{1,1} \oplus \delta_{1,2}, c) \oplus f_1(\mu \oplus \delta_{1,1}, c) \oplus f_1(\mu \oplus \delta_{1,2}, c)$;
    $\Delta \leftarrow \Delta \oplus {f_2}^{-1}(\mu \oplus \delta_{2,1} \oplus \delta_{2,2}, c) \oplus {f_2}^{-1}(\mu \oplus \delta_{2,2}, c) \oplus {f_2}^{-1}(\mu \oplus \delta_{2,2}, c)$;
    for $i = 0$ to $n - 1$ do
        if $\Delta \not\equiv_T 0^n$ then
            $k \leftarrow k + 1$;
return $k/Q$
```

We do not explicitly list the difference sets used in our attack, but conclude this section by giving the statistics for the best attacks that we found for various reduced versions of SHA-1 *without padding* in Table 11.2, the highest number of attacked rounds being 62.

4.2 One-block preimages with padding

If we want the message to be properly padded, 65 out of the 512 bits of the last message blocks need to be fixed according to the padding rules, and this naturally restricts the positions of where one can now use message differences. This has in particular an adverse effect on the differences in the backward step, whose Hamming weight increases because of some features of SHA-1's message expansion algorithm. The overall process of finding attack parameters is otherwise unchanged from the non-padded case. We give statistics for the best attacks that we found in Table 11.3. One will note that the highest number of attacked rounds dropped from 62 to 56 when compared to Table 11.2.

4.3 Two-block preimages with padding

We can increase the number of rounds for which we can find a preimage with a properly padded message at the cost of using a slightly longer message of two blocks: if we are

Table 11.2 – One block preimage without padding. N is the number of attacked steps, *Split* is the separation step between the forward and the backward chunk, $d_{i,j}$ is the $\log_2$ of the cardinal of $D_{i,j}$ and α is the estimate for the false negative probability. The complexity is computed as described in Section 3.

N	*Split*	$d_{1,1}$	$d_{1,2}$	$d_{2,1}$	$d_{2,2}$	α	Complexity
58	25	7.6	9.0	9.2	9.0	0.73	157.4
59	25	7.6	9.0	6.7	6.7	0.69	157.7
60	26	6.5	6.0	6.7	6.0	0.60	158.0
61	27	4.7	4.8	5.7	5.8	0.51	158.5
62	27	4.7	4.8	4.3	4.6	0.63	159.0

Table 11.3 – One block preimage with padding. N is the number of attacked steps, *Split* is the separation step between the forward and the backward chunk, $d_{i,j}$ is the $\log_2$ of the cardinal of $D_{i,j}$ and α is the estimation for false negative probability. The complexity is computed as described in Section 3.

N	*Split*	$d_{1,1}$	$d_{1,2}$	$d_{2,1}$	$d_{2,2}$	α	Complexity
51	23	8.7	8.7	8.7	8.7	0.72	155.6
52	23	9.1	9.1	8.2	8.2	0.61	156.7
53	23	9.1	9.1	3.5	3.5	0.61	157.7
55	25	6.5	6.5	5.9	5.7	0.52	158.2
56	25	6	6.2	7.2	7.2	0.6	159.4

able to find *one-block pseudo preimages with padding* on enough rounds, we can then use the *one-block preimage without padding* to bridge the former to the IV. Indeed, in a pseudo-preimage setting, the additional freedom degrees gained from removing any constraint on the IV more than compensate for the ones added by the padding. We first describe how to compute such pseudo-preimages.

4.3.1 One-block pseudo-preimages

If we relax the conditions on the IV and do not impose anymore that it is fixed to the one of the specifications, it becomes possible to use a splice-and-cut decomposition of the function, as well as short properly padded bicliques built in the same way as the ones used by Knellwolf and Khovratovich [KK12].

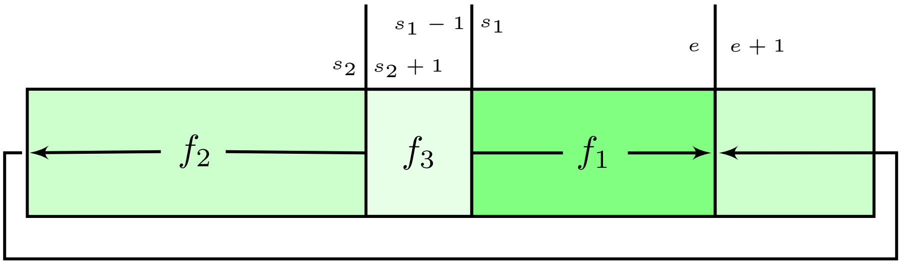


Figure 11.1 – A splice-and-cut decomposition using a biclique.

The reduced compression function of SHA-1 f is now decomposed into three smaller functions as $f = {f_2}^t \circ f_1 \circ f_3 \circ {f_2}^b$, f_3 being the rounds covered by the biclique. The function f_1 covers the steps s_1 to e, $f_2 = {f_2}^t \circ {f_2}^b$ covers s_2 to $e + 1$ through the feedforward, and f_3 covers $s_2 + 1$ to $s_1 - 1$, as shown in Figure 11.1.

Finding the parameters is done in the exact same way as for the one-block preimage attacks. Since we only use bicliques covering 7 steps, one can generate many of them from a single one by modifying some of the remaining message words outside of the biclique proper. Therefore, the amortised cost of their construction is small and considered negligible w.r.t. the rest of the attack. The statistics of the resulting attacks are shown in Table 11.4.

Table 11.4 – One block pseudo-preimage with padding. N is the number of attacked steps, $d_{i,j}$ is the $\log_2$ of the cardinal of the set $D_{1,2}$ and α is the estimate for the false negative probability. The various splits are labeled as in Figure 11.1. The complexity is computed as described in Section 3.

N	s_1	e	s_2	$d_{1,1}$	$d_{1,2}$	$d_{2,1}$	$d_{2,2}$	α	Complexity
61	27	49	20	7.0	7.0	7.5	7.5	0.56	156.4
62	27	50	20	5.8	5.7	7.2	7.2	0.57	157.0
63	27	50	20	6.7	6.7	7.7	7.7	0.57	157.6
64	27	50	20	3	3	4.5	4.7	0.69	158.7

4.3.2 Complexity of the two-block attacks

Using two one-block attacks, it is simple to mount a two-block attack at the combined cost of each of them. For a given target c, the process is the following:

1. The attacker uses a properly-padded pseudo-preimage attack, yielding the second message block μ_2 and an IV IV';

2. The attacker then uses a non-padded preimage attack with target IV', yielding a first message block μ_2.

From the Merkle-Damgård structure of SHA-1, it follows that the two-block message (μ_1, μ_2) is a preimage of c.

For attacks up to 60 rounds, we can use the pseudo-preimage attacks of [KK12]; for 61 and 62 rounds, we use the ones of this section. This results in attacks of complexities as shown in Table 11.5.

Table 11.5 – Two-block preimage attacks on SHA-1 reduced to N steps. The pseudo-preimage attacks followed by "$\star$" come from [KK12].

N	Second block complexity	First block complexity	Total Complexity
58	$156.3^\star$	157.4	157.9
59	$156.7^\star$	157.7	158.3
60	$157.5^\star$	158.0	158.7
61	156.4	158.5	158.8
62	157.0	159.0	159.3

5 Conclusion

This chapter showed how to extend the differential variant of the meet-in-the-middle framework for hash function preimage attacks with higher-order differentials, and how this could be applied to find better attacks on SHA-1. The source of the improvements comes from the fact that higher-order differentials lead to better message partitions, which then mechanically allow to mount attacks up to a larger number of steps than the previous best results.

This new technique is however not without downsides of its own, as it makes the attack procedure more complex. A consequence is that it does not seem to improve the best attacks on very weak functions such as MD4. Yet, it still produces the best known attacks on the more complex SHA-1, when the objective is to find a preimage for the highest possible number of rounds. It should also be noted that due to the strong similarity with the original meet-in-the-middle framework, this technique may be much less successful when applied to functions with a heavier message expansion, such as SHA-2, or in general to functions with better diffusion. A striking example in the latter case is that by applying the current framework to the BLAKE-512 hash function [AHMP10], we were only able to attack 2.75 out of 16 with a complexity close to a generic attack.

Contents

III New attacks on SHA-1 **133**

List of Figures

List of Tables

Bibliography

[ADK+14] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin, *Block Ciphers - Focus on the Linear Layer (feat. PRIDE)*, CRYPTO 2014 (Juan A. Garay and Rosario Gennaro, eds.), Lecture Notes in Computer Science, vol. 8616, Springer, 2014, pp. 57–76.

[AF13] Daniel Augot and Matthieu Finiasz, *Exhaustive Search for Small Dimension Recursive MDS Diffusion Layers for Block Ciphers and Hash Functions*, ISIT 2013, IEEE, 2013, pp. 1551–1555.

[AF15] Daniel Augot and Matthieu Finiasz, *Direct Construction of Recursive MDS Diffusion Layers using Shortened BCH Codes*, in Cid and Rechberger [CR15].

[AFK14] Daniel Augot, Pierre-Alain Fouque, and Pierre Karpman, *Diffusion Matrices from Algebraic-Geometry Codes with Efficient SIMD Implementation*, SAC 2014 (Antoine Joux and Amr M. Youssef, eds.), Lecture Notes in Computer Science, vol. 8781, Springer, 2014, pp. 243–260.

[AHMP10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan, *SHA-3 proposal BLAKE, version 1.3*, December 2010, `https://131002.net/blake/`.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner, *Ciphers for MPC and FHE*, in Oswald and Fischlin [OF15], pp. 430–454.

[AS08] Kazumaro Aoki and Yu Sasaki, *Preimage Attacks on One-Block MD4, 63-Step MD5 and More*, SAC 2008 (Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, eds.), Lecture Notes in Computer Science, vol. 5381, Springer, 2008, pp. 103–119.

[AS09] Kazumaro Aoki and Yu Sasaki, *Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1*, in Halevi [Hal09], pp. 70–89.

[Atm07] Atmel, *8-bit AVR Microcontroller with 1K Byte Flash*, Rev. 1006FS-AVR-06/07.

[Atm13] Atmel, *8-bit AVR Microcontroller with 8KBytes In-System Programmable Flash*, Rev. 2486AA-AVR-02/2013.

[BAK98] Eli Biham, Ross J. Anderson, and Lars R. Knudsen, *Serpent: A New Block Cipher Proposal*, FSE '98 (Serge Vaudenay, ed.), Lecture Notes in Computer Science, vol. 1372, Springer, 1998, pp. 222–238.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, and Benjamin Grégoire, *Compositional Verification of Higher-Order Masking: Application to a Verifying Masking Compiler*, IACR Cryptology ePrint Archive **2015** (2015), 506.

[BBK14] Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich, *Cryptographic Schemes Based on the ASASA Structure: Black-Box, White-Box, and Public-Key (Extended Abstract)*, ASIACRYPT 2014 (Palash Sarkar and Tetsu Iwata, eds.), Lecture Notes in Computer Science, vol. 8873, Springer, 2014, pp. 63–84.

[BC04] Eli Biham and Rafi Chen, *Near-Collisions of SHA-0*, in Franklin [Fra04], pp. 290–305.

[BC16] Christina Boura and Anne Canteaut, *Another View of the Division Property*, in Robshaw and Katz [RK16], pp. 654–682.

[BCD11] Christina Boura, Anne Canteaut, and Christophe De Cannière, *Higher-Order Differential Properties of Keccak and* Luffa, FSE 2011 (Antoine Joux, ed.), Lecture Notes in Computer Science, vol. 6733, Springer, 2011, pp. 252–269.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin, *PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract*, in Wang and Sako [WS12], pp. 208–225.

[BCJ+05] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby, *Collisions of SHA-0 and Reduced SHA-1*, in Cramer [Cra05], pp. 36–57.

[BCJ15] Eli Biham, Rafi Chen, and Antoine Joux, *Cryptanalysis of SHA-0 and Reduced SHA-1*, J. Cryptology **28** (2015), no. 1, 110–160.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk, *Keying Hash Functions for Message Authentication*, CRYPTO '96 (Neal Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer, 1996, pp. 1–15.

[BDPA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007.

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, *On the Indifferentiability of the Sponge Construction*, EUROCRYPT 2008 (Nigel P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, pp. 181–197.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, *The Keccak reference*, January 2011, `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[BGGM15] Razvan Barbulescu, Pierrick Gaudry, Aurore Guillevic, and François Morain, *Improving NFS for the Discrete Logarithm Problem in Non-prime Finite Fields*, in Oswald and Fischlin [OF15], pp. 129–155.

[BGLP13] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin, *Implementing Lightweight Block Ciphers on x86 Architectures*, in Lange et al. [LLL14], pp. 324–351.

[BHK$^+$99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway, *UMAC: Fast and Secure Message Authentication*, CRYPTO '99 (Michael J. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 216–233.

[BI15] Andrey Bogdanov and Takanori Isobe, *White-Box Cryptography Revisited: Space-Hard Ciphers*, CCS 2015 (Indrajit Ray, Ninghui Li, and Christopher Kruegel, eds.), ACM, 2015, pp. 1058–1069.

[Bir06] Alex Biryukov, *The Design of a Stream Cipher LEX*, SAC 2006 (Eli Biham and Amr M. Youssef, eds.), Lecture Notes in Computer Science, vol. 4356, Springer, 2006, pp. 67–75.

[BK03] Mihir Bellare and Tadayoshi Kohno, *A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications*, EUROCRYPT 2003 (Eli Biham, ed.), Lecture Notes in Computer Science, vol. 2656, Springer, 2003, pp. 491–506.

[BK12] Joppe W. Bos and Thorsten Kleinjung, *ECM at Work*, in Wang and Sako [WS12], pp. 467–484.

[BKL$^+$07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe, *PRESENT: An Ultra-Lightweight Block Cipher*, CHES 2007 (Pascal Paillier and Ingrid Verbauwhede, eds.), Lecture Notes in Computer Science, vol. 4727, Springer, 2007, pp. 450–466.

[BKR00] Mihir Bellare, Joe Kilian, and Phillip Rogaway, *The Security of the Cipher Block Chaining Message Authentication Code*, J. Comput. Syst. Sci. **61** (2000), no. 3, 362–399.

[BNN+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Vitkup, *Threshold implementations of small S-boxes*, Cryptography and Communications **7** (2015), no. 1, 3–33.

[BP95] Antoon Bosselaers and Bart Preneel (eds.), *Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*, Lecture Notes in Computer Science, vol. 1007, Springer, 1995.

[BR01] Paulo S.L.M. Barreto and Vincent Rijmen, *The KHAZAD Legacy-Level Block Cipher*, October 2001, `http://www.larc.usp.br/~pbarreto/KhazadPage.html`.

[BR03] Paulo S.L.M. Barreto and Vincent Rijmen, *The WHIRLPOOL Hashing Function*, May 2003, `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html`.

[BR14] Lejla Batina and Matthew Robshaw (eds.), *Cryptographic Hardware and Embedded Systems — CHES 2014*, Lecture Notes in Computer Science, vol. 8731, Springer, 2014.

[Bra90] Gilles Brassard (ed.), *Advances in Cryptology — CRYPTO '89*, Lecture Notes in Computer Science, vol. 435, Springer, 1990.

[BS01] Alex Biryukov and Adi Shamir, *Structural Cryptanalysis of SASAS*, EUROCRYPT 2001 (Birgit Pfitzmann, ed.), Lecture Notes in Computer Science, vol. 2045, Springer, 2001, pp. 394–405.

[BS12] Daniel J. Bernstein and Peter Schwabe, *NEON Crypto*, CHES 2012 (Emmanuel Prouff and Patrick Schaumont, eds.), Lecture Notes in Computer Science, vol. 7428, Springer, 2012, pp. 320–339.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers, *The SIMON and SPECK Families of Lightweight Block Ciphers*, IACR Cryptology ePrint Archive **2013** (2013), 404.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers, *SIMON and SPECK: Block Ciphers for the Internet of Things*, IACR Cryptology ePrint Archive **2015** (2015), 585, Presented at the NIST Lightweight Cryptography Workshop 2015.

[BTV16] Andrey Bogdanov, Elmar Tischhauser, and Philip S. Vejre, *Multivariate Linear Cryptanalysis: The Past and Future of PRESENT*, IACR Cryptology ePrint Archive **2016** (2016), 667.

[Can12] Anne Canteaut (ed.), *Fast Software Encryption — FSE 2012*, Lecture Notes in Computer Science, vol. 7549, Springer, 2012.

[CDL15] Anne Canteaut, Sébastien Duval, and Gaëtan Leurent, *Construction of Lightweight S-Boxes Using Feistel and MISTY Structures*, SAC 2015 (Orr Dunkelman and Liam Keliher, eds.), Lecture Notes in Computer Science, vol. 9566, Springer, 2015, pp. 373–393.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya, *Merkle-Damgård Revisited: How to Construct a Hash Function*, in Shoup [Sho05], pp. 430–448.

[CEJvO02] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot, *White-Box Cryptography and an AES Implementation*, SAC 2002 (Kaisa Nyberg and Howard M. Heys, eds.), Lecture Notes in Computer Science, vol. 2595, Springer, 2002, pp. 250–270.

[CG09] Christophe Clavier and Kris Gaj (eds.), *Cryptographic Hardware and Embedded Systems — CHES 2009*, Lecture Notes in Computer Science, vol. 5747, Springer, 2009.

[CGP⁺12] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain, *Higher-Order Masking Schemes for S-Boxes*, in Canteaut [Can12], pp. 366–384.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala, *Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity*, in Leander [Lea15], pp. 130–149.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala, *Secure Conversion between Boolean and Arithmetic Masking of Any Order*, in Batina and Robshaw [BR14], pp. 188–205.

[Cho10] Joo Yeon Cho, *Linear Cryptanalysis of Reduced-Round PRESENT*, CT-RSA 2010 (Josef Pieprzyk, ed.), Lecture Notes in Computer Science, vol. 5985, Springer, 2010, pp. 302–317.

[CJ98] Florent Chabaud and Antoine Joux, *Differential Collisions in SHA-0*, CRYPTO '98 (Hugo Krawczyk, ed.), Lecture Notes in Computer Science, vol. 1462, Springer, 1998, pp. 56–71.

[CJF⁺16] Tingting Cui, Keting Jia, Kai Fu, Shiyao Chen, and Meiqin Wang, *New Automatic Search Tool for Impossible Differentials and Zero-Correlation Linear Approximations*, IACR Cryptology ePrint Archive **2016** (2016), 689.

[CL04] Jia-Ping Chen and Chung-Chin Lu, *A Serial-In-Serial-Out Hardware Architecture for Systematic Encoding of Hermitian Codes via Gröbner Bases*, IEEE Transactions on Communications **52** (2004), no. 8, 1322–1332.

[CR15] Carlos Cid and Christian Rechberger (eds.), *Fast Software Encryption — FSE 2014*, Lecture Notes in Computer Science, vol. 8540, Springer, 2015.

[Cra05] Ronald Cramer (ed.), *Advances in Cryptology — EUROCRYPT 2005*, Lecture Notes in Computer Science, vol. 3494, Springer, 2005.

[CS09] Baudoin Collard and François-Xavier Standaert, *A Statistical Saturation Attack against the Block Cipher PRESENT*, CT-RSA 2009 (Marc Fischlin, ed.), Lecture Notes in Computer Science, vol. 5473, Springer, 2009, pp. 195–210.

[CS14] Shan Chen and John P. Steinberger, *Tight Security Bounds for Key-Alternating Ciphers*, in Nguyen and Oswald [NO14], pp. 327–350.

[CS15] Benoit Cogliati and Yannick Seurin, *On the Provable Security of the Iterated Even-Mansour Cipher Against Related-Key and Chosen-Key Attacks*, in Oswald and Fischlin [OF15], pp. 584–613.

[Dae95] Joan Daemen, *Cipher and Hash Function Design Strategies based on linear and differential cryptanalysis*, Ph.D. thesis, Katholieke Universiteit Leuven, 1995.

[Dam89] Ivan Damgård, *A Design Principle for Hash Functions*, in Brassard [Bra90], pp. 416–427.

[dB91] Bert den Boer and Antoon Bosselaers, *An Attack on the Last Two Rounds of MD4*, CRYPTO '91 (Joan Feigenbaum, ed.), Lecture Notes in Computer Science, vol. 576, Springer, 1991, pp. 194–203.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*, in Gollmann [Gol96], pp. 71–82.

[DDKL15] Itai Dinur, Orr Dunkelman, Thorsten Kranz, and Gregor Leander, *Decomposing the ASASA Block Cipher Construction*, IACR Cryptology ePrint Archive **2015** (2015), 507.

[Dea99] Richard Drews Dean, *Formal Aspects of Mobile Code Security*, Ph.D. thesis, Princeton University, January 1999.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel, *Related-Key Forgeries for Prøst-OTR*, in Leander [Lea15], pp. 282–296.

[DH77] Whitfield Diffie and Martin E. Hellman, *Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard*, Computer **10** (1977), 74–84.

[Din15] Itai Dinur, *Cryptanalytic Time-Memory-Data Tradeoffs for FX-Constructions with Applications to PRINCE and PRIDE*, in Oswald and Fischlin [OF15], pp. 231–253.

[DK08] Orr Dunkelman and Nathan Keller, *A New Attack on the LEX Stream Cipher*, ASIACRYPT 2008 (Josef Pieprzyk, ed.), Lecture Notes in Computer Science, vol. 5350, Springer, 2008, pp. 539–556.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen, *The Block Cipher Square*, FSE 1997 (Eli Biham, ed.), Lecture Notes in Computer Science, vol. 1267, Springer, 1997, pp. 149–165.

[DKS12] Orr Dunkelman, Nathan Keller, and Adi Shamir, *Minimalism in Cryptography: The Even-Mansour Scheme Revisited*, EUROCRYPT 2012 (David Pointcheval and Thomas Johansson, eds.), Lecture Notes in Computer Science, vol. 7237, Springer, 2012, pp. 336–354.

[DLPR13] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain, *White-Box Security Notions for Symmetric Encryption Schemes*, in Lange et al. [LLL14], pp. 247–264.

[Dob96] Hans Dobbertin, *Cryptanalysis of MD4*, in Gollmann [Gol96], pp. 53–69.

[DPAR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen, *The* NOEKEON *Block Cipher*, Nessie Proposal, October 2000.

[DR02] Joan Daemen and Vincent Rijmen, *The Design of Rijndael: AES — The Advanced Encryption Standard*, Information Security and Cryptography, Springer, 2002.

[DR06] Christophe De Cannière and Christian Rechberger, *Finding SHA-1 Characteristics: General Results and Applications*, ASIACRYPT 2006 (Xuejia Lai and Kefei Chen, eds.), Lecture Notes in Computer Science, vol. 4284, Springer, 2006, pp. 1–20.

[DR07] Joan Daemen and Vincent Rijmen, *Probability distributions of correlation and differentials in block ciphers*, J. Mathematical Cryptology **1** (2007), no. 3, 221–242.

[DR08] Christophe De Cannière and Christian Rechberger, *Preimages for Reduced SHA-0 and SHA-1*, CRYPTO 2008 (David Wagner, ed.), Lecture Notes in Computer Science, vol. 5157, Springer, 2008, pp. 179–202.

[Duu99] Iwan Duursma, *Weight distributions of geometric Goppa codes*, Transactions of the American Mathematical Society **351** (1999), no. 9, 3609–3639.

[EFK15] Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman, *Higher-Order Differential Meet-in-the-middle Preimage Attacks on SHA-1 and BLAKE*, in Gennaro and Robshaw [GR15], pp. 683–701.

[EM91] Shimon Even and Yishay Mansour, *A Construction of a Cipher From a Single Pseudorandom Permutation*, ASIACRYPT '91 (Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, eds.), Lecture Notes in Computer Science, vol. 739, Springer, 1991, pp. 210–224.

[FK13] Pierre-Alain Fouque and Pierre Karpman, *Security Amplification against Meet-in-the-Middle Attacks Using Whitening*, IMACC 2013 (Martijn Stam, ed.), Lecture Notes in Computer Science, vol. 8308, Springer, 2013, pp. 252–269.

[FKKM16] Pierre-Alain Fouque, Pierre Karpman, Paul Kirchner, and Brice Minaud, *Efficient and Provable White-Box Primitives*, ASIACRYPT 2016 (Jung Hee Cheon and Tsuyoshi Takagi, eds.), Lecture Notes in Computer Science, vol. 10031, 2016, pp. 159–188.

[For15a] CA/Browser Forum, *Ballot 152 - Issuance of SHA-1 certificates through 2016*, Cabforum mailing list, 2015, `https://cabforum.org/pipermail/public/2015-October/006048.html`.

[For15b] CA/Browser Forum, *Ballot 152 - Issuance of SHA-1 certificates through 2016*, Cabforum mailing list, 2015, `https://cabforum.org/pipermail/public/2015-October/006081.html`.

[FP15] Pooya Farshim and Gordon Procter, *The Related-Key Security of Iterated Even-Mansour Ciphers*, in Leander [Lea15], pp. 342–363.

[Fra04] Matthew K. Franklin (ed.), *Advances in Cryptology — CRYPTO 2004*, Lecture Notes in Computer Science, vol. 3152, Springer, 2004.

[Ful08] William Fulton, *Algebraic Curves — An Introduction to Algebraic Geometry*, January 2008.

[FY48] Ronald A. Fisher and Frank Yates, *Statistical tables for biological, agricultural and medical research*, 3 ed., Oliver and Boyd, 1948.

[GA11] Evgeny A. Grechnikov and Andrew V. Adinetz, *Collision for 75-step SHA-1: Intensive Parallelization with GPU*, IACR Cryptology ePrint Archive **2011** (2011), 641.

[GGNS13] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert, *Block Ciphers That Are Easier to Mask: How Far Can We Go?*, CHES 2013 (Guido Bertoni and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 8086, Springer, 2013, pp. 383–399.

[Gil16] Henri Gilbert, *On White-Box Cryptography*, Invited talk at FSE 2016, March 2016.

[GKN$^+$14] Jian Guo, Pierre Karpman, Ivica Nikolić, Lei Wang, and Shuang Wu, *Analysis of BLAKE2*, CT-RSA 2014 (Josh Benaloh, ed.), Lecture Notes in Computer Science, vol. 8366, Springer, 2014, pp. 402–423.

[GLRW10] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang, *Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2*, ASIACRYPT 2010 (Masayuki Abe, ed.), Lecture Notes in Computer Science, vol. 6477, Springer, 2010, pp. 56–75.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici, *LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations*, in Cid and Rechberger [CR15], pp. 18–37.

[Gol96] Dieter Gollmann (ed.), *Fast Software Encryption — FSE '96*, Lecture Notes in Computer Science, vol. 1039, Springer, 1996.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann, *The PHOTON Family of Lightweight Hash Functions*, CRYPTO 2011 (Phillip Rogaway, ed.), Lecture Notes in Computer Science, vol. 6841, Springer, 2011, pp. 222–239.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw, *The LED Block Cipher*, CHES 2011 (Bart Preneel and Tsuyoshi Takagi, eds.), Lecture Notes in Computer Science, vol. 6917, Springer, 2011, pp. 326–341.

[GPT15] Henri Gilbert, Jérôme Plût, and Joana Treger, *Key-Recovery Attack on the ASASA Cryptosystem with Expanding S-Boxes*, in Gennaro and Robshaw [GR15], pp. 475–490.

[GR15] Rosario Gennaro and Matthew Robshaw (eds.), *Advances in Cryptology — CRYPTO 2015*, Lecture Notes in Computer Science, vol. 9215, Springer, 2015.

[Hal09] Shai Halevi (ed.), *Advances in Cryptology — CRYPTO 2009*, Lecture Notes in Computer Science, vol. 5677, Springer, 2009.

[Ham09] Mike Hamburg, *Accelerating AES with Vector Permute Instructions*, in Clavier and Gaj [CG09], pp. 18–32.

[HLS95] Chris Heegard, J. Little, and Keith Saints, *Systematic Encoding via Gröbner Bases for a Class of Algebraic-Geometric Goppa Codes*, IEEE Transactions on Information Theory **41** (1995), no. 6, 1752–1761.

[IMGM14] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, and Sumio Morioka, *CLOC: Authenticated Encryption for Short Input*, in Cid and Rechberger [CR15], pp. 149–167.

[Int14] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2014.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner, *Private Circuits: Securing Hardware against Probing Attacks*, CRYPTO 2003 (Dan Boneh, ed.), Lecture Notes in Computer Science, vol. 2729, Springer, 2003, pp. 463–481.

[Jea] Jérémy Jean, *TiKZ for Cryptographers*, `https://www.iacr.org/authors/tikz`.

[Jel15] Hamza Jeljeli, *Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis. (Hardware and Software Accelerators for Sparse Linear Algebra over Finite Fields)*, Ph.D. thesis, University of Lorraine, Nancy, France, 2015.

[JN13] Thomas Johansson and Phong Q. Nguyen (eds.), *Advances in Cryptology — EUROCRYPT 2013*, Lecture Notes in Computer Science, vol. 7881, Springer, 2013.

[Jou04] Antoine Joux, *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions*, in Franklin [Fra04], pp. 306–316.

[Jou09] Antoine Joux, *Algorithmic cryptanalysis*, CRC Press, 2009.

[JP07] Antoine Joux and Thomas Peyrin, *Hash Functions and the (Amplified) Boomerang Attack*, CRYPTO 2007 (Alfred Menezes, ed.), Lecture Notes in Computer Science, vol. 4622, Springer, 2007, pp. 244–263.

[JV04] Pascal Junod and Serge Vaudenay, *FOX : A New Family of Block Ciphers*, SAC 2004 (Helena Handschuh and M. Anwar Hasan, eds.), Lecture Notes in Computer Science, vol. 3357, Springer, 2004, pp. 114–129.

[Kar15] Pierre Karpman, *From Distinguishers to Key Recovery: Improved Related-Key Attacks on Even-Mansour*, ISC 2015 (Javier Lopez and Chris J. Mitchell, eds.), Lecture Notes in Computer Science, vol. 9290, Springer, 2015, pp. 177–188.

[KG16] Pierre Karpman and Benjamin Grégoire, *The* Littlun *S-box and the* Fly *Block Cipher*, 2016.

[KK12] Simon Knellwolf and Dmitry Khovratovich, *New Preimage Attacks against Reduced SHA-1*, in Safavi-Naini and Canetti [SC12], pp. 367–383.

[KLL⁺14] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın, *Prøst*, CAESAR Proposal, 2014.

[KPS15] Pierre Karpman, Thomas Peyrin, and Marc Stevens, *Practical Free-Start Collision Attacks on 76-step SHA-1*, in Gennaro and Robshaw [GR15], pp. 623–642.

[KR01] Joe Kilian and Phillip Rogaway, *How to Protect DES Against Exhaustive Key Search (an Analysis of DESX)*, J. Cryptology **14** (2001), no. 1, 17–35.

[KRS12] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva, *Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family*, in Canteaut [Can12], pp. 244–263.

[KS05] John Kelsey and Bruce Schneier, *Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work*, in Cramer [Cra05], pp. 474–490.

[KS09] Emilia Käsper and Peter Schwabe, *Faster and Timing-Attack Resistant AES-GCM*, in Clavier and Gaj [CG09], pp. 1–17.

[KW13] Lars R. Knudsen and Huapeng Wu (eds.), *Selected Areas in Cryptography — SAC 2012*, Lecture Notes in Computer Science, vol. 7707, Springer, 2013.

[Lai94] Xuejia Lai, *Higher Order Derivatives and Differential Cryptanalysis*, Communications and Cryptography, Springer, 1994, pp. 227–233.

[Lea15] Gregor Leander (ed.), *Fast Software Encryption — FSE 2015*, Lecture Notes in Computer Science, vol. 9054, Springer, 2015.

[Lin99] Jacobus H. Van Lint, *Introduction to Coding Theory*, 3 ed., Graduate Texts in Mathematics, vol. 86, Springer, 1999.

[LLL14] Tanja Lange, Kristin Lauter, and Petr Lisonek (eds.), *Selected Areas in Cryptography — SAC 2013*, Lecture Notes in Computer Science, vol. 8282, Springer, 2014.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy, *Markov Ciphers and Differential Cryptanalysis*, EUROCRYPT '91 (Donald W. Davies, ed.), Lecture Notes in Computer Science, vol. 547, Springer, 1991, pp. 17–38.

[LMR15] Gregor Leander, Brice Minaud, and Sondre Rønjom, *A Generic Approach to Invariant Subspace Attacks: Cryptanalysis of Robin, iSCREAM and Zorro*, in Oswald and Fischlin [OF15], pp. 254–283.

[LP07] Gregor Leander and Axel Poschmann, *On the Classification of 4 Bit S-Boxes*, WAIFI 2007 (Claude Carlet and Berk Sunar, eds.), Lecture Notes in Computer Science, vol. 4547, Springer, 2007, pp. 159–176.

[LP13] Franck Landelle and Thomas Peyrin, *Cryptanalysis of Full RIPEMD-128*, in Johansson and Nguyen [JN13], pp. 228–244.

[Luc05] Stefan Lucks, *A Failure-Friendly Design Principle for Hash Functions*, ASIACRYPT 2005 (Bimal K. Roy, ed.), Lecture Notes in Computer Science, vol. 3788, Springer, 2005, pp. 474–494.

[Man11] Stéphane Manuel, *Classification and generation of disturbance vectors for collision attacks against SHA-1*, Des. Codes Cryptography **59** (2011), no. 1-3, 247–263.

[MBKL14] Andrea Miele, Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra, *Cofactorization on Graphics Processing Units*, in Batina and Robshaw [BR14], pp. 335–352.

[MDFK] Brice Minaud, Patrick Derbez, Pierre-Alain Fouque, and Pierre Karpman, *Key-Recovery Attacks on ASASA*, J. Cryptology, Invited, under submission.

[MDFK15] Brice Minaud, Patrick Derbez, Pierre-Alain Fouque, and Pierre Karpman, *Key-Recovery Attacks on ASASA*, ASIACRYPT 2015 (Tetsu Iwata and Jung Hee Cheon, eds.), Lecture Notes in Computer Science, vol. 9453, Springer, 2015, pp. 3–27.

[Men16] Bart Mennink, *XPX: Generalized Tweakable Even-Mansour with Improved Security Guarantees*, in Robshaw and Katz [RK16], pp. 64–94.

[Mer87] Ralph C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*, CRYPTO '87 (Carl Pomerance, ed.), Lecture Notes in Computer Science, vol. 293, Springer, 1987, pp. 369–378.

[Mer89] Ralph C. Merkle, *One Way Hash Functions and DES*, in Brassard [Bra90], pp. 428–446.

[Mic15] Microsoft, *SHA-1 Deprecation Update*, Microsoft blog, 2015.

[Min14] Kazuhiko Minematsu, *Parallelizable Rate-1 Authenticated Encryption from Pseudorandom Functions*, in Nguyen and Oswald [NO14], pp. 275–292.

[Moz15] Mozilla, *Continuing to Phase Out SHA-1 Certificates*, Mozilla Security Blog, 2015.

[MP08] Stéphane Manuel and Thomas Peyrin, *Collisions on SHA-0 in One Hour*, in Nyberg [Nyb08], pp. 16–35.

[MPRR06] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, *The Impact of Carries on the Complexity of Collision Attacks on SHA-1*, FSE 2006 (Matthew J. B. Robshaw, ed.), Lecture Notes in Computer Science, vol. 4047, Springer, 2006, pp. 278–292.

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein, *Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology*, TCC 2004 (Moni Naor, ed.), Lecture Notes in Computer Science, vol. 2951, Springer, 2004, pp. 21–39.

[MRS14] Florian Mendel, Vincent Rijmen, and Martin Schläffer, *Collision Attack on 5 Rounds of Grøstl*, in Cid and Rechberger [CR15], pp. 509–521.

[MS06] Florence Jessie MacWilliams and Neil James Alexander Sloane, *The Theory of Error-Correcting Codes*, 12 ed., North-Holland Mathematical Library, vol. 16, North-Holland, 2006.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[NIS93] National Institute of Standards and Technology, *FIPS 180: Secure Hash Standard*, May 1993, *Available at:* http://pages.saclay.inria.fr/pierre.karpman/fips_180.pdf.

[NIS95] National Institute of Standards and Technology, *FIPS 180-1: Secure Hash Standard*, April 1995, *Available at:* http://pages.saclay.inria.fr/pierre.karpman/fips_180-1.pdf.

[NIS01] National Institute of Standards and Technology, *FIPS 197: Advanced Encryption Standard (AES)*, November 2001.

[NIS15a] National Institute of Standards and Technology, *FIPS 180-4: Secure Hash Standard*, August 2015.

[NIS15b] National Institute of Standards and Technology, *FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015.

[NIS16] National Institute of Standards and Technology, *NISTIR 8114: DRAFT Report on Lightweight Cryptography*, August 2016.

[NO14] Phong Q. Nguyen and Elisabeth Oswald (eds.), *Advances in Cryptology — EUROCRYPT 2014*, Lecture Notes in Computer Science, vol. 8441, Springer, 2014.

[NVIa] Nvidia Corporation, *CUDA C Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[NVIb] Nvidia Corporation, *CUDA Toolkit*, https://docs.nvidia.com/cuda.

[NVIc] Nvidia Corporation, *Nvidia Geforce GTX 970 Specifications*, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications.

[Nyb08] Kaisa Nyberg (ed.), *Fast Software Encryption — FSE 2008*, Lecture Notes in Computer Science, vol. 5086, Springer, 2008.

[O'C95] Luke O'Connor, *On the Distribution of Characteristics in Bijective Mappings*, J. Cryptology **8** (1995), no. 2, 67–86.

[OF15] Elisabeth Oswald and Marc Fischlin (eds.), *Advances in Cryptology — EUROCRYPT 2015*, Lecture Notes in Computer Science, vol. 9056, Springer, 2015.

[Pey08] Thomas Peyrin, *Analyse de fonctions de hachage cryptographiques*, Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines, November 2008.

[PGV93] Bart Preneel, René Govaerts, and Joos Vandewalle, *Hash Functions Based on Block Ciphers: A Synthetic Approach*, CRYPTO '93 (Douglas R. Stinson, ed.), Lecture Notes in Computer Science, vol. 773, Springer, 1993, pp. 368–378.

[Pos09] Axel Poschmann, *Lightweight Cryptography*, Ph.D. thesis, Ruhr-Universität Bochum, 2009.

[RDP⁺96] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win, *The Cipher SHARK*, in Gollmann [Gol96], pp. 99–111.

[Riv90] Ronald L. Rivest, *The MD4 Message Digest Algorithm*, CRYPTO '90 (Alfred Menezes and Scott A. Vanstone, eds.), Lecture Notes in Computer Science, vol. 537, Springer, 1990, pp. 303–311.

[Riv92] Ronald L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992.

[RK16] Matthew Robshaw and Jonathan Katz (eds.), *Advances in Cryptology — CRYPTO 2016*, Lecture Notes in Computer Science, vol. 9814, Springer, 2016.

[Rou15] Joëlle Roué, *Analyse de la résistance des chiffrements par blocs aux attaques linéaires et différentielles. (On the resistance of block ciphers to differential and linear cryptanalyses)*, Ph.D. thesis, Pierre and Marie Curie University, Paris, France, 2015.

[SA09] Yu Sasaki and Kazumaro Aoki, *Finding Preimages in Full MD5 Faster Than Exhaustive Search*, EUROCRYPT 2009 (Antoine Joux, ed.), Lecture Notes in Computer Science, vol. 5479, Springer, 2009, pp. 134–152.

[Saa03] Markku-Juhani Olavi Saarinen, *Cryptanalysis of Block Ciphers Based on SHA-1 and MD5*, FSE 2003 (Thomas Johansson, ed.), Lecture Notes in Computer Science, vol. 2887, Springer, 2003, pp. 36–44.

[Saa11] Markku-Juhani O. Saarinen, *Cryptographic Analysis of All 4 × 4-Bit S-Boxes*, SAC 2011 (Ali Miri and Serge Vaudenay, eds.), Lecture Notes in Computer Science, vol. 7118, Springer, 2011, pp. 118–133.

[SC12] Reihaneh Safavi-Naini and Ran Canetti (eds.), *Advances in Cryptology — CRYPTO 2012*, Lecture Notes in Computer Science, vol. 7417, Springer, 2012.

[Sch12] Bruce Schneier, *When Will We See Collisions for SHA-1?*, Schneier on Security, 2012.

[SDMS12] Mahdi Sajadieh, Mohammad Dakhilalian, Hamid Mala, and Pouyan Sepehrdad, *Recursive Diffusion Layers for Block Ciphers and Hash Functions*, in Canteaut [Can12], pp. 385–401.

[SH95] Keith Saints and Chris Heegard, *Algebraic-geometric codes and multidimensional cyclic codes: a unified theory and algorithms for decoding using Grobner bases*, IEEE Trans. Information Theory **41** (1995), no. 6, 1733–1751.

[Sho05] Victor Shoup (ed.), *Advances in Cryptology — CRYPTO 2005*, Lecture Notes in Computer Science, vol. 3621, Springer, 2005.

[SKP16] Marc Stevens, Pierre Karpman, and Thomas Peyrin, *Freestart Collision for Full SHA-1*, EUROCRYPT 2016 (Marc Fischlin and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 9665, Springer, 2016, pp. 459–483.

[SMMK12] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi, *TWINE: A Lightweight Block Cipher for Multiple Platforms*, in Knudsen and Wu [KW13], pp. 339–354.

[SOR$^+$14] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar, *Microcontrollers as (In)Security Devices for Pervasive Computing Applications*, Proceedings of the IEEE **102** (2014), no. 8, 1157–1173.

[SSA$^+$09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger, *Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate*, in Halevi [Hal09], pp. 55–69.

[STA$^+$14] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose, *Minalpher*, CAESAR Proposal, 2014.

[Ste12] Marc Stevens, *Attacks on Hash Functions and Applications*, Ph.D. thesis, Leiden University, June 2012.

[Ste13] Marc Stevens, *New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis*, in Johansson and Nguyen [JN13], pp. 245–261.

[Sti09] Henning Stichtenoth, *Algebraic Function Fields and Codes*, 2 ed., Graduate Texts in Mathematics, vol. 254, Springer, 2009.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir, *Efficient Cache Attacks on AES, and Countermeasures*, J. Cryptology **23** (2010), no. 1, 37–71.

[TVN07] Michael Tsfasman, Serge Vlăduţ, and Dmitry Nogin, *Algebraic Geometric Codes: Basic Notions*, Mathematical Surveys and Monographs, vol. 139, American Mathematical Society, 2007.

[UDI+11] Markus Ullrich, Christophe De Cannière, Sebastian Indesteege, Özgül Küçük, Nicky Mouha, and Bart Preneel, *Finding Optimal Bitsliced Implementations of* 4 × 4*-bit S-boxes*, SKEW 2011, 2011.

[WS12] Xiaoyun Wang and Kazue Sako (eds.), *Advances in Cryptology — ASIACRYPT 2012*, Lecture Notes in Computer Science, vol. 7658, Springer, 2012.

[WWW12] Shengbao Wu, Mingsheng Wang, and Wenling Wu, *Recursive Diffusion Layers for (Lightweight) Block Ciphers and Hash Functions*, in Knudsen and Wu [KW13], pp. 355–371.

[WYY05a] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, *Finding Collisions in the Full SHA-1*, in Shoup [Sho05], pp. 17–36.

[WYY05b] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin, *Efficient Collision Search Attacks on SHA-0*, in Shoup [Sho05], pp. 1–16.

[YIN+08] Jun Yajima, Terutoshi Iwasaki, Yusuke Naito, Yu Sasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A strict evaluation method on the number of conditions for the SHA-1 collision search*, ASIACCS 2008 (Masayuki Abe and Virgil D. Gligor, eds.), ACM, 2008, pp. 10–20.

[YSN+07] Jun Yajima, Yu Sasaki, Yusuke Naito, Terutoshi Iwasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A New Strategy for Finding a Differential Path of SHA-1*, ACISP 2007 (Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, eds.), Lecture Notes in Computer Science, vol. 4586, Springer, 2007, pp. 45–58.

[ZBL+14] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede, *RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms*, IACR Cryptology ePrint Archive **2014** (2014), 84.

[ZL12] Jinmin Zhong and Xuejia Lai, *Improved preimage attack on one-block MD4*, Journal of Systems and Software **85** (2012), no. 4, 981–994.

[ZRHD08] Muhammad Reza Z'aba, Håvard Raddum, Matthew Henricksen, and Ed Dawson, *Bit-Pattern Based Integral Attack*, in Nyberg [Nyb08], pp. 363–381.

[ZWWD14] Jingyuan Zhao, Xiaoyun Wang, Meiqin Wang, and Xiaoyang Dong, *Differential Analysis on Block Cipher PRIDE*, IACR Cryptology ePrint Archive **2014** (2014), 525.

université
PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies
de l'information
et de la communication (STIC)

Titre : Analyse de primitives symétriques

Mots clefs : Cryptographie symétrique, Chiffres par bloc, Fonctions de hachage

Résumé : Cette thèse a pour objet d'étude les algorithmes de chiffrement par bloc et les fonctions de hachage cryptographiques, qui sont deux primitives essentielles de la cryptographie dite « symétrique ».

Dans une première partie, nous étudions des éléments utiles pour la conception de chiffres par bloc: tout d'abord des matrices de diffusion de grande dimension issues de codes correcteurs géométriques, puis une boîte de substitution offrant une bonne diffusion. Dans le second cas, nous montrons aussi comment utiliser cet élément pour construire un chiffre compact et efficace sur petits processeurs.

Dans une seconde partie, nous nous intéressons à des attaques en collision à initialisation libre sur la fonction de hachage SHA-1. Nous montrons comment les attaques classiques sur cette fonction peuvent être rendues plus efficaces en exploitant la liberté supplémentaire offerte par ce modèle. Ceci nous permet en particulier de calculer explicitement des collisions pour la fonction de compression de SHA-1 non réduite.

Title : Analysis of symmetric primitives

Keywords : Symmetric cryptography, Block ciphers, Hash functions

Abstract : This thesis focuses on block ciphers and cryptographic hash functions, which are two essential primitives of symmetric-key cryptography.

In the first part of this manuscript, we study useful building blocks for block cipher design. We first consider large diffusion matrices built from algebraic-geometry codes, and then construct a small S-box with good diffusion. In the second case, we show how the S-box can be used to define a compact and efficient block cipher targeting small processors.

In the second part, we focus on the SHA-1 hash function, for which we develop a freestart collision attack. We show how classical collision attacks can be made more efficient by exploiting the additional freedom provided by this model. This allows us in particular to compute explicit collisions for the full compression function of SHA-1.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France