Université Paris Diderot
(Paris 7)

École Normale Supérieure
Équipe Crypto

Thèse de doctorat

Construction et Analyse de Fonctions de Hachage

Spécialité : Informatique

présentée et soutenue publiquement le 30 septembre 2010 par

Gaëtan Leurent

pour obtenir le grade de

Docteur de l'université Paris Diderot

devant le jury composé de

Directeur de thèse :	David Pointcheval	(CNRS & École Normale Supérieure, France)
Responsable scientifique :	Pierre-Alain Fouque	(École Normale Supérieure, France)
Rapporteurs :	Anne Canteaut	(INRIA-Rocquencourt, France)
	Bart Preneel	(Katholieke Universiteit Leuven, Belgique)
Examinateurs :	Alex Biryukov	(Université du Luxembourg, Luxembourg)
	Orr Dunkelman	(Weizmann Institute of Science, Israël)
	Arnaud Durand	(Université Paris 7, France)
	Antoine Joux	(Université de Versailles & DGA, France)

Remerciements

Je tiens tout d'abord à remercier l'ENS, qui m'a fourni un environnement agréable, stimulant et épanouissant pendant mes années de normalien, et ensuite durant ma thèse dans l'équipe Crypto. Plus que la qualité des cours, c'est la liberté de cursus, l'ambiance, et les élèves qui font la richesse de cette École.

Merci à tous mes amis qui ont partagé ces années avec moi, et qui ont fait de Paris la plus belle ville du monde. Je ne vais pas tenter de faire une liste exhaustive, mais je suis sûr qu'ils se reconnaîtront. Merci aussi à toute l'équipe Crypto avec qui j'ai eu plaisir à travailler.

Je suis reconnaissant en particulier à tous ceux qui ont aiguillé mon parcours pour m'amener jusqu'ici : mes parents, bien sûr, qui ont cultivé mon esprit scientifique et m'ont toujours encouragé dans mes études, puis mes professeurs d'informatique en prépa qui m'ont fait découvrir l'algorithmique, et Louis Granboulan qui s'occupait du cours de cryptographie symétrique. C'est ensuite à Pierre-Alain Fouque et Phong Nguyen que je dois d'être tombé dans les fonctions de hachage. Le jeu entre les concepteurs et les cryptanalystes, et la façon dont chaque attaque demande d'inventer de nouvelles techniques m'a tellement plu que j'ai continué jusque ici, encadré par Pierre-Alain. Pierre-Alain a su me diriger en me laissant une grande liberté d'action, et la réussite de ma thèse lui doit probablement beaucoup.

J'ai eu l'occasion d'écrire de nombreux articles durant ces quatre ans de thèse et je remercie tous mes co-auteurs pour les collaborations fructueuses que nous avons pu mener.

Je remercie aussi les organismes qui ont financé mes recherches et mes déplacements durant ma thèse : la DGA pour ma bourse de thèse, l'ENS et l'INRIA pour l'environnement matériel, le projet européen ECRYPT (puis ECRYPT II), et le projet français SAPHIR (puis SAPHIR II).

Enfin, je remercie chaleureusement Anne Canteaut et Bart Preneel qui ont dû relire un manuscrit imposant, ainsi que les autres relecteurs qui m'ont signalé des erreurs dans le manuscrit : Pierre-Alain bien sûr, mais aussi Yann et Orr. Je remercie aussi les membres du Jury d'avoir accepté d'être là aujourd'hui, en particulier ceux qui viennent de l'étranger.

À Los Pumas

Sommaire

Table des matières

Notations

F	une fonction de hachage
H	un haché
M	un message
f	une fonction de compression
x	une variable de chaînage
m	un bloc de message
$\mathbb{Z}_{2^{32}}$	l'anneau des entiers modulo 2^{32}
$\mathbb{F}_{257}$	le corps à 257 éléments
$x^{\lll k}$	une rotation de k bits appliquée à x
$\boxplus$	addition modulaire
$\vee$	ou bit-à-bit (opération booléenne)
$\wedge$	et bit-à-bit (opération booléenne)
$\oplus$	ou exclusif bit-à-bit (opération booléenne)
$\mathbb{0}$	le mot de 32-bit composé uniquement de zéros (`0x00000000`)
$\mathbb{1}$	le mot de 32-bit composé uniquement de uns (`0xffffffff`)
$\overline{x}$	le complémentaire de x
$x^{[k]}$	le k+1-ème bit de x, *i.e.* $x^{[k]} = (x \ggg k) \bmod 2$
$x \leftarrow \$$	x est choisi aléatoirement

On utilisera des indices entre 0 et $n-1$ pour indexer n éléments.

Chapitre 1

Introduction

1.1 Introduction à la cryptographie

La cryptographie est utilisée depuis l'antiquité pour les communications militaires, mais elle prend toute son importance au vingtième siècle avec le développement des télécommunications. Elle permet d'assurer la confidentialité des données, mais aussi de garantir l'intégrité d'un document et d'identifier son auteur de façon certaine.

Ces problèmes se posent dans notre vie quotidienne : même si la confidentialité est rarement nécessaire, il est souvent utile d'identifier les utilisateurs d'un service, par exemple pour accéder à un réseau de téléphonie mobile, pour payer par carte de crédit, ou pour accéder à un réseau Wifi. Pour réaliser cette authentification, l'utilisateur possède un secret — par exemple un mot de passe — et un protocole cryptographique permet de vérifier cette information sans qu'un tiers puisse apprendre le secret.

La cryptographie a pu prendre une place importante dans notre société grâce au développement de l'informatique. Un système cryptographique est décrit par une série d'opérations à effectuer sur un message, un algorithme. Jusqu'à la deuxième guerre mondiale, ces opérations étaient effectuées à la main ou avec une machine mécanique. L'évolution de ces machines a donné naissance aux premiers ordinateurs ; Alan Turing, considéré comme le père de l'informatique, était un cryptographe britannique pendant la deuxième guerre mondiale. Les ordinateurs sont maintenant tellement répandus et les processeurs tellement petits qu'on peut intégrer des algorithmes cryptographiques partout : un téléphone cellulaire, une carte de crédit, certains passeports, ou même un passe *Navigo* effectuent des calculs cryptographiques.

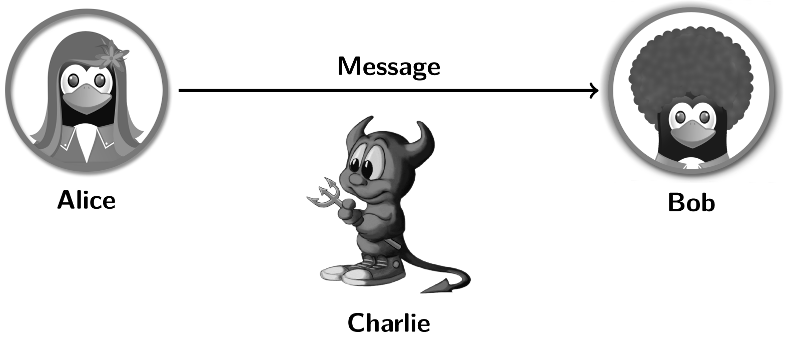
La cryptographie moderne suit les principes de Kerckhoffs, énoncés en 1883 [Ker83] : un système cryptographique doit utiliser un algorithme public, et seule une petite information, la clef, doit être gardée secrète. Ainsi, si la clef tombe aux mains de l'adversaire, on peut facilement la changer sans avoir à reconstruire tout le système en place. De plus, cela permet à l'algorithme d'être évalué publiquement. C'est ainsi que fonctionne la recherche académique en cryptographie : certains chercheurs proposent des systèmes cryptographiques, et la communauté scientifique étudie ces systèmes pour y trouver des faiblesses.

Quand on construit un système cryptographique, on précise le niveau de sécurité qu'il est censé atteindre. La sécurité est généralement définie par rapport aux attaques génériques *i.e.* les attaques qui s'appliquent à n'importe quelle construction. Pour un schéma de chiffrement, par exemple, les attaques génériques consistent à essayer toutes les clefs possibles jusqu'à trouver la bonne. Les paramètres du système sont choisis de façon à ce que ces attaques génériques soient impossibles à mettre en œuvre en pratique, même avec une très grande puissance de calcul. Quand une analyse montre qu'un système peut être attaqué plus facilement que prévu, on considère que le système est cassé, et on recommande d'utiliser un autre système, même si les attaques restent théoriques et ne sont pas réalisables en pratique.

Les interactions entre constructions et analyses sont très importantes, et permettent de se convaincre de la sécurité des algorithmes utilisés. On ne pourra jamais avoir de preuve de sécurité absolue, mais si un algorithme a été étudié pendant plusieurs années par des cryptographes émérites, il y a peu de chances qu'un attaquant trouve une faille *dans l'algorithme*. Cependant, il peut y avoir des problèmes dans l'implémentation de l'algorithme, ou dans son utilisation.

1.1.1 Aperçu des systèmes cryptographiques

Pour décrire un système cryptographique, on imagine une situation où deux utilisateurs (Alice et Bob) veulent échanger un message. Si Alice remet un message à Bob en main propre, elle peut s'assurer que le message n'est pas lu ou modifié par un tiers, et qu'il arrive bien au destinataire prévu. Mais si la communication se fait à distance, il est difficile de savoir si quelqu'un a pu avoir accès au message. En particulier si la transmission est faite par onde radio, il est facile d'intercepter le signal. On imagine donc un adversaire (Charlie) qui a accès au canal de communication. Suivant les cas, on pourra supposer qu'il peut seulement lire les messages qui transitent (adversaire passif) ou qu'il peut les modifier ou fabriquer de nouveaux messages (adversaire actif).



La cryptographie fournit tout un arsenal d'algorithmes pour garantir différentes propriétés de sécurité, mais les principales propriétés qu'on essaye d'obtenir sont :

Confidentialité : Charlie ne doit pas pouvoir lire le message d'Alice.

Intégrité : si Charlie modifie le message, Bob doit pouvoir le détecter.

Authenticité : si Charlie fabrique un message, Bob va détecter que le message ne vient pas d'Alice.

On distingue deux grandes familles d'algorithmes cryptographiques. Dans un système à clef secrète, Alice et Bob partagent la même clef secrète K. Cette clef est utilisée dans un algorithme de chiffrement, et dans l'algorithme de déchiffrement correspondant. Si Alice et Bob protègent correctement leur clef, Charlie n'y a pas accès et ne peut pas déchiffrer le message. Dans un système à clef privée, chaque paire d'utilisateurs qui veulent communiquer entre eux doit partager une clef secrète. Au contraire, dans un système à clef publique, Bob possède deux clefs différentes : une clef publique e et une clef privée d. La clef publique est utilisée pour chiffrer le message, mais c'est la clef privée qui est nécessaire pour déchiffrer. Ainsi, la clef publique peut être largement diffusée, et n'importe qui peut envoyer un message à Bob en le chiffrant avec sa clef publique. La cryptographie à clef publique permet à Alice et Bob d'échanger des messages de façon secrète sans avoir échangé de clef secrète au préalable !

Cryptographie symétrique. Les systèmes à clef privée sont aussi appelés cryptographie symétrique car les situations d'Alice et de Bob sont symétriques : ils connaissant la même clef. Les principales constructions utilisées en cryptographie symétrique sont :

Les schémas de chiffrement par bloc : un schéma de chiffrement par bloc permet de chiffrer et déchiffrer un bloc de message en utilisant une clef. Pour chiffrer des messages de taille arbitraire, on utilise un mode opératoire qui fait plusieurs appels au chiffrement par bloc. Beaucoup de constructions cryptographiques utilisent un schéma de chiffrement par bloc, et c'est souvent l'AES qui est utilisé. L'AES [DR98, Fips197] a été sélectionné en 2001 après une compétition internationale qui a mobilisé toute la communauté cryptographique.

Les schémas de chiffrement par flot : un schéma de chiffrement par flot produit une suite aléatoire à partir d'une clef. Pour chiffrer un message, on fait la somme de cette suite aléatoire et du message, et on peut déchiffrer en inversant cette opération. Les schémas de chiffrement par flot sont généralement plus rapides que les schémas de chiffrement par bloc, et plus adaptés à une implémentation matérielle. Les schémas de chiffrement par flot sont moins utilisés que les schémas de chiffrement par bloc, et moins bien étudiés. Le projet eSTREAM qui s'est achevé en 2008 a organisé une compétition pour choisir de nouveaux algorithmes.

Les fonctions de hachage : une fonction de hachage est une fonction publique qui doit se comporter comme une fonction aléatoire. En particulier il doit être difficile de trouver des collisions ou d'inverser la fonction. La définition d'une fonction de hachage ne fait pas intervenir de secret, mais les fonctions de hachages sont utilisées pour construire de nombreux protocoles cryptographiques. On les classe dans la cryptographie symétrique car les constructions de fonctions de hachage sont similaire aux constructions de schémas de chiffrement par bloc ou par flot. Les fonctions les plus utilisées actuellement sont MD5, SHA-1 et SHA-2. Une compétition est en cours pour sélectionner un nouvel algorithme (qui sera SHA-3), et la recherche autour des fonctions de hachage est actuellement très active.

Les codes d'authentification de message ou MAC : un MAC permet d'authentifier l'auteur d'un message grâce à une clef secrète. Les MAC les plus utilisés sont construits à partir d'une fonction de hachage (HMAC), d'une fonction de hachage universelle (UMAC), ou d'un schéma de chiffrement par bloc (CBC-MAC).

Cryptographie asymétrique. Les systèmes à clef publique sont aussi appelés cryptographie asymétrique car les situation d'Alice et de Bob ne sont pas symétriques : Alice ne connaît pas la clef privée de Bob. La sécurité de ces algorithmes est basée sur un problème mathématique difficile, comme la factorisation des entiers, le calcul du logarithme discret dans un groupe, ou la réduction des réseaux. Les principales constructions utilisées en cryptographie asymétrique sont :

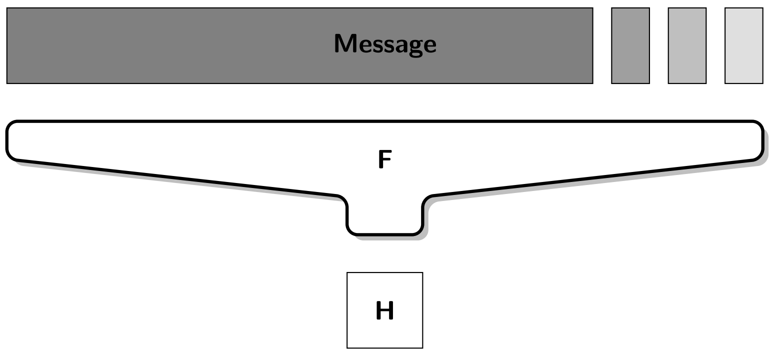
Les systèmes de chiffrement : un système de chiffrement asymétrique permet de chiffrer un message avec une clef publique, et de déchiffrer avec la clef privée associée. Les protocoles de chiffrement les plus utilisés sont basé sur la difficulté de la factorisation, comme le système RSA [RSA78], ou sur sur le problème du logarithme discret, comme le système ElGamal [ElG85] où le système de Schnorr [Sch89].

Les signatures : une signature permet d'identifier l'auteur d'un message. On utilise une clef privée pour calculer la signature, et une clef publique pour vérifier la signature. Les systèmes les plus utilisés sont basés sur les mêmes hypothèses que les systèmes de chiffrement : RSA et DSA (qui est une variante de ElGamal).

Les protocoles d'échange de clef : un protocole d'échange de clef permet à deux personnes de se mettre d'accord sur une valeur sans qu'un attaquant puisse connaître cette valeur en observant la communication. Cette valeur peut ensuite être utilisée comme clef secrète pour un système de chiffrement symétrique. Le protocole d'échange de clef le plus connu est celui de Diffie-Hellman [DH76], basé sur le logarithme discret.

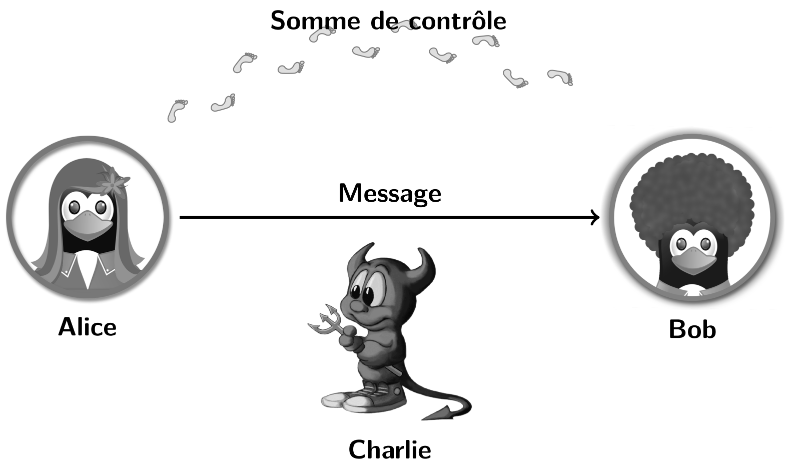
Dans cette thèse, je me suis intéressé aux fonctions de hachage. C'est une primitive qui est construite en utilisant les techniques de la cryptographie symétrique, mais elle sert aussi bien pour des schémas symétriques que pour des schémas asymétriques.

1.2 Les fonctions de hachage



Une fonction de hachage prend en entrée un message et calcule une empreinte de ce message. Le message est vu comme une suite de bits de longueur arbitraire, et l'empreinte, ou haché, est une suite de bits de taille fixe, par exemple 256 bits. La fonction de hachage va "comprimer" le message, et produire une empreinte qui dépend de tous les bits du message, et peut servir à le "représenter".

Les fonctions de hachage ont de nombreuses utilisations en informatique, basées sur l'idée que l'empreinte peut servir à identifier le message sous une forme plus compacte. Cela permet de construire une structure de données appelée table de hachage où les documents sont identifiés par leur empreinte, et qui permet de tester très efficacement si un document est déjà présent. On peut aussi utiliser une fonction de hachage comme somme de contrôle quand on transfère un document entre deux machines : s'il y a eu une erreur lors de la transmission, cela va probablement modifier la somme de contrôle, et on peut détecter l'erreur. Dans ces deux exemples, on peut se contenter d'une fonction de hachage très simple, par exemple la somme de tous les octets du message.



1.2.1 Fonctions de hachage cryptographiques

Dans ce document, on considère les fonctions de hachage d'un point de vue cryptographique. La fonction ne doit pas seulement avoir un bon comportement quand les entrées sont aléatoires, mais on va considérer des notions mettant en jeu un adversaire. Par exemple si on utilise une fonction de hachage comme somme de contrôle pour vérifier l'intégrité d'un document après un transfert, on ne va pas seulement s'intéresser aux erreurs de transmission aléatoires, mais on envisage le cas d'un adversaire qui va introduire des erreurs délibérément pendant la transmission. On peut supposer par contre que la somme de contrôle est envoyée par un moyen fiable, par exemple par téléphone.

Dans ce scénario, une fonction de hachage simple, telle que la somme de tous les octets du fichiers, ne protège pas le document. Charlie peut facilement créer un document quelconque dont la somme de contrôle soit la même que celle du document original, simplement en ajustant la valeur du dernier octet. La notion de sécurité dont on a besoin est ce qu'on appelle la sécurité contre les deuxièmes préimages : Charlie ne doit pas pouvoir construire un deuxième document avec la même empreinte que le document original.

Une fonction de hachage qui résiste à des attaques de ce type est appelée une fonction de hachage cryptographique. Dans la suite de ce document on ne parlera que de fonctions de hachage cryptographiques, et on se permettra d'omettre ce qualificatif.

Une fonction publique sans structure exploitable. La sécurité maximale qu'on peut espérer d'une fonction de hachage est de se comporter comme une fonction aléatoire. Une fonction de hachage est alors une fonction publique sans structure exploitable : on ne doit pas pouvoir construire de messages avec des propriétés particulières plus efficacement que sur une fonction aléatoire. Les propriétés les plus intéressantes pour l'étude des fonctions de hachage sont la recherche de collisions, la recherche d'antécédents (ou préimages), et la recherche de deuxièmes antécédents.

1.2.2 Utilisations

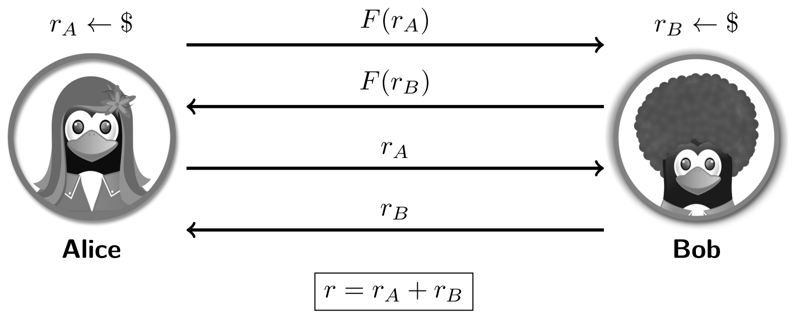
Les fonctions de hachage permettent de réaliser de nombreuses fonctionnalités, et sont très utilisées en cryptographie. C'est une primitive très polyvalente, et les fonctions actuellement utilisées sont relativement rapides, donc beaucoup de systèmes cryptographiques sont basés sur des fonctions de hachage. Ainsi les fonctions de hachage ont acquis une réputation de "couteau suisse" de la cryptographie.

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

Signatures Hash-and-Sign. Les schémas de signature comme RSA ou ElGamal permettent d'authentifier un message, mais ils demandent des calculs complexes et coûteux. En pratique, au lieu d'appliquer un schéma de signature directement à un long message, on applique la signature à un haché du message. Ainsi, l'opération de signature est faite sur un identifiant de petite taille et sera moins coûteuse.

Si on veut que le signataire ne puisse pas répudier ses signatures, il faut que la fonction de hachage soit résistante en collision. Si on veut qu'un adversaire ne puisse pas forger des signatures, il faut que la fonction soit résistante aux secondes préimages.

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



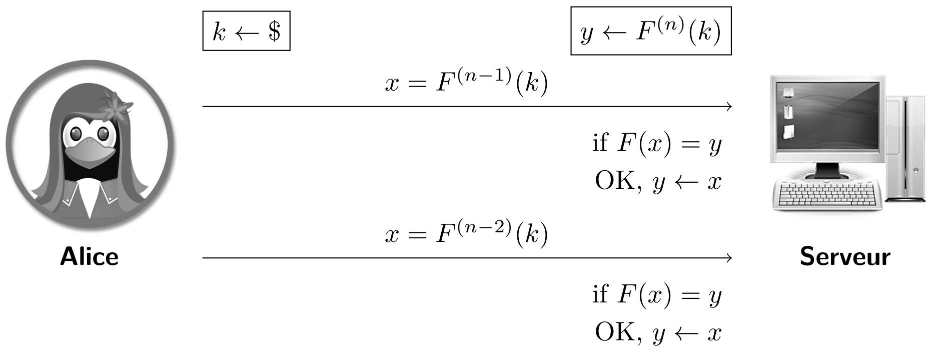
Un schéma d'engagement permet aussi d'horodater un document, pour garantir qu'il existait à une certaine date. Pour construire un horodatage, ou time-stamp, on enregistre le haché d'un message auprès d'une autorité, et cette autorité peut ensuite certifier la date à laquelle le haché, et donc le message,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

Pour ce type d'utilisation, on n'a pas forcément besoin de résistance en collision (si on sait fabriquer une collision, il est vrai que les deux messages sont connus au moment où le haché est publié), mais on a besoin d'une notion plus forte que la résistance en préimage. En effet, il est possible de s'engager sur une valeur particulière, pour laquelle il sera plus facile de modifier le message ultérieurement. Ceci correspond à la notion de Chosen Target Forced Prefix preimage resistance définie dans [KK06].

Utilisation comme fonction à sens unique. Une autre utilisation importante des fonctions de hachage est leur usage en tant que fonction à sens unique.

Stockage de mot de passe. Les fonctions de hachage sont utilisées pour éviter de stocker des mots de passe en clair. Ainsi, quand on se connecte sur un ordinateur, la machine calcule un haché du mot de passe, et le compare au haché préalablement connu. Ceci permet d'éviter de stocker le mot de passe en clair, et si la machine est compromise, l'attaquant ne pourra pas retrouver les mots de passe des utilisateurs.

One-Time-Password. Une utilisation astucieuse d'une fonction de hachage permet de mettre en place un système de mot de passe à usage unique. L'utilisateur choisit une valeur aléatoire k, et itère la fonction de hachage plusieurs fois sur cette valeur. Le résultat du calcul, $F^{(n)}(k)$ est stocké sur le serveur. À chaque fois que l'utilisateur se connecte au système, il doit fournir une préimage de la valeur connue par le serveur. L'utilisateur légitime peut facilement calculer $F^{(n-1)}(k)$ à partir de k, mais un attaquant ne peut pas le calculer à partir de $F^{(n)}(k)$. À chaque fois que l'utilisateur s'identifie correctement, le système conserve la valeur de la préimage, et la prochaine identification se fera avec une préimage de cette nouvelle valeur. Ainsi, chaque identification demande un mot de passe différent. Ceci permet de limiter les risques encourus si quelqu'un risque d'espionner le mot de passe. Un système construit sur ce principe est disponible pour certains systèmes d'exploitation [RFC2289].



Forward security. La notion de forward security garantit que si les clefs privées à long terme d'un serveur sont corrompues, cela ne permet pas d'attaquer les communications effectuées avant la corruption. On peut construire un système assurant la forward security à partir d'un générateur pseudo-aléatoire dont l'état est transformé de façon non inversible à chaque itération, par exemple avec une fonction de hachage [BY03].

La même idée est aussi utilisée pour se protéger des attaques par canaux auxiliaires, qui permettent de récupérer de l'information sur la clef en utilisant des faiblesses de l'implémentation physique du cryptosystème (comme la consommation de courant ou le rayonnement électromagnétique). Si on remplace régulièrement la clef par son haché, les informations partielles ne suffisent pas à reconstruire un état complet [KJ01].

Utilisation pour masquer la structure de l'entrée. Dans certaines utilisations, la fonction de hachage sert à détruire la structure de l'entrée. On ne sait pas toujours définir exactement quelle propriété est nécessaire dans ce cas, mais on peut prouver que certaines constructions sont sûres si la fonction de hachage est modélisée par un oracle aléatoire.

Extraction d'entropie. Une fonction de hachage permet d'extraire l'entropie d'une source aléatoire non uniforme pour construire une clef uniforme. Par exemple, après un échange de clef Diffie-Hellman, les deux participants possèdent une clef secrète, mais cette clef n'est pas une suite de bits aléatoires, c'est un entier modulo un nombre premier. Pour utiliser cette clef dans un autre algorithme cryptographique, on en calcule le haché pour obtenir une suite de bits aléatoires. Les fonctions de hachage sont aussi utilisées pour produire un aléa de bonne qualité à partir d'un générateur physique ne produisant pas des bits indépendants et uniformément distribués.

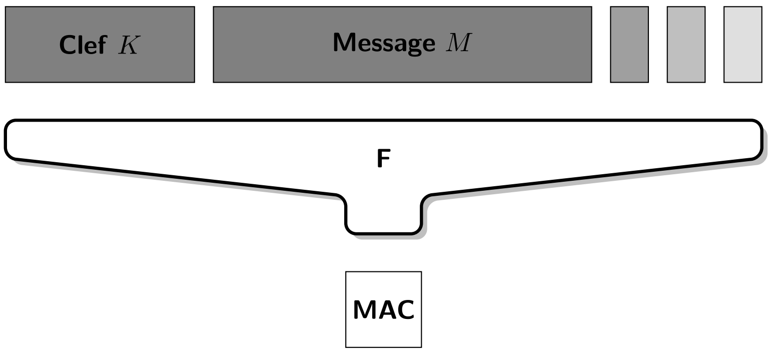
Dérivation de clef. Une fonction de hachage peut aussi servir à dériver plusieurs clefs indépendantes à partir d'une même information secrète, la clef maître. Ceci permet d'utiliser la même clef maître pour un algorithme de chiffrement et un algorithme d'authentification, sans risquer une mauvaise interaction entre les deux protocoles. Ceci peut aussi servir de contre-mesure contre les attaques physiques : si on ne peut pas empêcher une partie de la clef de fuir à chaque utilisation, on peut limiter les risques en générant fréquemment des nouvelles clefs à partir d'une clef maître.

Générateur pseudo-aléatoire. On peut facilement construire un générateur pseudo-aléatoire à partir d'une fonction de hachage. Par exemple, si x est la graine du générateur, on peut utiliser $F(x\|0), F(x\|1), F(x\|2), \ldots$ Cette suite aléatoire peut être utilisée comme suite chiffrante dans un schéma de chiffrement par flot.

Une autre possibilité est de mettre à jour l'état avec $x \leftarrow F(x\|0)$ et de sortir à chaque itération $F(x\|1)$.

Utilisation comme MAC. Un code d'authentification de message, ou MAC, est l'équivalent symétrique d'une signature. Pour authentifier un message M, Alice et Bob partagent une clef K, et ajoutent un identifiant $\mathrm{MAC}_K(M)$ au message. Un MAC doit être difficile à forger, *i.e.* un adversaire ne doit pas pouvoir calculer de MAC sans connaître la clef.

Une façon simple de construire un MAC et de faire rentrer le message et la clef dans une fonction de hachage : $\mathrm{MAC}_K(M) = F(K\|M)$:



Cette construction s'appelle secret-prefix MAC, elle est sûre si la fonction de hachage se comporte bien comme une fonction aléatoire, puisqu'on ne peut pas prévoir la valeur de la fonction en un point en connaissant seulement sa valeur en d'autres points. Cependant, on verra dans Section 1.3.3 que des fonctions très utilisées comme MD5 ou SHA-1 ne peuvent pas être utilisées de cette façon pour construire un MAC. Pour construire un MAC à partir d'une fonction de hachage, on utilise donc des constructions légèrement plus complexes, comme HMAC [BCK96a].

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

Les protocoles par défi/réponse sont très utilisés en pratique. Par exemple, le mode d'authentification CRAM-MD5 [RFC2195], utilisé dans SASL, POP3, IMAP, et SMTP, est un protocole défi/réponse construit avec HMAC-MD5. Des protocoles plus vieux, comme APOP [RFC1939] ou CHAP [RFC1994] utilisent des constructions plus simples en concaténant le mot de passe et le défi, comme dans la construction secret-prefix MAC.

Le modèle de l'oracle aléatoire. Le modèle de l'oracle aléatoire est un modèle de calcul dans lequel les participants ont accès à un oracle qui renvoie une chaîne aléatoire pour toute nouvelle requête, mais renvoie la même valeur si on fait plusieurs fois la même requête. Cet oracle modélise ainsi une fonction aléatoire. Ce modèle est utilisé en théorie de la complexité, où il permet de définir de nouvelles classes de complexité. En cryptographie, il a été popularisé par Bellare et Rogaway [BR93], qui proposent d'utiliser des constructions à base de fonction de hachage pour instancier l'oracle. Aujourd'hui, la plupart des nouvelles constructions de fonctions de hachage ont pour but d'être utilisables en tant qu'oracle aléatoire.

Le modèle de l'oracle aléatoire est une façon d'idéaliser une fonction de hachage. C'est un modèle très utilisé car il permet de prouver la sécurité de nombreuses constructions, comme RSA-OAEP [BR94] ou RSA-PSS [BR96]. Pour une instanciation pratique, l'oracle est remplacé par une fonction de hachage, et la preuve garantit qu'il n'existe pas d'attaque générique sur le schéma : une attaque doit utiliser une propriété de la fonction de hachage.

Ce modèle est aussi très critiqué car c'est une hypothèse extrêmement forte sur la fonction de hachage. De plus, un oracle aléatoire possède une entropie infinie, alors qu'une fonction de hachage doit avoir une faible entropie pour pouvoir être calculée efficacement. Cette différence permet de construire des schémas qui sont sûrs dans le modèle de l'oracle aléatoire, mais qu'on peut attaquer pour n'importe quelle instanciation de l'oracle par une fonction de hachage [MRH04].

1.2.3 Définition

Formellement, une fonction de hachage est une fonction de l'ensemble des suites binaires (de longueur quelconque, non bornée) vers les suites de longueur n :

$$F : \{0,1\}^* \to \{0,1\}^n$$

Pour une utilisation cryptographique, on s'attend à ce qu'une fonction de hachage se comporte comme une fonction choisie aléatoirement parmi toutes les fonctions de $\{0,1\}^*$ vers $\{0,1\}^n$. Il ne doit pas être possible de trouver des relations entre l'entrée et la sortie, ou de trouver des sorties avec des relations particulières autrement que par recherche exhaustive, *i.e.* en calculant la fonction sur suffisamment d'entrées.

Sécurité. Une fonction de hachage idéale doit se comporter comme un oracle aléatoire : le seul moyen d'apprendre de l'information sur le haché d'un message est de le calculer, et le résultat doit être “aléatoire”. Malheureusement, on ne peut pas formaliser correctement cette notion de sécurité, et on considère donc des notions plus faibles, mais plus facile à définir, et qui suffisent pour utiliser la fonction dans certaines constructions. Certaines constructions utilisant une fonction de hachage ont besoin de propriétés de sécurité spécifiques mais les propriétés les plus courantes sont liées à l'utilisation d'une fonction de hachage pour produire des identifiants uniques : la résistance aux attaques en préimage, en seconde préimage, et en collision.

Attaque en antécédent, ou préimage. L'attaque la plus forte est l'attaque en préimage. Dans cette attaque, on donne à l'adversaire un haché, et il doit trouver un message qui donne ce haché. On peut définir deux notions différentes suivant que le haché est tiré aléatoirement parmi l'ensemble d'arrivée, ou qu'on calcule le haché d'un message aléatoire de l'ensemble de départ (suivant une certaine distribution). Dans le second cas, l'antécédent sera en général différent du message initial, puisqu'il n'y a pas unicité. Contre une fonction idéale, la meilleure attaque est la recherche exhaustive, qui a une complexité de 2^n.

Attaque en deuxième antécédent, ou deuxième préimage. Dans une attaque en deuxième préimage, on donne un message à l'adversaire, et il doit en trouver un autre qui ait le même haché. Cette attaque est en général plus facile qu'une attaque en préimage, car on peut réutiliser certaines parties du premier message pour construire le second. Cependant, la meilleure attaque générique reste la recherche exhaustive, qui nécessite un temps de 2^n.

Attaque en collision. L'attaque la plus «simple» est l'attaque en collision. Ici, on demande simplement à l'adversaire de trouver deux messages ayant le même haché. Une attaque en collision est plus facile qu'une attaque en préimage car l'adversaire a le choix des deux messages. Cela permet une attaque générique en $2^{n/2}$ basée sur le paradoxe des anniversaires.

Attaque	Entrées	Sorties	Propriété	Cplx.
Collision	-	M_1, M_2	$F(M_1) = F(M_2)$	$2^{n/2}$
2^e^ Préimage	M_1	M_2	$F(M_1) = F(M_2)$, $M_1 \neq M_2$	2^n
Préimage	$F(M_1)$	M_2	$F(M_1) = F(M_2)$	2^n

Ces notions de sécurité correspondent à l'intuition d'une fonction injective à sens unique. Comme l'ensemble de sortie est fini, toutes ces attaques sont possibles par recherche exhaustive, avec une complexité de 2^n pour les préimages, et $2^{n/2}$ pour les collisions. Quand on utilise une fonction de hachage, on choisira n assez grand pour que ces attaques soient impossible *en pratique*. Les plus gros calculs effectués publiquement sont de l'ordre de 2^{64} opérations, donc on prendra au moins $n = 160$. Ainsi, pour une bonne fonction de hachage, il sera impossible *en pratique* de trouver des collisions ou des préimages. Ceci permet d'utiliser une fonction de hachage comme si elle était injective.

Paradoxe des anniversaires. Le paradoxe des anniversaires est une propriété mathématique qui limite la résistance en collision d'une fonction aléatoire. Si on choisit k éléments dans un ensemble de taille N, on s'attend à avoir deux éléments identiques à partir du moment où $k \approx \sqrt{N}$. Intuitivement, ceci est dû au fait que k éléments permettent de construire k^2 paires, et chaque paire a une probabilité $1/N$ d'être une collision. Une étude rigoureuse montre qu'il faut en moyenne $\sqrt{\pi N/2}$ éléments avant de trouver la première collision [vOW99, Appendix A].

Pour utiliser cette propriété dans une attaque en collision, on peut calculer les hachés de $2^{n/2}$ messages aléatoires, stocker tous les hachés dans une liste, et trier la liste pour trouver une collision. Ceci demande un temps de $2^{n/2}$, mais aussi une mémoire de $2^{n/2}$, ce qui va être le facteur limitant en pratique. On utilise donc des algorithmes de recherche de collision sans mémoire, telle que la méthode rho de Pollard, ou des variantes utilisant des points distingués pour pouvoir paralléliser le calcul [QD89, vOW99].

On peut généraliser cette propriété aux cas où on construit deux ensembles différents de taille k_1 et k_2. Cela définit $k_1 k_2$ paires, donc on a une bonne probabilité d'avoir une valeur commune entre les deux ensembles quand $k_1 k_2 \approx N$.

Problèmes de définition. En cryptographie, on utilise généralement la présence d'une information secrète pour définir la sécurité d'un système. On dit qu'un système est sûr si un adversaire ne peut pas retrouver d'information sur le secret, ou pour une notion encore plus forte, si l'adversaire ne sait pas faire la différence entre un vrai système avec une clef secrète et un faux système qui ne renvoie que des données aléatoires.

Dans le cas d'une fonction de hachage, la fonction est *publique*, et on ne peut donc pas utiliser ce type de définition : il est facile de reconnaître une fonction publique ! De plus, si la fonction est publique, il est possible de faire un précalcul, et de construire une attaque utilisant le résultat de ce précalcul. En particulier, une fonction de hachage ne peut pas être résistante aux collisions : il existe des collisions, et il existe donc une attaque qui “connaît” une paire de messages qui collisionne, et qui se contente de l’imprimer.

La façon la plus simple d’éviter ce problème est de considérer une *famille* de fonctions de hachage (par exemple, en modifiant les constantes utilisées dans la description de la fonction), et d’imaginer que la fonction précise qui sera utilisée est choisie au moment de l’exécution du protocole. Ainsi on peut faire un précalcul qui dépend de la famille de fonctions, mais pas un précalcul qui dépend de la fonction exacte qui sera utilisée. Cela permet de résoudre le problème de définition, mais ça ne correspond pas à la façon dont les fonctions de hachage sont réellement utilisées : tout le monde utilise la même version de SHA-1, par exemple.

Une autre façon de résoudre le problème a été proposée par Rogaway dans [Rog06]. L’idée est d’utiliser une preuve par réduction : plutôt que de prouver qu’un protocole est sûr s’il est utilisé avec une fonction sans collision, on prouve qu’une attaque contre le protocole fourni explicitement une collision. Ainsi, tant qu’on ne connais pas de collision, le protocole est sûr. Cette façon d’aborder le problème correspond à la façon dont les fonctions de hachage sont utilisées en pratique, mais elle ne s’applique pas à toutes les utilisations des fonctions de hachage.

De même, quand on demande que les sorties d’une fonction de hachage soient aléatoires, cela n’a pas vraiment de sens quand il s’agit d’une fonction *publique* et *fixée*. On peut tenter de formaliser cette idée d’aléatoire en utilisant la notion de distingueur : si on trouve une propriété particulière satisfaite par la fonction de hachage, mais qui n’est pas satisfaite par une fonction aléatoire (ou avec une probabilité différente), cela indique un problème dans la construction. Il faut cependant écarter les distingueurs triviaux qui utilisent simplement le fait que la fonction soit publique.

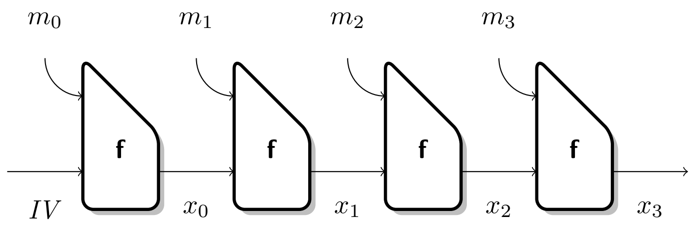
Malheureusement, on ne connaît pas de définition satisfaisante de la sécurité d’une fonction de hachage qui capture toutes les propriétés dont on peut avoir besoin. On doit donc faire preuve de bon sens pour décider si une attaque est valide ou non.

Relations entre les notions de sécurité. De façon générale, une attaque en préimage peut être utilisée en deuxième préimage, en ignorant simplement le premier message, et en n’utilisant que son haché. De même, une attaque en seconde préimage peut être utilisée pour trouver des collisions, en choisissant arbitrairement le premier message. Ceci montre que si une fonction résiste aux attaques en collision, elle résistera aussi aux attaques en préimage et aux attaques en deuxième préimage. Cependant, ces réductions ont des limitations :

– L’utilisation d’une attaque en préimage pour construire des deuxièmes préimages ne marche pas en toute généralité car une attaque en préimage peut renvoyer toujours le même antécédent. Il peut aussi y avoir des problèmes suivant la façon exacte dont on définit la notion de préimage (pour plus de détails, voir [RS04]).
– Si on connaît une attaque en deuxième préimage, on peut l’utiliser pour construire des collisions, mais cela ne donnera une attaque en collision que si la complexité est inférieure à $2^{n/2}$. Le niveau de sécurité attendu n’est pas le même pour une attaque en préimage et une attaque en collision.

1.3 Mode opératoire

Plutôt que de construire directement une fonction qui compresse une entrée de taille arbitraire, on construit une fonction de hachage à partir d’une fonction de compression. Une fonction de compression accepte une entrée de taille fixe — p bits d’état et m bits de message — et la compresse vers p bits. Réciproquement, toute fonction de hachage qui utilise une mémoire bornée et traite le message en flux peut être vue comme l'itération d'une fonction de compression. L'idée d'utiliser une construction itérative pour construire une fonction de hachage a été initialement proposée par Rabin en 1978 [Rab78].

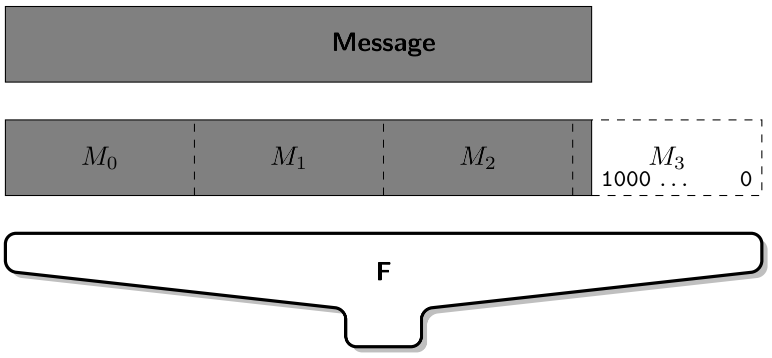


1.3.1 Construction de Merkle-Damgård

La façon la plus simple de construire une fonction de hachage itérée est d'utiliser le dernier état calculé, x_{k-1}, comme valeur de sortie. Dans ce cas la taille de l'état interne p, est la même que la taille de sortie n. Cette technique a été rendue populaire pas Merkle et Damgård qui ont indépendemment proposé une preuve de sécurité de cette construction en supposant que la fonction de compression résiste aux collisions [Mer89, Dam89]. Depuis, cette méthode est connue sous le nom de construction de Merkle-Damgård.

On a donc un état interne x de $p = n$ bits, qu'on initialise à une valeur constante IV. Puis on itère la fonction de compression pour mettre à jour l'état : $x_i = f(x_{i-1}, m_i)$. Enfin, la valeur de sortie sera juste x_{k-1} la dernière valeur de l'état interne.

Préparation du message. Cette construction permet facilement de hacher des messages de taille arbitrairement longue, mais elle ne gère que les messages de taille multiple de m. On doit donc d'abord préparer le message et l'encoder avec des blocs de m bits. Il est important que cet encodage soit injectif, car une collision dans l'encodage donnerait une collision dans la fonction de hachage. L'encodage le plus utilisé consiste à ajouter des bits de remplissage (ou padding) à la fin : on ajoute un bit 1, suivi d'un nombre variable de bits 0 (entre 0 et $m-1$), de façon à atteindre une taille multiple de m. Si le message de départ fait km bits, le message encodé utilisera $k+1$ blocs, et le dernier bloc contiendra uniquement les bits de remplissage. Sinon, les bits de remplissage sont inclus dans le dernier bloc. Le message M est ensuite découpé en k blocs de m bits $m_0, \ldots m_{k-1}$.



Pour plus de sécurité, certaines constructions ajoutent la taille du message comme entrée dans le dernier bloc. Cette façon de traiter le message est connue sous le nom de MD-Strengthening.

1.3.2 Réduction de sécurité

La sécurité de la fonction de hachage itérée est liée à la sécurité de la fonction de compression. On utilise les notions de sécurité suivantes :

Collision : une collision dans la fonction de compression est la donnée d'une paire d'entrées $(x, m), (x', m')$, avec $(x, m) \neq (x', m')$ et $f(x, m) = f(x', m')$.

Deuxième antécédent, ou deuxième préimage : étant donné une entrée (x, m), une deuxième préimage pour la fonction de compression est une autre entrée (x', m') avec $(x, m) \neq (x', m')$ et $f(x, m) = f(x', m')$.

Antécédent ou préimage : étant donné une sortie h, une préimage pour la fonction de compression est une entrée (x', m') avec $f(x, m) = h$.

Pour définir ces notions, on autorise l'attaquant à choisir la valeur de chaînage. Ainsi une collision dans la fonction de compression ne suffit pas à construire une collision dans la fonction de hachage. C'est pourquoi ces propriétés sont aussi appelées pseudo-collision et pseudo-préimage, ou collision libre et préimage libre. On utilise aussi parfois la notion de collision semi-libre pour signifier que la valeur de chaînage est choisie mais qu'elle est la même pour les deux entrées.

Collisions. L'itération de Merkle-Damgård avec MD-strengthening permet une preuve de sécurité très simple : si la fonction de compression résiste aux collisions, alors la fonction de hachage résiste aux collisions.

Démonstration. Soit $M \neq M'$ deux messages de longueurs respectives k et k' blocs (après le padding) tels que $F(M) = F(M')$.

Si $k \neq k'$, alors le dernier bloc (avec le padding) de M et M' est différent, puisqu'il contient la taille du message. On a une collision $f(x_{k-2}, m_{k-1}) = f(x'_{k'-2}, m'_{k'-1})$.

Sinon, on va chercher le dernier bloc pour lequel $x_i = x'_i$. Soit $k_1 = max\{i : m_i \neq m'_i\}$ (cet ensemble est non-vide car les messages sont différents et de même longueur) et $k_2 = min\{i \geq k_1 : x_i = x'_i\}$ (cet ensemble est non-vide car $x_{k-1} = x'_{k-1}$). Si $k_1 = k_2$, alors $F(x_{k_1-1}, m_{k_1}) = x_{k_1} = x'_{k_1} = F(x'_{k_1-1}, m'_{k_1})$ fournit une collision dans f car $m_{k_1} \neq m'_{k_1}$. Sinon, on a $f(x_{k_2-1}, m_{k_2}) = x_{k_2} = x'_{k_2} = f(x'_{k_2-1}, m'_{k_2})$, d'où une collision car $x_{k_2-1} \neq x'_{k_2-1}$.

□

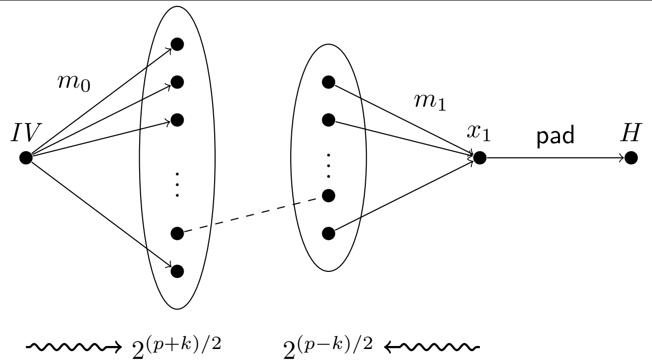
Ainsi, pour construire une fonction de hachage résistante aux collisions, il est *suffisant* de construire une fonction de compression résistante aux collisions. Il est à noter que ce n'est pas une condition *nécessaire*, car une attaque en collision sur la fonction de compression peut choisir la variable de chaînage x.

Une attaque en collision sur la fonction de compression invalide la preuve de sécurité. De plus, c'est souvent le signe qu'il y a un problème avec la construction. Par exemple, la première attaque sur MD5 était une attaque en collision sur la fonction de compression [dBB93]. Cette attaque n'affectait pas la fonction de hachage itérée, mais l'idée principale de cette attaque est un élément important qui a permis de construire une vraie attaque en collision [WY05].

Préimages. On peut aussi facilement construire une preuve de sécurité pour les préimages : si la fonction de compression résiste aux préimages, alors la fonction de hachage résiste aux préimages.

Démonstration. Étant donné une préimage pour la fonction de hachage, il suffit de prendre les entrées du dernier appel à la fonction de compression pour avoir une préimage pour la fonction de compression. □

Figure 1.1 Attaque en deuxième préimages par meet-in-the-middle



On verra plus loin que ce résultat est faux pour les deuxièmes préimages : même si la fonction de compression est parfaite, on peut calculer des deuxièmes préimages de messages longs en moins de 2^n.

Dans le cas des préimages, on a aussi une forme de réciproque : si on a une attaque en préimage ou deuxième préimage sur la fonction de compression avec complexité 2^k, on peut construire une attaque en deuxième préimage contre la fonction de hachage avec complexité $2^{(p+k)/2}$, si la valeur de chaînage est de taille p. L'attaque utilise la technique de meet-in-the-middle [LM92] illustrée par la figure 1.1 : on calcule d'un côté $2^{(p-k)/2}$ préimages d'une valeur intermédiaire, et de l'autre $2^{(p+k)/2}$ valeurs intermédiaires en partant de la valeur initiale. Ces deux ensembles définissent 2^p paires, et on a une bonne probabilité d'avoir une valeur qui apparaît dans les deux ensembles. En utilisant les messages qui ont mené à cette valeur, on obtient une deuxième préimage. Dans le cas où l'état interne est de la même taille que la sortie, *i.e.* $p = n$, ceci donne une attaque avec complexité $2^{(n+k)/2}$, et on a $(n+k)/2 < n$ si $k < n$.

En général, cette technique ne s'applique pas pour une attaque en préimage car il faut gérer le MD-strengthening qui ajoute des bits de remplissage et la taille du message à la fin.

Fonctions éponge. La sécurité de la fonction de hachage peut se réduire à la sécurité de la fonction de compression, mais cela ne signifie pas qu'une attaque sur la fonction de compression donne une attaque sur la fonction de hachage. En particulier, si l'état interne p est suffisamment grand par rapport à la taille de sortie n, on peut avoir une fonction de hachage résistante aux attaques en collision ou en préimage même s'il est facile de trouver des collisions ou des préimage quand on contrôle la variable de chaînage.

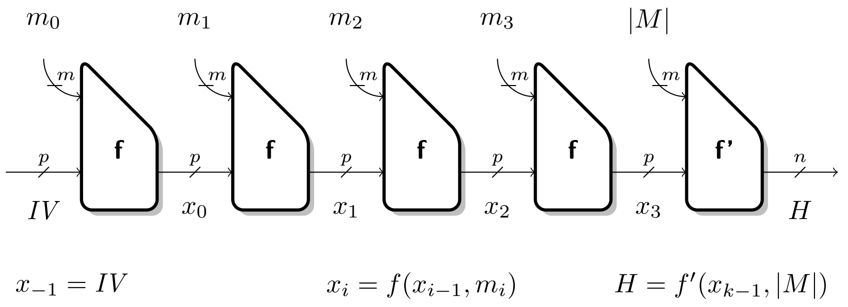
Cette propriété est utilisée dans la construction éponge [BDPA08], qui permet de construire une fonction de hachage à partir d'une permutation fixe. Dans cette construction, la sécurité repose sur le fait que la permutation se comporte comme une permutation aléatoire.

1.3.3 Attaque par extension

Le principal problème de la construction Merkle-Damgård est l'attaque par extension. Comme la sortie de la fonction de hachage contient tout l'état interne, on peut l'utiliser pour continuer le calcul et obtenir le haché d'un suffixe de M à partir de $F(M)$ seulement (*i.e.* sans connaître M). Ceci constitue une attaque contre la construction *secret-prefix* MAC et montre que l'itération de Merkle-Damgård n'est pas une bonne construction de fonction aléatoire.

Cette attaque s'applique à la plupart des fonctions utilisées actuellement comme MD4, MD5, SHA-1 où SHA-2. Elle ne menace pas la résistance en collision ou en préimage, et n'a pas d'impact

Figure 1.2 Fonction de hachage itérée avec fonction de finalisation



sur la sécurité de la plupart des applications utilisant ces fonctions de hachage. Cependant cela pose un vrai problème pour les MAC, car la construction la plus naturelle n'est pas sûre. HMAC a donc été développé pour construire des MAC à partir de ces fonctions de hachage.

Pour éviter cette attaque, on ajoute une fonction de finalisation à la fin du calcul. Si la fonction de finalisation est non-inversible, on ne peut pas retrouver l'état interne à partir du haché, et cela évite l'attaque par extension. On peut prouver que certaines fonctions de finalisation donnent une fonction de hachage qui résiste aux attaques génériques jusqu'à une sécurité en $2^{p/2}$ [CDMP05].

La fonction de finalisation prend p bits d'état et éventuellement des bits de message, et produit une sortie de n bits. Ceci permet facilement d'avoir $p > n$. La fonction de finalisation est souvent construite à partir de la fonction de compression. La figure 1.2 décrit une fonction de hachage construite selon le principe de Merkle-Damgård avec une fonction de finalisation. Dans cet exemple, la fonction de finalisation prend la taille du message en entrée, mais pas de bits de message.

1.3.4 Attaques génériques

Au cours des dernières années, on a découvert des propriétés inattendues des fonctions construites selon le principe de Merkle-Damgård. La preuve du mode Merkle-Damgård montre que c'est une bonne méthode pour construire une fonction résistante aux collisions, mais elle ne dit pas comment se comporte ce type de fonction face à d'autres notions de sécurité.

En effet, les fonctions de hachage itérées sont sensibles à certaines attaques génériques qui ne dépendent que de la taille de l'état interne, p. Dans la construction Merkle-Damgård, on a $p = n$, et cela donne des attaques plus efficaces que les attaques génériques sur une fonction idéale. À cause de ces problèmes, il est préférable d'utiliser un état interne plus gros que la taille de la sortie, avec $p \geq 2n$. L'itération Merkle-Damgård avec un état double est appelée chop-MD [CDMP05] ou wide-pipe [Luc05].

Une autre solution pour éviter certaines de ces attaques est d'ajouter un compteur dans chaque appel à la fonction de compression comme dans la construction HAIFA [BD07]. La table 1.1 montre la complexité de ces attaques génériques en fonction de mode utilisé.

Collisions internes. Plusieurs attaques sont basées sur la notion de collision interne. Une collision interne est une collision sur l'état interne x utilisé pour itérer la fonction de compression. Les collisions internes sont à opposer aux collisions externes qui ont lieu dans la fonction de finalisation avec $x_k \neq x'_{k'}$, mais $H = H'$. Quand on a une collision interne entre deux messages de même longueur M et M', on peut ajouter un suffixe quelconque N aux deux messages, et cela fournit une nouvelle collision $F(M\|N) = F(M'\|N)$.

Figure 1.3 Une 8-collision. On peut construire 8 messages qui ont tous le même haché H : $m_0\|m_1\|m_2$, $m_0\|m_1\|m'_2$, $m_0\|m'_1\|m_2$, $m_0\|m'_1\|m'_2$, $m'_0\|m_1\|m_2$, $m'_0\|m_1\|m'_2$, $m'_0\|m'_1\|m_2$, et $m'_0\|m'_1\|m'_2$.

$$\begin{tikzcd} IV \arrow[r, bend left, "m_0"] \arrow[r, bend right, "m'_0"'] & x_0 \arrow[r, bend left, "m_1"] \arrow[r, bend right, "m'_1"'] & x_1 \arrow[r, bend left, "m_2"] \arrow[r, bend right, "m'_2"'] & x_2 \arrow[r, "\mathsf{pad}"] & H \end{tikzcd}$$

Cette propriété permet de distinguer une fonction de hachage avec $2^{p/2}$ appels à la fonction de compression.

Attaque contre les MACs. Preneel et van Oorschot ont montré une attaque sur les MAC itérés basée sur des collisions internes [PvO95]. L'attaque s'applique aux MAC construits à partir d'une fonction de hachage, comme le secret-prefix MAC ou HMAC, et montre que les fonctions de hachage ne se comportent pas de façon idéale. En effet, dans un MAC basé sur une fonction de hachage, l'attaquant ne connaît pas l'état de la fonction puisqu'il dépend de la clef, mais il peut détecter les collisions dans l'état à travers les collisions dans le MAC.

L'attaquant commence par collecter $2^{p/2}$ MAC de messages de la même longueur et cherche les collisions dans cet ensemble de MAC. Le calcul de ces MAC passe par $2^{p/2}$ états internes, donc on s'attend à avoir une collision interne. Si M, M' est une collision dans le MAC, on peut tester si c'est une collision interne en demandant les MACs de $M\|N$ et $M'\|N$ pour un suffixe aléatoire N : si la collision est bien dans l'état interne, le fait de rajouter un bloc conserve la collision. Une fois qu'une collision interne M, M' est identifiée, on demande le MAC de $M\|N'$ pour un nouveau suffixe N', et on sait qu'on aura $\mathrm{MAC}(M'\|N') = \mathrm{MAC}(M\|N')$.

Ceci donne donc une attaque en forge existentielle avec complexité $2^{p/2}$.

Multi-collisions. Une r-multi-collision est un ensemble de r messages qui ont tous le même haché. Pour construire une r-multi-collision dans une fonction aléatoire de taille de sortie n bits, il faut évaluer la fonction $2^{n\cdot(r-1)/r}$ fois, ce qui tend très vite vers 2^n quand r est grand.

Dans une fonction de hachage itérée, on peut construire des multi-collisions plus efficacement en utilisant des collisions internes. On commence par chercher une paire de messages m_0, m'_0 qui donne une collision interne en partant de l'IV. Ensuite on cherche une paire de messages m_1, m'_1 qui donne une collision interne en partant de l'état x_0 après avoir traité m_0. Si on répète cette opération k fois, on obtient k collisions internes, et on peut construire 2^k messages en choisissant l'un ou l'autre des messages à chaque étape. Ceci est illustré par la figure 1.3.

Cette attaque décrite par Joux [Jou04] permet de construire une 2^k-collision en temps $k \cdot 2^{p/2}$.

Seconde préimage pour des messages longs. Étant donné un long message de 2^k blocs, on peut construire une préimage avec 2^{p-k} évaluations de la fonction de compression. En effet, si on arrive à atteindre un des états intermédiaires $x_0, \dots x_{2^k-1}$, on peut utiliser la fin du message original pour atteindre la même valeur de haché. Le fait d'utiliser la longueur du message à la fin du calcul pourrait empêcher ces attaques, mais Dean [Dea99], puis Kelsey et Schneier [KS05] ont montré comment construire des messages élastiques qui permettent d'ajuster la longueur du deuxième message.

Cette attaque n'a pas beaucoup de conséquences pratiques car le gain n'est significatif que pour des messages extrêmement longs.

Figure 1.4 Attaque par herding

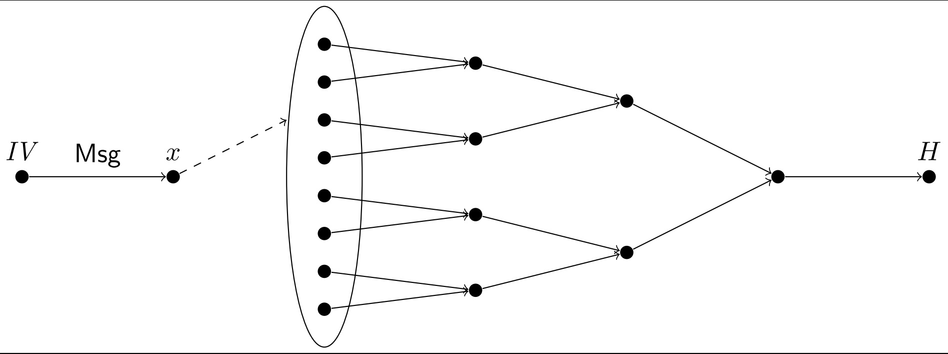


Table 1.1 Attaques génériques contre les modes basés sur Merkle-Damgård

	Fonction idéale	WP	Haifa	NP	MD
Attaque par extension	-	-	-	-	1
Collisions internes	-	2^n	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$
Attaques contre les MAC (forge existentielle)	2^n	2^n	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$
Multi-collisions	2^n	2^n	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$
Engagement (attaque par herding)	2^n	2^n	$2^{2n/3}$	$2^{2n/3}$	$2^{2n/3}$
Seconde préimage messages longs	2^n	2^n	2^n	2^{n-k}	2^{n-k}

WP : Merkle-Damgård wide-pipe avec fonction de finalisation ($p = 2n$)
Haifa : Merkle-Damgård avec fonction de finalisation et compteur ($p = n$)
NP : Merkle-Damgård (narrow-pipe) avec fonction de finalisation ($p = n$)
MD : Merkle-Damgård (narrow-pipe) sans fonction de finalisation ($p = n$)

L'attaque par herding. Kelsey et Kohno [KK06] ont construit une attaque contre les protocoles d'engagement. L'idée est de construire une valeur de haché spéciale, qui sera plus facile à atteindre que les autres. Cette attaque ressemble à l'attaque en seconde préimage contre les messages longs, mais elle fonctionne avec des messages courts.

Dans une première étape, on commence par choisir un ensemble de variables de chaînage, et on cherche des messages qui donnent des collisions entre ces valeurs. Si on commence avec 2^k valeurs de chaînage, on peut trouver des messages qui font collisionner ces valeurs vers 2^{k-1} états en temps $2^{(p+k)/2}$. On répète ensuite pour construire un nouvel ensemble avec 2^{k-2} états, et on itère jusqu'à atteindre une valeur unique. C'est cette valeur qu'on va utiliser pour l'engagement. La complexité totale de cette étape est de l'ordre de $2^{(p+k)/2}$

Dans une deuxième étape, on veut ouvrir l'engagement vers un message qu'on a choisi après la phase d'engagement. Pour cela, on commence par hacher le message donné, puis on hache un bloc aléatoire jusqu'à tomber sur une des valeurs dans la structure. Cette étape se fait en temps 2^{p-k}, car il y a 2^k valeurs dans la structure.

La valeur optimale pour cette attaque est de choisir $k = p/3$, ce qui donne une complexité de $2^{2n/3}$ pour chaque phase de l'attaque. L'attaque est décrite par la figure 1.4.

1.4 Description de la famille de MD4

Afin de donner une idée plus précise de la construction des fonctions de hachage, nous allons décrire brièvement la famille de MD4. MD4 est une fonction de hachage proposée par Ron Rivest en 1990 [Riv90]. C'était une des premières propositions de fonction de hachage, et de nombreuses fonctions ont ensuite été construites en suivant les mêmes idées : MD5, SHA-1, SHA-2, RIPEMD, HAS-160, HAVAL... La plupart de fonctions de hachage utilisées aujourd'hui font partie de cette famille, et même MD4 est encore utilisé dans certains contextes. Ainsi, il est important de comprendre le fonctionnement de ces fonctions, et d'étudier leur sécurité.

Les fonctions de famille de MD4 utilisent l'itération Merkle-Damgård, avec un état interne de la même taille que l'état de sortie, et sans fonction de finalisation. Elles sont donc sensibles aux attaques génériques décrites précédemment. La fonction de compression utilise la construction de Davies-Meyer : elle est construite à partir d'un algorithme de chiffrement par bloc, dans lequel le message est utilisé comme clef : $h(x, m) = E_m(x) \boxplus x$. Les fonctions de la famille de MD4 utilisent une expansion de message et un schéma de Feistel déséquilibré ; la fonction de compression utilise un petit nombre de registres, qui sont mis à jour un par un en ajoutant à la valeur précédente une fonction des autres registres et une sous-clef. La figure 1.5 représente la structure générale de ces fonctions.

La fonction de tour du schéma de Feistel est basée sur des opérations simples et efficaces sur la plupart des processeurs :
– des rotations (notées $\lll$) ;
– des additions $\mod 2^{32}$ (notées $\boxplus$) ;
– des fonctions booléennes bit-à-bit, parmi les suivantes :

$$\begin{aligned}
\mathsf{IF}(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z)\\
\mathsf{MAJ}(x,y,z) &= (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)\\
\mathsf{XOR}(x,y,z) &= x \oplus y \oplus z\\
\mathsf{ONX}(x,y,z) &= (x \vee \neg y) \oplus z.
\end{aligned}$$

La sécurité attendue de ces fonctions vient de la répétition d'un grand nombre de tours simples, qui font intervenir des opérations qui interagissent de façon complexe. Une description précise des différentes fonctions de la famille est donnée en Annexe A.

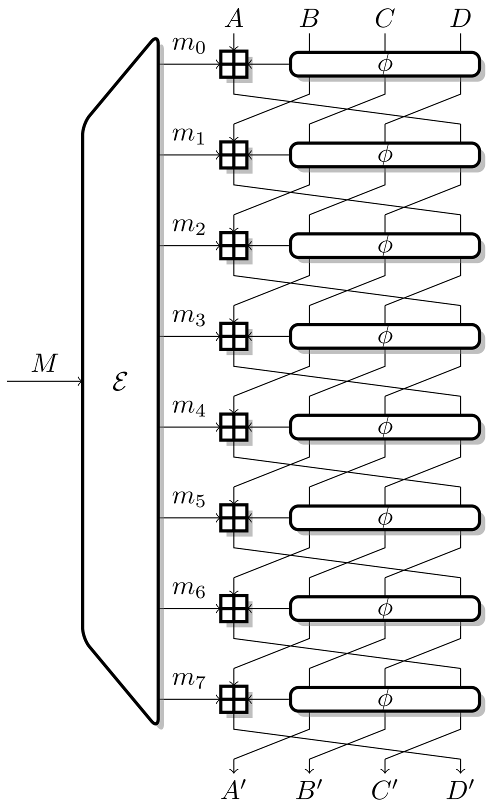
Dans le cas de MD4 et MD5, l'expansion de message est simplement basée sur des permutations du message initial. Le message est composé de 16 mots de 32 bits, et chaque tour (*i.e.* chaque groupe de 16 pas de la fonction de compression) utilise ces 16 mots dans un ordre différent. MD4 utilise 3 tours, donc 48 pas, alors que MD5 utilise 4 tours, donc 64 pas. Les fonctions de tour de MD4 et MD5 sont décrites par la figure 1.6, où les ϕ_i sont des fonctions booléennes bit-à-bit qui dépendent du tour, les rotations s_i dépendent du pas, et les constantes k_i dépendent du tour pour MD4 et du pas pour MD5.

La plupart des fonctions de hachage construites avant 2005 sont inspirées de cette famille. Depuis d'autres types de construction ont vu le jour, avec notamment des fonctions inspirées par les schémas de chiffrement par flot, qui intègrent un petit bloc de message dans un gros état interne, et des constructions basées sur une ou plusieurs permutations fixes au lieu d'utiliser un schéma de chiffrement par bloc. Il y a énormément de diversité dans les fonctions soumises à la compétition SHA-3.

1.4.1 Cryptanalyse

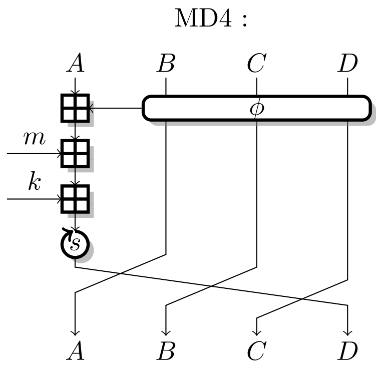
La première attaque contre la famille de MD4 est une attaque en collision contre MD4 due à Dobbertin en 1996 [Dob96a]. Ensuite des attaques contre la fonction de compression de MD5 ont

Figure 1.5 Schéma des fonctions de hachage de la famille de MD4

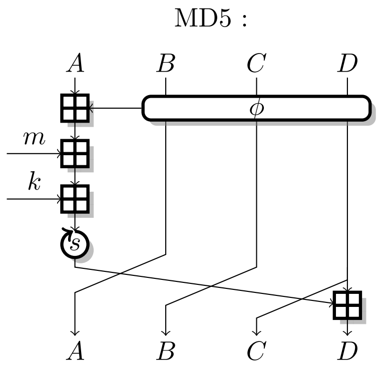


Entrée :	$M \leftarrow$ **Message**
	$(A, B, C, D) \leftarrow$ **Variable de chaînage**
Sortie :	$(A + A', B + B', C + C', D + D')$

Figure 1.6 Fonctions de tour de MD4 et MD5



$$D = \begin{pmatrix} A \boxplus m_i \boxplus k_i \boxplus \\ \Phi_i(D, C, B) \end{pmatrix}^{\lll s_i}$$



$$D = \begin{pmatrix} A \boxplus m_i \boxplus k_i \boxplus \\ \Phi_i(D, C, B) \end{pmatrix}^{\lll s_i} \boxplus B$$

été trouvées par den Boer et Bosselaers[dBB93], puis par Dobbertin [Dob96b], et une attaque en collision sur SHA-0, la version initiale de SHA-1, a été proposée par Chabaud et Joux [CJ98], puis améliorée par Biham *et al.* [BCJ$^+$05]. Cependant, les résultats les plus marquants contre la famille de MD4 ont été obtenus par l'équipe de Xiaoyun Wang en 2005 [WLF$^+$05, WY05, WYY05b]. En utilisant des améliorations des attaques différentielles, ils ont découvert une attaque en collision contre MD5 et SHA-1, et une nouvelle attaque en collision extrêmement efficace contre MD4. Ces résultats ont surpris la communauté cryptographique, et ont suscité un renouveau d'intérêt pour l'étude des fonctions de hachage. Nous avons maintenant une meilleure compréhension de la sécurité de fonctions de la famille de MD4, et de leurs faiblesses.

Plus précisément, les premiers membres de la famille, MD4 et MD5, sont assez faibles et des attaques sérieuses ont été construites. MD4 et MD5 ne devraient plus être utilisées. SHA-1 est aussi plus faible que prévu, même si les attaques actuelles ne sont pas utilisables en pratique. Seules les fonctions de la famille SHA-2 (SHA-224, SHA-256, SHA-384 et SHA-512) peuvent être recommandées pour de nouvelles constructions ou protocoles.

1.4.2 La compétition SHA-3

En réponse à ces attaques, le NIST (National Institute of Standards and Technology) a décidé d'organiser une compétition pour sélectionner une nouvelle fonction de hachage. Cette nouvelle fonction pourra remplacer SHA-2 s'il s'avère que les attaques précédentes peuvent être étendues à SHA-2.

Cette compétition suit le modèle de la compétition organisée pour l'AES : un appel à candidature public a été lancé en 2007, et les fonctions sont étudiées par la communauté cryptographique, afin de permettre au NIST de sélectionner les meilleurs candidats. Le NIST a reçu 64 candidatures, et en a accepté 51 pour le premier tour de la compétition. Après un an de cryptanalyse par la communauté, le NIST a sélectionné 14 candidats pour le second tour en juillet 2009.

Chapitre 2

Présentation de mes travaux

Mon travail durant cette thèse s'est articulé autour de la compétition SHA-3. Dans une première phase, j'ai travaillé sur les attaques de Wang *et al.*, qui ont suscité un renouveau d'intérêt pour les fonctions de hachage et ont motivé l'organisation de cette compétition. Dans cette partie, j'ai étudié les fonctions de hachage MD4 et MD5, et j'ai proposé de nouvelles attaques contre ces fonctions. Dans une deuxième phase, j'ai construit un candidat pour la compétition SHA-3. J'ai utilisé ma connaissance des attaques contre la famille de MD4 pour construire une fonction qui résiste aux principaux types d'attaques. Cette compétition était une grande opportunité pour proposer une nouvelle construction, et j'ai construit mon candidat, SIMD, en intégrant différentes idées originales. SIMD a été accepté pour le deuxième tour de la compétition SHA-3. Enfin, dans une troisième phase, j'ai développé de nouvelles attaques contre certains candidats dans la compétition SHA-3. J'ai donc abordé les deux principaux aspects de la cryptographie symétrique : la construction et l'analyse. Ces deux domaines sont complémentaires, et j'ai largement utilisé mes connaissances en cryptanalyse pour construire SIMD.

Ma thèse s'articule en trois parties, en suivant cette distinction thématique et chronologique.

2.1 Attaques contre la famille de MD4

La première partie est consacrée à l'étude des fonctions de la famille de MD4, et plus particulièrement MD4 et MD5.

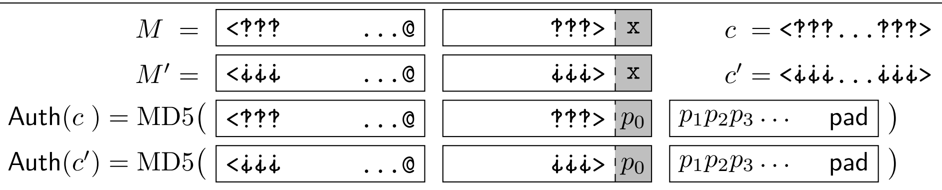
2.1.1 Attaques de Wang *et al.*

Les meilleures attaques contre MD4, MD5, et SHA-1 sont dues à une équipe de chercheurs chinois dirigée par Xiaoyun Wang [WLF$^+$05, WY05, WYY05b]. Leurs attaques sont basées sur le principe des attaques différentielles, mais ils ont introduit quelques nouvelles idées qui rendent ces attaques dévastatrices :

- L'utilisation d'une différentielle signée
- La combinaison de chemins différentiels linéaires et non-linéaires
- L'expression d'un chemin différentiel par des conditions sur l'état interne
- L'utilisation de modifications de message pour satisfaire certaines conditions de manière déterministe.

Mes premiers travaux se basent sur ces attaques, et proposent des améliorations des attaques, ainsi que de nouvelles applications.

Figure 2.1 Attaque contre APOP



2.1.2 Attaque contre APOP [Leu07]

APOP est un des schémas d'authentification disponibles dans le protocole de messagerie électronique POP3. APOP utilise un protocole challenge-réponse très simple : le serveur envoie un identifiant de message c, et le client calcule un haché de l'identifiant et de son mot de passe p, et répond avec $r = \mathrm{MD5}(c\|p)$. Le serveur peut alors calculer le haché de son côté et comparer les valeurs : elles correspondent si le client connaît le bon mot de passe.

Ce protocole peut être attaqué de façon très simple si on sait générer des collisions dans MD5. L'attaquant se met à la place du serveur et construit des challenges qui permettent de révéler de l'information sur le mot de passe. L'idée essentielle est d'utiliser des challenges de longueur variable, de façon à ce que le mot de passe soit à cheval entre deux blocs de message. Pour commencer, on utilise des challenges de 127 octets, pour qu'un seul octet du mot de passe soit dans le deuxième bloc de message. On construit une paire de challenges c, c' telle que $\mathrm{MD5}(c\|\text{'x'}) = \mathrm{MD5}(c'\|\text{'x'})$. Comme le montre la figure 2.1, on aura $\mathrm{MD5}(c\|p) = \mathrm{MD5}(c'\|p)$ si le mot de passe p commence par le caractère 'x'. Ceci permet de tester le premier caractère sans connaître les autres, en utilisant une paire de challenges pour tester chaque caractère, jusqu'à identifier le bon. Ensuite on utilise des challenges de 126 octets c, c' tels que $\mathrm{MD5}(c\|p_0\|\text{'y'}) = \mathrm{MD5}(c'\|p_0\|\text{'y'})$, où p_0 est le premier caractère du mot de passe que l'on a découvert. Ceci permet de tester si le deuxième caractère est égal à 'y'. On peut retrouver les caractères un par un en temps linéaire, en répétant ce processus.

Cette technique avait déjà été utilisée par Bart Preneel et Paul C. van Oorschot pour attaquer la construction *envelope MAC* [PvO96]. Dans le cas de APOP, l'attaque est intéressante car on peut la mettre en œuvre en pratique en utilisant quelques améliorations de l'attaque de Wang. J'ai programmé l'attaque et j'ai vérifié son fonctionnement avec la plupart des clients de messagerie, tels que Thunderbird ou Evolution. Cependant, dans sa forme actuelle, l'attaque demande d'utiliser des challenges qui ne respectent pas la spécification des identifiants de message, et nous avons recommandé aux éditeurs de clients de messagerie de refuser ces identifiants non standards. Avec les versions actuelles de ces logiciels, l'attaque n'est plus utilisable.

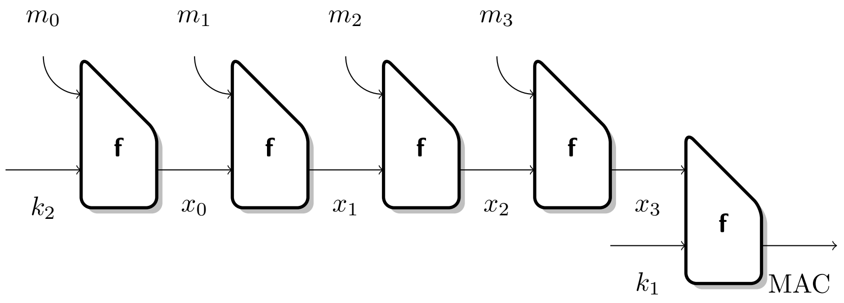
2.1.3 Attaque contre HMAC-MD4 [FLN07b]

HMAC est la principale méthode pour construire un MAC à partir d'une fonction de hachage. HMAC est standardisée par plusieurs organismes, et utilisée dans de nombreux protocoles. La base théorique de HMAC est NMAC, qui utilise une fonction de hachage modifiée, en changeant la valeur initiale. HMAC et NMAC sont définies ainsi :

$$\mathrm{NMAC}_{k_1,k_2}(M) = H_{k_1}(H_{k_2}(M))$$
$$\mathrm{HMAC}_k(m) = H(\bar{k} \oplus \mathrm{opad} \,\|\, H(\bar{k} \oplus \mathrm{ipad} \,\|\, M))$$

NMAC est illustrée par la figure 2.2.

Figure 2.2 La construction NMAC



Durant ma thèse j'ai découvert la première attaque par recouvrement de clef contre HMAC et NMAC utilisés avec une fonction de la famille de MD4. L'attaque permet de retrouver les clefs k_1, k_2 de NMAC, ou une information équivalente à la clef dans le cas de HMAC.

Cette attaque est basée sur des chemins différentiels du type de ceux de Wang *et al.*, mais avec des propriétés particulières. Ici on utilise des chemins dont la probabilité de succès dépend de la valeur initiale. Ceci permet de retrouver des bits de la valeur initiale en utilisant un grand nombre de paires de messages et en détectant s'il y a des collisions. On peut ainsi retrouver la clef simplement en observant seulement des collisions dans la sortie. Pour obtenir des chemins adaptés à cette attaque j'ai développé un algorithme de recherche de chemins différentiels pour MD4, qui permet de trouver les chemins nécessaires pour l'attaque contre HMAC. De plus, cet algorithme permet de trouver des chemins qui améliorent d'autres attaques contre MD4.

La partie la plus difficile d'une attaque contre NMAC est la recherche de la clef externe k_1, car les entrées de H_{k_1} ne sont pas directement contrôlées : elles sont de la forme $H_{k_2}(M)$. Dans notre attaque, on a seulement besoin de construire des paires de messages tels que la *différence* $H_{k_2}(M') \boxminus H_{k_2}(M)$ soit fixée, sans contrôler les valeurs. Nous montrons comment créer ces paires de messages en utilisant une adaptation de l'attaque en collision de Wang *et al.*

Ceci donne une attaque en recouvrement de clef contre NMAC et HMAC avec une complexité de 2^{95}, en utilisant 2^{88} messages choisis.

2.1.4 Attaque en préimage contre MD4 [Leu08a]

Enfin, j'ai trouvé une attaque en préimage contre MD4, qui est la première attaque de ce type contre une fonction de la famille de MD4. Les attaques en préimage sont généralement assez rares, même si plusieurs attaques ont été proposées récemment.

Mon attaque utilise une technique inspirée par les attaques différentielles pour limiter la propagation des différences dans la fonction de compression. L'idée principale est de considérer la fonction de compression comme un système d'équations, où les valeurs calculées à chaque étape sont des variables indépendantes. Pour simplifier le système, on fixe certaines variables à des valeurs particulières, qui permettent d'éliminer une partie des équations. Ensuite, on exprime un mot de la sortie en fonction de certains degrés de liberté restants, et on arrive à inverser l'expression, ce qui permet de choisir la valeur d'un des mots de la sortie. Cela donne une attaque en préimage partielle, qui peut être utilisée pour une attaque en préimage.

Pour étendre l'attaque sur la fonction de compression en une attaque sur la fonction de hachage, je propose une technique qui exploite les particularités de l'attaque en préimage partielle pour ne pas perdre trop d'efficacité. Finalement, on obtient une attaque en préimage de complexité 2^{102}.

Dans ce manuscrit, les améliorations des attaques de Wang *et al.* que j'ai développées sont présentées dans le chapitre 3, et les nouvelles applications de ces attaques dans le chapitre 4. Enfin, l'attaque en préimage contre MD4 est détaillée dans le chapitre 5.

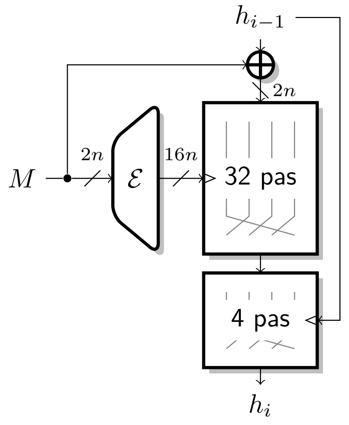
2.2 La fonction de hachage SIMD

Depuis 2008, le NIST (National Institute of Standards and Technology) organise une compétition pour sélectionner une nouvelle fonction de hachage qui sera standardisée et utilisée en remplacement de SHA-2. Cette compétition mobilise une grande partie de la communauté cryptographique mondiale, et les meilleurs cryptographes ont soumis des candidats. En octobre 2008, 64 fonctions ont été soumises et le NIST en a sélectionné 14 pour le deuxième tour de la compétition en juillet 2009.

2.2.1 Construction de SIMD [LBF08]

Pour cette compétition, j'ai construit la fonction de hachage SIMD. La principale innovation de SIMD est son expansion de message, qui est construite en utilisant des techniques de théorie des codes. Grâce à la grande distance minimale du code, on peut prouver que SIMD résiste aux attaques différentielles. Je me suis beaucoup investi dans la création de SIMD, depuis la construction jusqu'à l'analyse de sécurité en passant par les implémentations.

Figure 2.3 Le mode Davies-Meyer modifié de SIMD



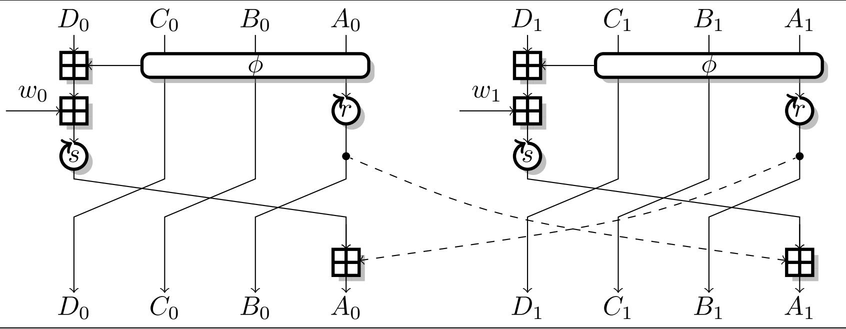
SIMD est construit en utilisant une variante du mode Davies-Meyer, à partir d'un schéma de chiffrement par bloc avec une dérivation de clef particulièrement solide. La figure 2.3 montre le mode de construction de la fonction de compression. La fonction de tour est construite avec 4 schémas de Feistel parallèles (respectivement 8 pour SIMD-512), utilisant des mots de 32 bits, et elle est illustrée par la figure 2.4.

Plus précisément, on peut décrire la fonction de tour ainsi :

$$D_j \leftarrow \left(D_j \boxplus W_j^{(i)} \boxplus \phi^{(i)}(A_j, B_j, C_j)\right)^{\lll s^{(i)}} \boxplus A_{p^{(i)}(j)}{}^{\lll r^{(i)}}$$
$$(A_j, B_j, C_j, D_j) \leftarrow (D_j, A_j^{\lll r^{(i)}}, B_j, C_j)$$

où j identifie un des schémas de Feistel, et i indique le numéro du tour. A, B, C, and D représentent les 4 registres, tandis que les fonctions $\phi^{(i)}$ sont des fonctions booléennes bit-à-bit,

Figure 2.4 La fonction de tour de SIMD



SIMD-256 :
4 Feistel parallèles.
SIMD-512 :
8 Feistel parallèles.

qui sont soit IF soit MAJ suivant le tour. Les schémas de Feistel parallèles échangent des valeurs par les permutations $p^{(i)}$, qui sont de la forme $p^{(i)}(j) = j \oplus \alpha_i$. Il n'y a pas de constantes explicites dans la fonction de tour, mais il y a des constantes implicites dans l'expansion de message.

L'expansion de message de SIMD est la partie la plus innovante. Elle est construite comme un code correcteur avec une grande distance minimale, pour éviter les attaques différentielles. L'expansion de message utilise les opérations suivantes :

1. Une transformation NTT (c'est à dire une transformée de Fourrier sur le corps fini $\mathbb{F}_{257}$) double la taille du message. Cette transformation est en fait un code de Reed-Solomon.
2. On fait deux copies de la sortie de la NTT.
3. La première copie est multiplié par 185, et la deuxième copie par 233. Cette étape double aussi la taille du message, car le résultat est utilisé comme un entier sur 16 bits.
4. Les mots de 16 bits sont permutés, et groupé par paires pour former les mots de 32 bits du message étendu.

SIMD est conçu pour pouvoir être parallélisable efficacement en utilisant les instructions vectorielles disponibles sur la plupart des processeurs. Sur la plupart des architectures, SIMD est plus rapide que SHA-2, et se situe parmi les candidats relativement rapides de la compétition SHA-3.

2.2.2 Analyse de sécurité [BFL10a]

En analysant la sécurité de SIMD, nous avons obtenu trois résultats importants.

Premièrement, nous avons analysé la sécurité de SIMD contre les attaques par symétrie et par auto-similarités. Nous avons montré qu'on peut utiliser des messages particuliers pour exploiter la symétrie entre les schémas de Feistel parallèles, et construire un distingueur pour la fonction de compression. L'idée principale est de construire des messages étendus symétriques, en utilisant le fait que la NTT est linéaire. On peut ainsi facilement imposer des égalités entre les sorties de la NTT, et les étapes suivantes de l'expansion de message préservent les égalités. Ceci montre que les constantes implicites dans l'expansion de message de SIMD ne sont pas suffisantes, et nous prévoyons de rajouter des constantes à la fin de l'expansion de message. Cependant la propriété de symétrie de SIMD est beaucoup plus faible que les propriétés de symétries de *Lesamnta* ou CubeHash, et elle ne remet pas en question la sécurité de la fonction de hachage. En effet, les propriétés de symétries permettent en général d'atteindre un grand nombre d'états symétriques à partir d'un unique état symétrique, mais ça n'est pas le cas pour SIMD.

Deuxièmement, nous avons donné une preuve de sécurité du mode opératoire de SIMD, l'itération de Merkle-Damgård avec un état double, en présence de certaines faiblesses dans la fonction de compression. Notre preuve autorise notamment des faiblesses basées sur des propriétés de symétrie, ou sur des chemins différentiels avec différence non nulle dans la variable de chaînage (y compris des chemins avec probabilité 1). L'idée de base utilisée dans la preuve est que ces distingueurs ne peuvent être utilisés que sur certaines valeurs particulières des variables de chaînage. Or, quand on itère la fonction de compression, on ne contrôle pas les variables de chaînage, et la probabilité de tomber sur une valeur faible est négligeable si la taille de l'état est assez grande. Ce résultat montre que les propriétés obtenues jusqu'ici sur la fonction de compression n'affectent pas la fonction de hachage, et s'applique aussi à d'autres fonctions basées sur le même mode opératoire.

Troisièmement, nous proposons une analyse des attaques différentielles contre SIMD. Nous étudions les chemins différentiels avec une différence non nulle dans le message, car l'expansion de message nous assure qu'il y aura au moins 520 bits de différence dans le message étendu (respectivement 1032 bits pour SIMD-512). Ceci nous permet d'obtenir une borne sur la probabilité des meilleurs chemins différentiels, qui est de l'ordre de $2^{n/2}$. Nous donnons aussi des résultats plus forts basés sur des heuristiques réalistes. Ces résultats sont intéressants, car ils étudient la sécurité du schéma de chiffrement par bloc interne contre les attaques *à clefs liées*, c'est à dire des attaques en utilisant plusieurs clefs reliées par une relation particulière. Ces attaques sont très importantes pour certaines constructions de fonctions de hachage, notamment les fonctions construites selon le schéma de Davies-Meyer. Cependant, ce type d'attaque est assez peu étudié, et il n'y avait pas de techniques connues pour faire ce type d'analyse. En particulier, le schéma de chiffrement par bloc AES a été construit pour résister aux attaques différentielles, et de nombreuses analyses confirment sa résistance à ce type d'attaque [PSLL03]. Cependant, des attaques récentes ont montré qu'il peut être attaqué par des attaques différentielles *à clefs liées* [BKN09].

En combinant les résultats des deuxième et troisième paragraphes, on montre que SIMD résiste aux attaques différentielles : un chemin différentiel avec une différence non nulle sur la variable de chaînage ne permet pas d'attaque la fonction de hachage, tandis qu'un chemin avec une différence non nulle sur le message ne peut pas donner une attaque efficace à cause de l'expansion de message.

La construction de SIMD est présentée dans le chapitre 6, tandis que l'analyse de sécurité est présentée dans le chapitre 7. SIMD est un des 14 candidats sélectionnés pour le deuxième tour de la compétition SHA-3.

2.3 Nouvelles attaques

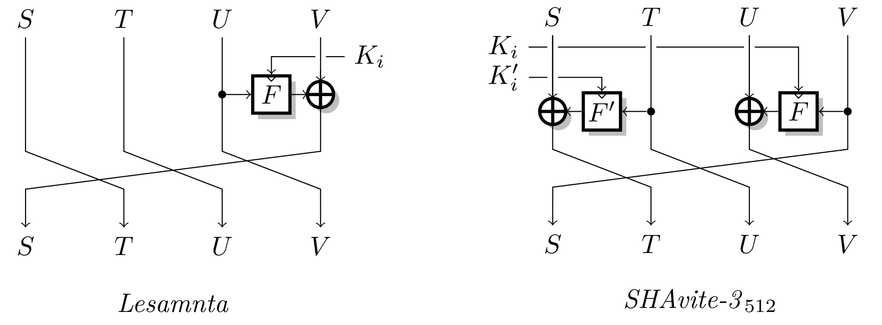
Durant ma thèse j'ai développé des attaques contre plusieurs propositions récentes de fonctions de hachage, principalement des candidats à la compétition SHA-3. Une description plus complète des fonctions évoquées ci-dessous est donnée en Annexe A.

2.3.1 Attaques contre *Lesamnta*

Lesamnta est un candidat pour SHA-3 construit par Shoichi Hirose, Hidenori Kuwakado, et Hirotaka Yoshida [HKY08]. *Lesamnta* est construite en utilisant un schéma de Feistel à 4 brins, représenté par la figure 2.5. La fonction de tour est basé sur l'AES, et alterne des transformations semblables aux SubBytes, ShiftRows et MixColumns de l'AES.

Attaque par auto-similarité [BDFL10]. J'ai développé une attaque contre *Lesamnta* en utilisant une propriété d'auto-similarité, c'est à dire, une transformation simple appliquée à la valeur de chaînage et au message qui se transmet à travers la fonction de tour de *Lesamnta*. Ce type

Figure 2.5 Structures de *Lesamnta* et *SHAvite-3*$_{512}$



d'attaque est inspirée par la propriété de complémentation de DES : si on chiffre le complémentaire $\overline{P}$ d'un texte clair P avec le complémentaire $\overline{K}$ d'une clef DES K, on obtient le complémentaire du chiffré : $DES_{\overline{K}}(\overline{P}) = \overline{DES_K(P)}$. Dans le cas de *Lesamnta*, la transformation utilisée est basée sur l'échange des deux moitiés de l'état, ce qui commute avec les transformations de l'AES. Plus précisément, la fonction de compression de *Lesamnta* vérifie :

$$F((x_0, x_1), (m_0, m_1)) = (h_0, h_1) \iff F((x_1 \oplus 1, x_0 \oplus 1), (m_1 \oplus 1, m_0 \oplus 1)) = (h_1, h_0)$$

Ceci permet une attaque en collision sur la fonction de compression avec complexité $2^{n/4}$, et une attaque en *herding* sur la fonction de hachage avec complexité $2^{n/2}$.

Cette attaque est intéressante car elle exploite la structure des constantes utilisées dans *Lesamnta*. De nombreuses fonctions utilisent des constantes très structurées, par exemple un simple compteur de tours, et ce travail montre que cette structure peut parfois être exploitée. Réciproquement, il est facile d'éviter cette attaque en changeant les constantes utilisées dans *Lesamnta*.

Attaque par annulation [BDLF09]. Pour attaquer la version modifiée de *Lesamnta* avec des constantes non structurées, j'ai développé une nouvelle technique pour attaquer les constructions basée sur un schéma de Feistel généralisé. Les schémas de Feistel généralisés sont utilisés par exemple pour construire une permutation sur $4n$ bits à partir d'une fonction de tour sur n bits, comme le montrent les constructions de la figure 2.5. Ce type de construction est très fréquent pour la construction de schémas de chiffrement par bloc ou de fonctions de hachage, et de nombreux travaux théoriques ont étudié sa sécurité en supposant que les fonctions de tour sont aléatoires et indépendantes, en suivant les travaux de Michael Luby et Charles Rackoff [LR85].

La nouvelle attaque utilise le fait que dans la plupart des constructions pratiques, les fonctions de tour F_i sont en fait dérivées à partir d'une unique fonction (ou permutation) F, sous la forme $F_i(x) = F(K_i \oplus x)$. Ceci permet de construire une propriété d'annulation : si la fonction F reçoit la même entrée à deux tours différents, la sortie sera identique, et la somme s'annule. De plus, dans le cas d'une fonction de hachage, on peut contrôler certaines valeurs de l'état et s'assurer que deux fonctions reçoivent effectivement la même entrée.

L'attaque utilise cette propriété pour limiter la diffusion dans la structure de Feistel. En considérant la fonction de compression comme un système d'équations, on utilise des degrés de libertés pour fixer certaines valeurs, et éliminer des termes dans l'expression d'une des valeurs de sortie. Ceci permet d'avoir une expression suffisamment simple pour l'inverser, et construire une attaque en préimage partielle. La table 2.1 montre comment limiter la diffusion : au tour 7 on

Table 2.1 Valeurs des registres pour 5 tours de *Lesamnta*

i	S_i	T_i	U_i	V_i
2	a	b	c	d
3	$F_2(c) \oplus d$	a	b	c
4	$F_3(b) \oplus c$	$F_2(c) \oplus d$	a	b
5	$F_4(a) \oplus b$	$F_3(b) \oplus c$	$F_2(c) \oplus d$	a
6	$F_5(F_2(c) \oplus d) \oplus a$	$F_4(a) \oplus b$	$F_3(b) \oplus c$	$F_2(c) \oplus d$
7	$\cancel{F_6(F_3(b) \oplus \underline{c})} \oplus \cancel{F_2(c)} \oplus d$	$F_5(F_2(c) \oplus d) \oplus a$	$F_4(a) \oplus b$	$F_3(b) \oplus c$

peut éliminer la dépendance en c de S_7 si $F_3(b) = K_2 \oplus K_6$. En effet dans ce cas on a :

$$F_6(F_3(b) \oplus c) \oplus F_2(c) = F(K_6 \oplus F_3(b) \oplus c) \oplus F(K_2 \oplus c) = 0$$

Alternativement, on peut voir cette propriété en tant que chemin différentiel tronqué : quand on impose certaines contraintes faciles à satisfaire sur l'état, une différence sur un des mots de l'état se propage de façon déterministe, et permet de contrôler la valeur d'un des mots en sortie. Le chemin différentiel utilisé pour attaquer 22 tours de *Lesamnta* est donné dans la table 2.2. Cette attaque utilise trois fois la propriété d'annulation, ce qui se traduit par trois contraintes sur l'état, et il reste un degré de liberté.

Je décris des attaques génériques sur la structure de *Lesamnta* (*i.e.* des attaques qui ne dépendent pas de la fonction de tour F), et des attaques spécifiques qui utilisent des propriétés particulières de la fonction F. Les attaques génériques permettent d'attaquer 22 tours sans utiliser de mémoire, ou 24 tours avec une mémoire de taille $2^{n/2}$, tandis que les attaques spécifiques permettent d'attaquer 24 tours sans mémoire.

Ces attaques montrent la différence entre un schéma de Feistel avec des fonctions de tour indépendantes et les schémas de Feistel utilisés en pratique, où les fonctions de tour sont toutes générées à partir d'une même fonction. De plus, elles suggèrent que les fonctions de hachage basées sur des schémas de Feistel généralisés nécessitent plus de tours que les schémas de chiffrement par blocs basés sur les mêmes constructions.

2.3.2 Attaques contre *SHAvite-3*$_{512}$ [BDLF09, GLM+10]

SHAvite-3 est une fonction construite par Eli Biham et Orr Dunkelman pour la compétition SHA-3 [BD09]. *SHAvite-3* est construite avec un schéma de Feistel, en utilisant plusieurs tours d'AES pour la fonction de tour. La version *SHAvite-3*$_{256}$ utilise 12 tours d'un Feistel à deux brins, tandis que *SHAvite-3*$_{512}$ utilise 14 tours d'un Feistel à quatre brins, représenté par la figure 2.5.

L'attaque par annulation utilisée contre *Lesamnta* s'applique aussi d'une façon similaire contre *SHAvite-3*$_{512}$, bien que la structure du schéma de Feistel soit différente. Cependant, il y a une difficulté supplémentaire à résoudre : les fonctions de tour de *SHAvite-3*$_{512}$ ne sont pas de la forme $F_i(x) = F(K_i \oplus x)$, mais utilisent plusieurs clefs :

$$F(k_i^0, k_i^1, k_i^2, k_i^3, x) = P(k_i^3 \oplus P(k_i^2 \oplus P(k_i^1 \oplus P(k_i^0 \oplus x))))$$

où P est un tour d'AES.

Pour utiliser la propriété d'annulation, il faut donc avoir les mêmes clefs k^1, k^2, k^3 pour deux fonctions de tour différentes, afin qu'elles se simplifient en $F_i(x) = P'(k_i^0 \oplus x)$ et $F_j(x) = P'(k_j^0 \oplus x)$, avec $P'(x) \triangleq P(k_i^3 \oplus P(k_i^2 \oplus P(k_i^1 \oplus P(x)))) = P(k_j^3 \oplus P(k_j^2 \oplus P(k_j^1 \oplus P(x))))$. Ceci impose des contraintes sur l'expansion de message, et nous donnons un algorithme pour trouver des

Table 2.2 Propriété d'annulation sur 22 tours de *Lesamnta*

i	S_i	T_i	U_i	V_i	
0	x	-	-	-	
1	-	x	-	-	
2	-	-	x	-	
3	y	-	-	x	$\underline{x \to y}$
4	x	y	-	-	
5	-	x	y	-	
6	z	-	x	y	$y \to z$
7	$\underline{\text{-}}$	z	-	x	$\underline{x \to y}$
8	x	$\underline{\text{-}}$	z	-	
9	w	x	$\underline{\text{-}}$	z	$\underline{z \to w}$
10	z	w	x	$\underline{\text{-}}$	
11	x_1	z	w	x	$x \to x_1$
12	r	x_1	z	w	$w \to x \oplus r$
13	$\underline{\text{-}}$	r	x_1	z	$\underline{z \to w}$
14	?	$\underline{\text{-}}$	r	x_1	
15	$x_1 + t$	?	$\underline{\text{-}}$	r	$\underline{r \to t}$
16	r	$x_1 + t$	?	$\underline{\text{-}}$	
17	?	r	$x_1 + t$	?	
18	?	?	r	$x_1 + t$	
19	$\underline{x_1}$	?	?	r	$\underline{r \to t}$
20	?	$\underline{x_1}$	?	?	
21	?	?	$\underline{x_1}$	?	
22	?	?	?	$\underline{x_1}$	
FF	?	?	?	x_1	

messages qui respectent les contraintes nécessaires. Ceci permet de construire une attaque contre *SHAvite-3*$_{512}$ réduit à 9 tours, en utilisant deux fois la propriété d'annulation. De plus en utilisant le fait qu'il reste deux degrés de liberté disponibles, on peut étendre l'attaque à 10 tours.

Enfin, l'attaque peut être étendue à des versions comportant plus de tours en réutilisant la propriété d'annulation. Dans le cas de la structure de *SHAvite-3*$_{512}$, les utilisations supplémentaires de la propriété d'annulation donnent des contraintes sur les sous-clefs, mais pas de contraintes sur l'état. Pour satisfaire ces contraintes, nous utilisons des degrés de liberté supplémentaires dans le choix du sel de la fonction de hachage. En effet, la version de *SHAvite-3* du premier tour de la compétition avait un sel faible qui permet de mettre toutes les sous-clefs à zéro, ce qui est suffisant pour la propriété d'annulation. *SHAvite-3* a été modifié pour éviter cette faiblesse, mais nous montrons qu'il est encore possible d'annuler un grand nombre de sous-clefs, et que cela permet aussi d'utiliser la propriété d'annulation. De plus nous décrivons une famille de sels faibles, avec lesquels on peut aussi exploiter la propriété d'annulation. Ceci conduit à une attaque en préimage *à sel choisi* contre les 14 tours de la fonction de compression de *SHAvite-3*$_{512}$.

2.3.3 Attaques contre Edon-$\mathcal{R}$ [Leu10]

EDON-$\mathcal{R}$ [GØM$^+$08] est une fonction de hachage construite par Danilo Gligoroski *et al.* pour la compétition SHA-3. C'était une des fonctions les plus rapides dans le premier tour de la compétition.

J'ai développé une attaque contre la fonction de hachage Edon-$\mathcal{R}$ quand elle est utilisée pour construire un MAC avec la construction *secret-prefix* : $\mathrm{MAC}_k(m) = F(k \| m)$. Cette construction n'est pas la construction recommandée pour construire un MAC, mais l'attaque traduit néanmoins une faiblesse importante de la fonction de hachage. La première étape de l'attaque consiste à demander le MAC d'une paire de messages M, M', où M est un préfixe de M'. Cette étape est similaire à l'attaque en extension contre le MAC *secret-prefix* quand la fonction de hachage a un état de petite taille. Dans le cas de Edon-$\mathcal{R}$, l'état est plus grand que la taille de sortie, donc cela ne donne pas une attaque, mais permet quand même d'obtenir de l'information sur l'entrée et la sortie d'un appel à la fonction de compression. En étudiant la construction de la fonction de compression, on peut se ramener à un petit système d'équations dont la solution donne l'entrée et la sortie complète, et permet de retrouver la clef du MAC. Ce système peut être résolu en utilisant des techniques d'algèbre linéaire pour se ramener à un système partiellement triangulaire. Enfin, en utilisant plusieurs paires de requêtes, on peut obtenir plus d'information sur le système, et le résoudre encore plus efficacement.

Dans le cas de Edon-$\mathcal{R}$256, l'attaque est réalisable en pratique, avec environ 32 appels à l'oracle de MAC, et 2^{30} calculs, avec un précalcul de 2^{52} à faire une fois avant de commencer l'attaque.

La principale autre attaque connue contre Edon-$\mathcal{R}$ est une attaque en préimage avec $2^{2n/3}$ calculs et une mémoire de taille $2^{2n/3}$. Edon-$\mathcal{R}$ n'a pas été sélectionné pour le deuxième tour de la compétition SHA-3.

2.3.4 Attaques contre *ESSENCE* [NPRA+10, BDFL10]

ESSENCE est une fonction de hachage construite par Jason Worth Martin pour la compétition SHA-3 [Mar08]. *ESSENCE* est construite à partir d'un registre à décalage avec 8 mots de 32 bits (64 bits pour *ESSENCE*-512), où la fonction de rétroaction est composée d'une fonction bit-à-bit F de 7×32 bits vers 32 bits, et d'une fonction linéaire L de 32 bits vers 32 bits qui mélange les bits. *ESSENCE* utilise un registre à décalage pour l'expansion de message et un registre à décalage pour traiter la variable de chaînage, avec une rétroaction du premier registre vers le second. Ceci correspond au mode Davies-Meyer.

Avec María Naya-Plasencia, Andrea Röck, Jean-Philippe Aumasson, Yann Laigle-Chapuy, Willi Meier, et Thomas Peyrin, nous avons construit une attaque en collision contre *ESSENCE*. L'attaque est basée sur un chemin différentiel avec un petit nombre de mots actifs. L'idée principale est d'absorber les différences dans la fonction bit-à-bit F, afin de limiter la diffusion. Le chemin est construit pour minimiser le nombre de bits actifs dans le registre qui compresse la variable de chaînage, en utilisant plus de différences dans l'expansion de message. On peut cependant trouver une paire de messages qui suive le chemin efficacement, en utilisant les degrés de liberté du message, et construire une attaque en collision sur la fonction de hachage. L'attaque complète a une complexité de 2^{68} pour *ESSENCE*-256 et 2^{135} pour *ESSENCE*-512. Comme la paire de messages est indépendante de la valeur de chaînage, cela donne aussi une attaque contre des constructions de MAC (notamment HMAC) avec une complexité similaire.

J'ai aussi travaillé sur une autre attaque sur *ESSENCE*, en utilisant les mêmes idées que l'attaque par auto-similarités sur *Lesamnta*. Dans le cas de *ESSENCE*, on utilise une rotation de la variable de chaînage et du message, qui commute avec les opérations bit-à-bit. De plus, la seule opération non bit-à-bit de *ESSENCE*, la fonction linéaire L, est basée sur un LFSR. Une opération de rotation ressemble beaucoup à la structure du LFSR et commute avec le LFSR avec une bonne probabilité. Ainsi, une rotation de 1 bit commute avec la fonction L avec probabilité 1/4. Ceci permet de construire une paire d'entrées reliées par une rotation, telles que les sorties soient aussi reliées par une rotation, pour un coût de 2^{48} opérations. Ce type d'attaque a été indépendamment proposé par Dmitry Khovratovich et Ivica Nikolić sous le nom de cryptanalyse rotationelle, et appliqué notamment à des versions réduites de Skein [KN10].

2.3.5 Attaques contre IFSB [FL08]

IFSB est une fonction de hachage basée sur la théorie des codes construite par Finiasz, Gaborit, et Sendrier [FGS07]. IFSB est une fonction de la famille FSB, pour *Fast Syndrome Based*. Ces fonctions reposent sur le problème du décodage de syndrome, et possèdent une preuve de sécurité : une attaque en collision revient à décoder un syndrome dans un certain code. Cependant, plusieurs versions de FSB ont été cassées : la première version a été attaquée par Coron et Joux [CJ04] en utilisant le paradoxe des anniversaires généralisé de Wagner, et la deuxième version a été attaquée par Saarinen [Saa07] en utilisant une technique de linéarisation. Durant ma thèse, j'ai proposé une nouvelle attaque sur IFSB qui exploite le fait que le code soit cyclique [FL08]. L'attaque utilise des messages périodiques, qui sont transformés en des hachés périodiques par la fonction de compression. Ceci permet de faire une attaque sur une seule période et réduit dramatiquement la sécurité. De plus, pour certaines variantes, on peut utiliser une technique de linéarisation sur une seule période, ce qui conduit à une attaque extrêmement efficace (moins de 1 appel à la fonction de compression pour construire une collision).

Une nouvelle version de FSB a été proposée pour la compétition SHA-3 mais n'a pas été retenue pour le second tour de la compétition.

Dans ce manuscrit, j'ai décidé de présenter les résultats les plus importants, qui introduisent des nouvelles techniques pouvant s'appliquer à différentes fonctions de hachage, et même à certains schémas de chiffrement par blocs. Je présente les attaques par auto-similarités dans le chapitre 8, avec des applications aux fonctions de hachage *Lesamnta* et *ESSENCE*, ainsi qu'aux schémas de chiffrement par blocs XTEA et $\mathcal{PURE}$. Les attaques par annulation contre les schémas de Feistel généralisés sont présentés dans le chapitre 9, avec des applications aux fonctions de hachage *SHAvite-3*$_{512}$ et *Lesamnta*, ainsi qu'au schéma de chiffrement par blocs *SMS4*.

2.4 Autres travaux

Certains de mes travaux ne concernent pas directement la construction et l'analyse des fonctions de hachage, mais étudient comment les fonctions de hachage sont utilisées. J'ai travaillé sur des attaques physiques contre les fonctions de hachage, et sur la modélisation qu'on utilise dans les preuves de sécurité. Ces travaux ne sont pas présenté en détail dans ce manuscrit.

2.4.1 Attaque par canal auxiliaire contre HMAC-SHA1 [FLRV09]

Avec Pierre-Alain Fouque, Denis Réal, et Frédéric Valette, nous avons proposé une attaque par canal caché contre HMAC-SHA1. Pour monter notre attaque, on suppose que l'on a accès à la distance de Hamming entre l'ancien contenu et le nouveau contenu d'un registre à chaque fois qu'il est mis à jour (c'est à dire le nombre de bits du registre qui changent de valeur). Ainsi, pendant le calcul de la fonction de hachage, on recueille de l'information sur la clef. La difficulté est de combiner cette information pour retrouver la clef, en utilisant aussi peu de mesures que possible.

Notre attaque diffère des attaques précédentes car l'attaque sur la fonction compression vise à retrouver le *message*, au lieu d'attaquer la variable de chaînage. Nous utilisons le fait que les mots de message sont utilisés l'un après l'autre dans la fonction de compression, ce qui permet d'avoir de l'information qui ne dépend que du ou des premiers mots. De plus, dans HMAC, la clef est utilisé deux fois en tant que message : une fois pour la fonction de hachage interne, et une fois pour la fonction de hachage externe : $\mathrm{HMAC}_k(m) = H(\bar{k} \oplus \mathrm{opad} \| H(\bar{k} \oplus \mathrm{ipad} \| M))$. En combinant l'information qui vient des deux calculs, on peut retrouver les mots de la clef.

Si on peut mesurer efficacement la distance de Hamming à chaque fois qu'un registre est modifié, on peut retrouver la clef secrète en ayant accès à un seul calcul de HMAC. Des expériences avec un processeur NIOS synthétisé sur un FPGA confirment qu'on peut retrouver la distance de Hamming des registres en mesurant le champ électromagnétique.

2.4.2 Remarques sur le modèle de l'oracle aléatoire [LN09]

Dans un travail avec Phong Nguyen, nous avons étudié la modélisation des fonctions de hachage par l'oracle aléatoire. Nous avons montré que deux schémas prouvés sûrs dans le modèle de l'oracle aléatoire peuvent avoir besoin de propriétés très différentes sur la fonction de hachage. De même, une faiblesse dans la fonction de hachage peut avoir des conséquences très différentes : dans la plupart des schémas de signature, une collision sur la fonction de hachage permet seulement de construire une forge existentielle, mais dans certains cas cela permet de retrouver la clef secrète.

De plus, certaines constructions qui ont été proposées pour instancier un oracle aléatoire à partir d'une fonction de hachage ont de gros problèmes de sécurité. Nous montrons des attaques contre plusieurs constructions proposées dans la littérature. Construire une fonction qui se comporte comme un oracle aléatoire avec une sortie de taille arbitraire n'est pas facile.

2.5 Mes publications

[Leu07]	Message Freedom in MD4 and MD5 Collisions : Application to APOP, Gaëtan Leurent (FSE '07)
[Leu08b]	Practical key-recovery attack against APOP, an MD5 based challenge-response authentication, Gaëtan Leurent (IJACT)
[FLN07a]	Automatic Search of Differential Paths in MD4, Pierre-Alain Fouque, Gaëtan Leurent, et Phong Nguyen (Hash Workshop '07)
[FLN07b]	Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5, Pierre-Alain Fouque, Gaëtan Leurent, et Phong Nguyen (Crypto '07)
[Leu08a]	MD4 is Not One-Way, Gaëtan Leurent (FSE '08)
[FL08]	Cryptanalysis of a Hash Function Based on Quasi-Cyclic Codes, Pierre-Alain Fouque et Gaëtan Leurent (CT-RSA '08)
[LBF08]	SIMD is a Message Digest, Gaëtan Leurent, Charles Bouillaguet, et Pierre-Alain Fouque (SHA-3 submission)
[LN09]	How risky is the Random-Oracle Model ?, Gaëtan Leurent et Phong Nguyen (Crypto '09)
[FLRV09]	Practical Electromagnetic Template Attack on HMAC, Pierre-Alain Fouque, Gaëtan Leurent, Denis Réal, et Frédéric Valette (CHES '09)
[BDFL10]	Another Look at the Complementation Property, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, et Gaëtan Leurent (FSE '10)
[NPRA$^+$10]	Cryptanalysis of ESSENCE, María Naya-Plasencia, Andrea Röck, Jean-Philippe Aumasson, Yann Laigle-Chapuy, Gaëtan Leurent, Willi Meier, et Thomas Peyrin (FSE '10)
[Leu10]	Practical Key Recovery Attack against Secret-IV Edon-$\mathcal{R}$, Gaëtan Leurent (CT-RSA '10)
[BDLF10]	Attacks on Hash Functions based on Generalized Feistel - Application to Reduced-Round Lesamnta and *SHAvite-3*$_{512}$, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, et Gaëtan Leurent (SAC '10)
[GLM$^+$10]	Cryptanalysis of the 10-Round Hash and Full Compression Function of *SHAvite-3*$_{512}$, Praveen Gauravaram, Gaëtan Leurent, Florian Mendel, María Naya-Plasencia, Thomas Peyrin, Christian Rechberger, et Martin Schläffer (Africacrypt '10)
[BFL10b]	Security Analysis of SIMD , Charles Bouillaguet, Pierre-Alain Fouque, et Gaëtan Leurent (SAC '10)

Part One

Analysis of the MD4 family

MD5? That's a dead hash function! This hash function is extinct! It is no longer a hash function! *Ron Rivest*

Abstract

MD4 was one of the first dedicated hash functions. It has been designed by Ron Rivest in 1990 [Riv90], and it is still used today. More importantly, the main ideas used to design MD4 have been reused in the design of many of the hash functions in used today: MD5, SHA-1, SHA-256, SHA-512, RIPEMD, RIPEMD-128 and RIPEMD-160. Each of these functions brings new elements in the design but they do share some very important features. Most current applications using a hash function are using one of those functions, therefore it is very important to study the security of these designs.

After dozen of years of study of the MD4 family, we have learnt unexpected properties of its operating mode, and the earlier members of the family have been broken. The first successful attack was due to Dobbertin on MD4 [Dob96a]. Later, attacks have been found on the compression function of MD5 by den Boer and Bosselaers, and by Dobbertin [dBB93, Dob96b], as well as a collision attack on SHA-0, the initial version of SHA-1 by Chabaud and Joux[CJ98]. However, the most important cryptanalytic results came when a team of researchers led by Wang Xiaoyun found collision attacks against MD5 [WY05] and SHA-1 [WYY05b], the two most widely used hash functions. They also found significant improvements over collision attacks on MD4 and SHA-0, and attacks on other hash functions such as HAVAL or RIPEMD. This caused a renewal of interest in hash function cryptanalysis. We now have a better understanding of the security of this family, its strengths and its weaknesses.

More precisely, the first designs of the family, MD4 and MD5 are quite badly broken, and should not be used anymore. SHA-0 is also quite badly broken, and the attacks show why the tweak that turned it into SHA-1 was necessary. SHA-1 is also considered broken, but the attacks are only theoretical so far. Only SHA-256 and SHA-512 (known together as the SHA-2 family) can still be recommended for use in new protocols.

In this part, I will first describe the main ideas used by Wang to break the hash functions of the MD4 family. Then I will show some applications of these results that I developed: a practical key-recovery attack on the APOP protocol, and a key-recovery attack against HMAC-MD4. In the last chapter I will describe a preimage attack against MD4.

Contents

Chapter 3

Wang *et al.*'s Attacks

3.1 Introduction

In a breakthrough work in 2004–2005, Wang *et al.* discovered devastating collision attacks on the main hash functions from the MD4 family, *e.g.* MD4 [WLF+05], RIPEMD-0 [WLF+05], MD5 [WY05], SHA-0 [WYY05c] and SHA-1 [WYY05b]. Such attacks can find collisions significantly faster than the birthday paradox. The attacks are differential attacks in which one closely follows the inner steps of the underlying compression function, based on a so-called *differential path*.

Roughly speaking, attacks *à la Wang* first select a specific *message difference* Δ such that carefully selected message pairs of the form $(M, M+\Delta)$ will collide for the hash function. To do this, one specifies a *differential path*: during the computation of the hash function on respectively M and $M+\Delta$, the internal state of the hash function varies at each step of the compression function, depending on the particular value of M; the differential path specifies a particular variation that guarantees $(M, M+\Delta)$ to be a hash collision. Next, one computes a *set of sufficient conditions* on the internal state (and sometimes on the message) such that if the message M satisfies all the conditions, then the pair $(M, M+\Delta)$ is guaranteed to follow the differential path, and will therefore give a collision. Finally, using *message modifications*, one shows how to satisfy many conditions deterministically, and therefore efficiently find messages M satisfying all the sufficient conditions. The original papers are quite hard to read, and it took some time for the community to fully understand these attacks.

In the following years, much work [Kli05, YS05, SNKO05, LL05, Ste06, BCH06, Kli06, SNY+06] has been devoted to better understand the new attacks. The early works used the differential paths from Wang *et al.* as a black box, and improved the efficiency of the attacks by designing better message modification techniques. The original attack on MD5 had a complexity of 2^{39} [WY05] and was reduced to 2^{33} [Kli05], then 2^{26} [Ste06], slightly improved by [Kli06], and later 2^{24} [Ste07] using several differential paths for the second block. Similarly, on MD4, Wang *et al.*'s attack had a complexity of 2^{8} [WLF+05] and was improved to 2^{2} [NSKO05]. The final stages where one computes a set of sufficient conditions and finds suitable message modifications are now well-understood. However, the search for a suitable differential path is a much more complex task. It is generally assumed that such differential paths were found "by hand".

Later, new techniques have been developed to automate the construction of differential paths for MD4 [SO06, FLN07a, SWOK07], for MD5 [Ste07], and for SHA-1 [CR06]. This has led to another level of improvement of the attacks: the complexity was again reduced, and new kinds of attacks have been developed beyond simple collisions: chosen-prefix collisions [SLdW07], HMAC key-recovery attacks [FLN07b], an improved APOP attack [SWOK08] and attacks on combiners [MRS09]. The complexity of a collision attack on MD5 has been reduced to 2^{20} by Xie *et al.* [XLF08], and even to 2^{16} by Stevens *et al.* [SSA+09]. On MD4 the complexity of a collision attack is now 2^1 [SWOK07]. On SHA-1 the original attack had a complexity of 2^{69} [WYY05b], and has been improved to 2^{64} using a generalisation of message modification based on auxiliary differential paths [JP07]. Note that several further improvements to the SHA-1 collision attack have been announced: Wang *et al.* announced an attack in 2^{63} [WYY05a], Rechberger *et al.* another attack with a complexity of 2^{60} [MRR07], and recently an attack with complexity 2^{52} was announced [MHP09] and withdrawn. However, these improvements have not been peer-reviewed and published, and no collision has been disclosed. So far the real complexity of a collision attack on SHA-1 is unclear.

Collision attacks on MD4 and MD5 are now very efficient, but the impact on the security of existing hash-based cryptographic schemes is hard to assess, because application often rely on various properties weaker than collision resistance (such as pseudo-randomness, or security when used in some MAC modes).

3.1.1 Impact of these Attacks

One drawback with Wang's attacks when used against practical schemes is that due to the message modification technique, the colliding blocks cannot be chosen and look random. However, these attacks work with any IV, so one can choose a common prefix for the two colliding messages, and the Merkle-Damgård construction allows to add a common suffix to the colliding messages. Therefore, an adversary can choose a prefix and a suffix, but he must somehow hide the colliding blocks (1 block in MD4 and SHA-0, and 2 blocks in MD5 and SHA-1). The poisoned message attack [DL05] exploits this property to create two different PostScript files that display two different chosen texts but whose digests are equal. In this construction, the two different texts are in both PS files and the collision blocks are used by an `if-then-else` instruction to choose which part to display. This attack was extended to other file formats in [GIS06]. Lenstra and de Weger also used the free prefix and free suffix property to create different X.509 certificates for the same *Distinguished Name* but with different secure RSA moduli in [LdW05]. Here, the colliding blocks are hidden in the second part of the RSA moduli.

Later, stronger attacks have appeared. Stevens, Lenstra and de Weger developed a chosen prefix collision attack for MD5 in [SLdW07]. This means we can have a collision between two messages with two *different* chosen prefixes. They used an approach suggested by Wang to find a near-collision for different IVs and used different differential paths to absorb the remaining differences. However, the message blocks leading to the collision are not controlled, and this randomness must still be hidden in the moduli. They used this technique to build colliding X.509 certificates for two different *Distinguished Names*. Later, they improved the attack and managed to create a rogue CA certificate by building a collision between a website certificate and a CA certificate, and having the website certificate signed by a commercial CA [SSA+09].

This shows quite dramatically that MD5 should no longer be used to issue new signature (but signatures that have been issued before these attacks was developed are still safe).

Other applications of Wang *et al.*'s collisions attacks include a practical attack on APOP, the main authentication scheme of the POP3 protocol [Leu07, SWOK08, 4.6]. The attack allows to recover the key in practice in a man-in-the middle scenario, using a few hundred authentications from the client. An attack on HMAC with several hash functions was described in [KBPH06, CY06]. The techniques use Wang's differential path as a black box but with particular messages to recover some keys in the related-key model or to construct advanced distinguishers.

Further attacks have been developed later using tools to generate differential paths. The first one was an attack on HMAC using special paths whose collision probability depends on the value of the IV [FLN07b, 4.2]. Such tools have also been used to attack combiners such as MD5‖SHA-1 more efficiently than using generic attacks [MRS09]. This attack is based on paths that can turn a near collision in MD5 to a collision without using a difference in the message.

3.1.2 My Work

The first part of my work has been to study these attacks, which I will describe in Section 3.3. After getting a good understanding of these attacks, I've been able to improve two specific points of the attacks. Firstly, I show in Section 3.4 how to keep some freedom over the colliding messages, in spite of the message modification technique. This available freedom was widely underestimated and we show a few applications where such freedom is useful. Secondly, I designed an algorithm to automate the search of the differential path in MD4, which is described in Section 3.5. This algorithm can improve some of the paths used in attacks against MD4, as seen in Section 3.5.6.

These two improvements allowed me to design new attacks against some schemes built on top of a hash function. By automating the search for a differential path, I've been able to find specific paths which can be used to attack HMAC-MD4. On the other hand, the degrees of freedom inside the colliding blocks can be used to mount a practical attack against `APOP`, an authentication scheme used in the POP3 mail protocol. Those attacks are described in the next chapter.

This work improves our understanding of the consequences of Wang's discoveries. Collisions as such are not really a concern for most constructions based on a hash function, but Wang's method can be adapted to get more than just mere collisions. In particular, a tool to find differential paths is very useful in order to design new applications of these attacks, and those tools have already led to several such applications.

3.2 Background and Notations

3.2.1 Description of the MD4 Family

MD4 has been introduced by Rivest in 1990. It is still used in some contexts, and the most commonly used hash functions (MD5, SHA-1, SHA-2) are based on the design principles of MD4.

Members of the MD4 family are designed using the Merkle-Damgård paradigm. They use an internal state with the same size n as the output digest, and they do not have a finalisation function. The message is padded and cut into blocks of m bits, and the digest is computed by iterating a compression function f, starting with an initial value IV.

$$\begin{gathered} f : \{0,1\}^{n+m} \to \{0,1\}^n \\ x_{-1} = IV, \qquad x_i = f(x_{i-1}, m_i) \\ F(m_0, m_1, \ldots, m_{k-1}) = x_{k-1} \end{gathered}$$

The padding uses the MD strengthening: it is designed to be invertible, and includes the size of the message. The message is first padded with a single `1` bit followed by a variable number of `0`'s, so that there is just enough room to include the size of the message in the last block (64 bits for MD4, MD5, SHA-1 and SHA-256; and 128 bits for SHA-512). Then the last block is filled with the size of the original message.

The compression functions of members of the MD4 family are built using the Davies-Meyer paradigm with a custom block cipher: $f(x, m) = E_m(x) \boxplus x$. The block cipher is built with a simple unbalanced Feistel scheme, with 32-bit registers (64-bit registers for SHA-512). Each step updates one of the registers, using some function of the other registers, and one input word from the message. The steps are designed to be very efficient using operations implemented in hardware in most processors:

- rotations denoted by $\lll$;
- additions mod 2^{32} $\boxplus$;
- bitwise Boolean functions, among which:

$$\begin{aligned}\mathsf{IF}(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z)\\ \mathsf{MAJ}(x,y,z) &= (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)\\ \mathsf{XOR}(x,y,z) &= x \oplus y \oplus z\\ \mathsf{ONX}(x,y,z) &= (x \vee \neg y) \oplus z.\end{aligned}$$

The security of MD4-based hash functions is based on the iteration of many such simple steps: from 48 in MD4 to 80 in SHA-512. These steps are grouped in rounds, and the step function is the same in a given round. MD4 and MD5 have rounds of 16 steps (respectively, 3 and 4 rounds); SHA-1 has 4 rounds of 20 steps each; SHA-256 has 64 steps, and SHA-512 has 80 steps.

Before being used in the Feistel rounds, the message is processed through a message expansion. In MD4 and MD5, the message expansion is very simple: each round reads the 16 message words in a different order. The general structure of members of the MD4 family is show in Figure 1.5, page 19.

Complete descriptions of the hash functions are given in Appendix A.

MD4. MD4 was the first function of the family. It computes a 128-bit digest, and uses message blocks of 512 bits.

The compression function of MD4 uses 48 steps, with an internal state of four 32-bit registers. In our description of MD4, we will assign the name Q_i to the value computed in the step i: we now have 48 internal state variables, and each one is computed from the 4 preceding ones:

$$Q_i = (Q_{i-4} \boxplus \phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i$$

We use Q_{-4} to Q_{-1} to denote the input chaining value, and the output after the feed-forward corresponds to $Q_{-4} \boxplus Q_{44}$, $Q_{-3} \boxplus Q_{45}$, $Q_{-2} \boxplus Q_{46}$, $Q_{-1} \boxplus Q_{47}$.

MD5. MD5 is an evolution of MD4 designed to increase its security. Like MD4, it computes a 128-bit digest, and uses message blocks of 512 bits.

The compression function of MD5 uses 64 steps, with an internal state of four 32-bit registers. The round function has been modified to increase the diffusion by reusing the value computed at the previous step:

$$Q_i = Q_{i-1} \boxplus (Q_{i-4} \boxplus \phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i$$

The input chaining value is in Q_{-4} to Q_{-1}, and the output after the feed-forward corresponds to $Q_{-4} \boxplus Q_{60}$, $Q_{-3} \boxplus Q_{61}$, $Q_{-2} \boxplus Q_{62}$, $Q_{-1} \boxplus Q_{63}$.

SHA-0 and SHA-1. Because of the evolution of computing power, a 128-bit hash function with 64-bit security against collision attacks became too small for long-term security. This motivated the design of SHA in 1993, with a 160-bit output. SHA uses a state of five 32-bit registers, and process them through 80 steps. SHA was tweaked in 1995 to become SHA-1, and the original SHA was renamed SHA-0. SHA-0 and SHA-1 use an internal state of five 32-bit registers.

The main innovation in SHA is the use of a stronger message expansion. Instead of using permutations of the message words, SHA uses a linear code.

SHA-2. SHA-2 is a set of 4 hash functions designed with even larger output sizes: SHA-224, SHA-256, SHA-384 and SHA-512. SHA-256 and SHA-512 are the main designs of the set; SHA-224 is a truncation of SHA-256 and SHA-384 is a truncation of SHA-512.

The SHA-2 functions use a Feistel structure with eight registers; SHA-256 uses 32-bit registers while SHA-512 uses 64-bit registers. Each step updates two registers using two Boolean functions, but since it only uses one message word, one can write an equivalent description where a single register is updated. SHA-256 has 64 steps, while SHA-512 has 80 steps.

3.2.2 Important Properties of MD4-based hash functions

Invertible Round Function. An important property of the Feistel structure is that it can be easily inverted. Moreover, given the state before round i (Q_{i-4}, Q_{i-3}, Q_{i-2}, Q_{i-1}) and the state after round i (Q_{i-3}, Q_{i-2}, Q_{i-1}, Q_i), one can recover the message word m_i used in that round (see Algorithm 3.1 and 3.2 for explicit formulas).

This is quite useful to design attacks. For instance, given an input chaining value $Q_{-4,-3,-2,-1}$ one can choose a state $Q_0 \dots Q_{15}$ with particular properties, and compute the corresponding message.

Algorithm 3.1 Step functions of MD4

```
1: function MD4StepForward($i$)
2:     $Q_i \leftarrow (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_i \boxplus k_i)^{\lll s_i}$
3: end function
4: function MD4StepBackward($i$)
5:     $Q_{i-4} \leftarrow Q_i^{\ggg s_i} \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus m_i \boxminus k_i$
6: end function
7: function MD4StepMessage($i$)
8:     $m_i \leftarrow Q_i^{\ggg s_i} \boxminus Q_{i-4} \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus k_i$
9: end function
```

Algorithm 3.2 Step functions of MD5

```
1: function MD5StepForward($i$)
2:     $Q_i \leftarrow Q_{i-1} \boxplus (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_i \boxplus k_i)^{\lll s_i}$
3: end function
4: function MD5StepBackward($i$)
5:     $Q_{i-4} \leftarrow (Q_i \boxminus Q_{i-1})^{\ggg s_i} \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus m_i \boxminus k_i$
6: end function
7: function MD5StepMessage($i$)
8:     $m_i \leftarrow (Q_i \boxminus Q_{i-1})^{\ggg s_i} \boxminus Q_{i-4} \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus k_i$
9: end function
```

Absorbing Boolean Function. An important property of the round functions is that they can absorb a difference in one of the inputs. Note that this is not a weakness in the choice of the functions as such: any non-linear function has an absorption property for at least one variable. [1]

1. If a given input x_i of a Boolean function $f(x_0, \dots, x_{k-1})$ can not be absorbed, then the function can be written as $x_i \oplus f'(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{k-1})$.

This is mostly used in the first two rounds of MD4 where the Boolean functions are IF and MAJ. For these functions, if there is a single active input bit, there is a probability 1/2 that the output is active. More precisely, we have the following, for all c and for all x:

Absorb 1st input	Absorb 2nd input	Absorb 3rd input
$\mathsf{IF}(x, \mathsf{c}, \mathsf{c}) = \mathsf{c}$	$\mathsf{IF}(\mathsf{0}, x, \mathsf{c}) = \mathsf{c}$	$\mathsf{IF}(\mathsf{1}, \mathsf{c}, x) = \mathsf{c}$
$\mathsf{MAJ}(x, \mathsf{c}, \mathsf{c}) = \mathsf{c}$	$\mathsf{MAJ}(\mathsf{c}, x, \mathsf{c}) = \mathsf{c}$	$\mathsf{MAJ}(\mathsf{c}, \mathsf{c}, x) = \mathsf{c}$

This shows that the output is independent of x if some conditions on the other inputs are satisfied. In Appendix 3.E we give the exact conditions for all the Boolean functions used in the MD4 family.

In MD4 this can be used to build very low diffusion paths. Indeed, if a difference is absorbed by all the Boolean functions, there is no diffusion, and a single active bit in $Q_{i-4,i-3,i-2,i-1}$ will give a single active bit in $Q_{i,i+1,i+2,i+3}$ four rounds later. This does not apply to MD5 because the round function has been modified to reuse Q_{i-1}.

Den Boer and Bosselaers's Attack on MD5. In [dBB93], den Boer and Bosselaers described a collision attack on the compression function of MD5. They could not use it to break the full hash function, but this result has been used in many recent attacks.

The attack is based on a specific weakness of the MD5 round function. If there is a difference in the most significant bit of each word of the state, and no difference on the message, then we have:

$$\begin{aligned}
Q_i &= Q_{i-1} \boxplus (Q_{i-4} \boxplus \phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i} \\
Q'_i &= Q_{i-1} \oplus 2^{31} \boxplus (Q_{i-4} \oplus 2^{31} \boxplus \phi_i(Q_{i-1} \oplus 2^{31}, Q_{i-2} \oplus 2^{31}, Q_{i-3} \oplus 2^{31}) \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i}
\end{aligned}$$

If the difference is transmitted through ϕ_i, *i.e.* $\phi_i(Q_{i-1} \oplus 2^{31}, Q_{i-2} \oplus 2^{31}, Q_{i-3} \oplus 2^{31}) = \phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \oplus 2^{31}$, then we have $Q'_i = Q_i \oplus 2^{31}$. If we start with initial chaining values such that

$$Q'_{-4} = Q_{-4} \oplus 2^{31} \qquad Q'_{-3} = Q_{-3} \oplus 2^{31} \qquad Q'_{-2} = Q_{-2} \oplus 2^{31} \qquad Q'_{-1} = Q_{-1} \oplus 2^{31}$$

and if all the Boolean functions transmit the difference on bit 31, then we end up with

$$Q'_{60} = Q_{60} \oplus 2^{31} \qquad Q'_{61} = Q_{61} \oplus 2^{31} \qquad Q'_{62} = Q_{62} \oplus 2^{31} \qquad Q'_{63} = Q_{63} \oplus 2^{31}$$

and the difference cancels in the feed-forward. This gives a collision attack on the compression function of MD5.

When three inputs are active, the Boolean functions IF, MAJ, and ONX transmit the difference with probability 1/2, while XOR transmits the difference with probability 1. For the full MD5 we have a probability 2^{-48} for the full path to be followed (MD5 has 32 IF rounds, 16 XOR rounds, and 16 ONX rounds). Therefore, if we use cMD5 to denote the compression function of MD5, we have

$$\text{cMD5}(x, m) = \text{cMD5}(x \oplus \Delta_{msb}, m) \quad \text{with probability } 2^{-48}$$

where $\Delta_{msb} = (2^{31}, 2^{31}, 2^{31}, 2^{31})$.

More precisely, the conditions of the first two rounds are independent of the message, and depend only the chaining value. The input chaining value x has to satisfy the dBB condition: the most significant bit of the last three words are all equal. If it is the case, then the probability of finding a collision with a random message is 2^{-46}.

3.3 Description of Wang *et al.*'s Attacks

Wang *et al.* attacks against the hash functions of the MD4 family are differential attacks, and have the same structure with two main parts:

1. A precomputation phase:
   – choose a message difference Δ;
   – find a differential path;
   – compute a set of sufficient conditions.
2. Search for a message M satisfying all the conditions; then $H(M) = H(M + \Delta)$.

We give an overview of the attack in this section, but more details are given later in this chapter. The differential path specifies how the computations of $F(M)$ and $F(M+\Delta)$ are related: it tells how the differences introduced in the message will evolve in the internal state Q_i. If we choose Δ with a low Hamming weight, and some extra properties, we can find some differences in the Q_i's that are very likely. Then we look at each step of the compression function, and we can express a set of sufficient conditions that will make the Q_i's follow the path. These conditions are on the bits of Q_i, so we cannot directly find a message satisfying them (and the probability that a random message fulfills them is too low).

Wang introduced three important ideas to make such an attack possible:

– The path is specified with a signed difference on the bits, which contains both the modular difference and the XOR-difference;
– Once a path is chosen, it is possible to compute a set of conditions on the internal states Q_i which are sufficient for a message M to collide with $M + \Delta$;
– Some of these conditions can be fulfilled deterministically through message modifications, and the rest will be probabilistic, using random messages until all the conditions are satisfied. The number of messages needed is low enough for a practical attack.

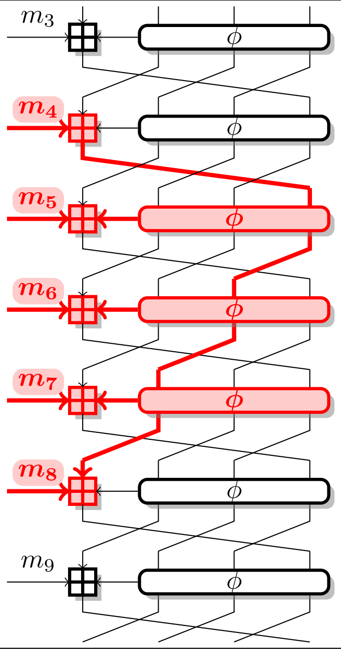
3.3.1 Building a Differential Path

The differential path is the core of any differential attack. Starting from a given difference in the chaining value and in the message, the path describes a possible way for the difference to propagate through the non-linear operations. Such a path is usually built before the attack itself, and the adversary then just has to find a message pair conforming to this path *i.e.* a message pair such that the non-linear operations behave as predicted.

The differential paths used in attacks following Wang's approach are usually divided in two parts: a complex part in the first rounds, and a simple part in the end. The simple part is built by assuming a simple behaviour of the round function; it is usually a *linear path*, built by linearizing the hash function. The message difference is chosen according to this simplified behaviour, and this gives a path with a good probability. However, the full path cannot be based on the simplified behaviour: in MD4 and MD5 it can only be used for one or two rounds, and in SHA-1 the probability is too low if we search for a simple path for the full compression function. To build an efficient path to attack the hash function, Wang *et al.* resorted to a *non-linear* path in the first rounds. This non-linear path is much more complex but it will allow to go from a zero difference in the state to the difference needed by the simple path in the second part. The probability of the *non-linear* path is quite low, but since it is at the beginning of the computation, it is possible to use degrees of freedom in the message to find efficiently a message pair that follows it.

The *linear path* is usually built from local collisions. A local collision happens when a difference in the state is introduced by a message difference, but this difference is cancelled by a new message difference a few rounds later. Figure 3.1 shows an example of local collision. A difference in m_4

Figure 3.1 A local collision



δ_m	δ_A	δ_B	δ_C	δ_D	
Δ	–	–	–	–	
d_5	–	–	–	Δ	$d_5 = \delta\phi_5$
d_6	–	–	Δ	–	$d_6 = \delta\phi_6$
d_7	–	Δ	–	–	$d_7 = \delta\phi_7$
$-\Delta$	Δ	–	–	–	
–	–	–	–	–	

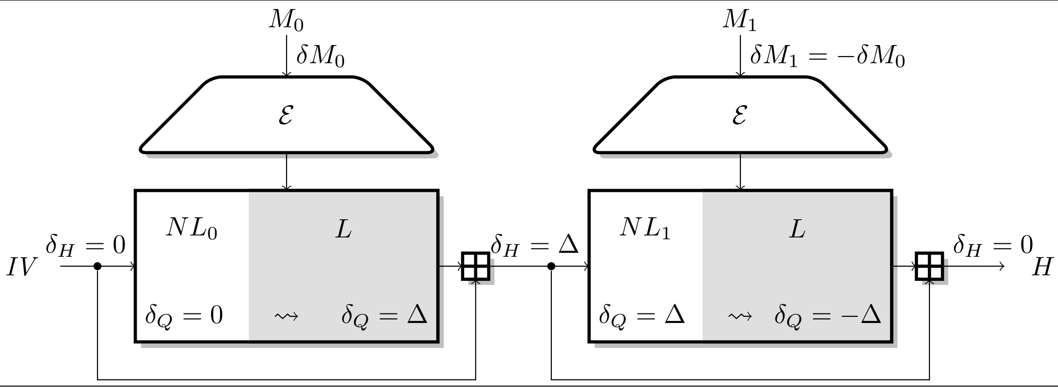
introduces a difference in the state, and differences in m_5, m_6, m_7 are used to stop the propagation of this difference by cancelling the difference in the output of the Boolean function ϕ. Then, a new message difference in m_8 cancels the differences in the state.

Another important tool used by Wang *et al.* is the multi-block technique, which allows more freedom in the choice of the linear part of the path. This technique was also proposed by Biham *et al.* in [BCJ+05], but it reaches its full potential when using non-linear paths. For a single-block attack, one needs a path without state difference at the end of the compression function, but a linear path L with a final state difference Δ can be used in a two-blocks attack as shown in Figure 3.2. One needs two non-linear paths NL_0 and NL_1 to connect the linear path to a zero input difference, and to the Δ input difference, respectively. Then, the feed-forward at the end of the second compression function will give a collision in the hash function. In a two-block collision attack, the first block gives a near-collision with output difference Δ, while the second block gives a pseudo-collision from the input difference Δ.

Using a given path, such as the MD5 path of Wang *et al.* [WY05], finding a message for the first block is more expensive than finding a message for the second block. This is because at the end of the first block, one needs to specify exactly the difference Δ in order to use the second path NL_1. For the second block, there are some degrees of freedom for the sign of the difference on the most significant bits, so the path is slightly less constrained. On the other hand, if one is able to build non-linear differential paths efficiently, it is not necessary to specify exactly the difference Δ at the end of the first block, since NL_1 can be tuned to the actual difference. In this case, the second block will be the most expensive.

Differential Paths for MD4. The most efficient paths for collision attacks on MD4 are based on a local collision in the third round [WLF+05, SWOK07]. The message difference is built in a way that gives a local collision in the third round with a very high probability (2^{-2} or 2^{-1}), and a non-linear path is used in the first round. Alternatively, some paths use a single active message

Figure 3.2 A multi-block collision attack



and a long non-linear part [YWZW05, FLN07b]. In this case, the active words are chosen so that several steps at the end are inactive. These paths tend to have a smaller number of conditions in total, but more conditions in round three: they are not efficient for collision search but can be used for other kinds of attacks.

Differential Paths for MD5. The differential paths on MD5 are based on the attack of den Boer and Bosselaers on the compression function of MD5 [dBB93]. They showed that if the most significant bit of all the state words are active, this gives a differential path with a very high probability. Wang *et al.* used this property to build the linear path of a two-block attack [WY05], and connected it with non-linear paths as shown in Figure 3.2. More recently, new paths have been built which do not reuse the same linear path in both blocks [SWOK07, XLF08]. In this case, the first block yields a difference in the most significant bit of each word, and the second follows the pseudo-collision path of den Boer and Bosselaers [dBB93], without any message difference.

Differential Paths for SHA-1. To attack SHA-1, the linear part of the path is built by combining several local collisions, following the ideas of Chabaud and Joux [CJ98]. The difficulty in doing so is that the message expansion of SHA-1 is a linear code, and one cannot set a random difference in the expanded message. However, it turns out that the message expansion code is a circulant code: a valid codeword can be shifted one step down, or each word can be rotated, and this still gives a valid codeword. To use this property, one starts with a low-weight codeword. This codeword is used as an error vector, and corresponds to the positions where message differences will introduce a difference in the state. Then we use several shifted versions of the error vector to reach the positions were the message difference will be used to cancel the state differences in order to get a local collision. We can combine these codewords by linearity of the message expansion, and this gives the message differences to be used in the attack. The linear path is then used in a two-block attack, with two different non-linear paths.

The structure of a differential path will be explained in more details in Section 3.5, where we describe an algorithm to find differential paths for MD4.

3.3.2 Finding a Conforming Message

When a collision path is known, one has to find a message pair that conforms to the path. In a differential attack against a block cipher, one just tries random inputs until a conforming pair is found. However, in the case of a hash function attack, there is no secret key, so it is possible to build message pairs that follow the path with a better probability than for a random message pair.

Simple message modifications exploit the fact that the first steps of the compression function use the message words one by one. Therefore, if a given message conforms to the first few steps of the path, we can generate many messages that will also conform to these steps by modifying the message words that have not yet been used. This allows to go through the first steps for an amortized cost of one.

More precisely, in the attack by Wang *et al.*, the differential path is used to build a set of sufficient conditions that guarantee that a message pair will result in a collision. This process will be explained in detail in Section 3.5.3. The conditions constrain bits of the state words Q_i and sometimes bits of the message words m_i. For most of the sufficient conditions, Wang *et al.* have designed particular message modifications *i.e.* transformations to apply to the message, essentially flipping a few bits of the message. These message modifications transform a message that does not satisfy a condition into a new message that satisfies it and allow to deterministically build a message that follows the beginning of the path.

Algorithm 3.3 Wang *et al.*'s message finding algorithm

```
1: procedure FindMessageWang
2:     repeat
3:         choose a random message
4:         for all step $i$ do
5:             StepForward($i$)
6:             if not CheckConditions($i$) then
7:                 try to modify the message
8:             end if
9:         end for
10:    until all conditions are fulfilled
11: end procedure
```

We now focus on the attacks on MD4 and MD5. Wang's method to find a message satisfying a set of conditions is roughly described in Algorithm 3.3: one basically picks many messages at random, modifies them to fulfill some of the conditions, and checks if the other conditions are fulfilled.

The best message modifications known allow to satisfy every condition up to round 22 in MD4 (which gives a collision probability of 2^{-2}) and up to round 24 in MD5 (which gives a collision probability of 2^{-29}). Basically, message modifications for the conditions in the first round are very easy, but in the second round it becomes much more difficult because we cannot freely change message words without breaking the Q_i's in the first round (and therefore also in the second round). At the beginning of the second round it is still possible to find message modifications using several message words, but it becomes increasingly difficult as we move forward. Wang's differential paths are chosen with this constraint in mind, and most of their conditions are in the first round and at the beginning of the second.

The algorithm can be rewritten more efficiently: instead of choosing a random message and modifying it, we can choose the Q_i's in the first round and compute the corresponding message. Since all the conditions in MD4 and MD5 are on the Q_i's, this avoids the need for message modifications in the first round.

To further enhance this algorithm, Klima introduced the idea of *tunnels* in [Kli06], which is closely related to Biham and Chen's *neutral bits* used in the cryptanalysis of SHA-0 [BC04].

A tunnel is a message modification that does not affect the conditions up to some step $p_v - 1$ (*point of verification*). The tunnels are built from local collisions in the first round so that the message words used for the local collision are used as late as possible in the second round. This is essentially the same idea as the boomerang attack for hash functions by Joux and Peyrin [JP07].

Therefore, if we have one message that fulfills the conditions up to $p_v - 1$ and τ tunnels, we can generate 2^τ messages that fulfill conditions up to step $p_v - 1$. In the case of MD4 and MD5, this does not change the number of messages we have to try, because we can design message modifications up to round $p_v - 1$. However, it greatly reduces the cost of a single try, and therefore significantly speeds up collision. Alternatively, we can avoid designing advanced message modification and use the tunnels to amortize the cost of finding a first message that conforms up to step $p_v - 1$. The use of tunnels is described in Algorithm 3.8 in Appendix 3.C.

In MD4 and MD5, the point of verification will be in the second round, and we put it after the last condition of the second round (step 22 in MD4, 24 in MD5). We have message modifications for almost every condition before the point of verification, and it seems impossible to find message modifications for round 3 and later.

3.4 Message Freedom Inside Collision Blocks

In this section we show an extension of Wang *et al.*'s attacks which addresses the question of message freedom *inside* the colliding blocks. We show that we can select some part at the end of the message blocks which will collide. Our attack can use any differential path, and only requires a set of sufficient conditions. We are able to choose the last three message words in a one-block MD4 collision, and three specific message words in a two-block MD5 collision with almost no overhead. Additionally, we show how to choose the last 11 words out of the 16 words of one-block MD4 collision with a work factor of about 2^{31} MD4 computations.

An important point is that the technique used is nearly as efficient as the best message modifications on MD4 or MD5, even when we choose some parts of the messages. This contradicts the usual assumption that Wang's collisions are mostly random. As a first application of this new message modification technique, we show that a countermeasure proposed by Szydlo and Yin in [SY06] is almost useless for MD4 and can be partially broken for MD5. This can also be used to handle the padding *inside* the colliding blocks.

3.4.1 Related Work

The first MD4 collision was found by Dobbertin [Dob98a], and his attack has a time complexity of about 2^{20} MD4; it combines algebraic techniques and optimization techniques such as genetic algorithms. His attack also allows to choose a large part of the colliding blocks at some extra cost: one can choose 11 words out of 16 with a complexity of about 2^{30} MD4 computations (little details are given in the paper and only an experimental time complexity is mentioned).

More recently, Yu *et al.* [YWZW05] proposed a differential path for MD4 collisions with a small number of sufficient conditions. This allows to build a collision (M, M') which is close to a given message $\bar{M}$ (about 44 different bits). This is quite different from what we are trying to do since the changed bits will be spread all over the message. We are trying to choose many consecutive bits, which is useful for different applications. However, we also studied this work and propose some improvements in Appendix 3.A.

De Cannière and Rechberger announced at the rump session of CRYPTO '06 that they can find reduced-SHA-1 collisions and choose up to 25% of the message. However, they gave few details on their technique, and the conference version does not talk about this aspect of their work. Their idea seems to be to compute a differential path with the chosen message as conditions.

Lastly, new results by Sasaki *et al.* [SWOK08] present a new differential path for MD5 which is specially tailored to give message freedom in the second block.

3.4.2 Main Ideas

Following the framework of Wang *et al.*, our algorithm takes as input a set of sufficient conditions on the internal state variables Q_i that produces collisions. We will try to find a message M such that when one computes a hash of this message, the conditions on the Q_i's hold. We first describe the general idea that applies to both MD4 and MD5, and we then study in more details those two hash functions.

In contrast to previous works [NSKO05, Kli06, BCH06], we do not focus on a particular path and give message modification techniques for every single condition, but we give a generic algorithm that can take any path as input, like Klima in [Kli05] and Stevens in [Ste06]. Our method is based on two main ideas:

1. We can search for a suitable internal state rather than searching for a suitable message, because the step update function is invertible: if Q_{i+1}, Q_{i+2} and Q_{i+3} are known, then we can compute any one of Q_i, Q_{i+4} or m_i from the two others (see Algorithm 3.1 and 3.2 for explicit formulas).
2. Unlike the original method of Wang *et al.*, our algorithm does *not* fix the Q_i's from the beginning to the end, but we start from the middle. This allows us to deal with the first round and the beginning of the second round at the same time. Additionally we can choose the end of the message before running the search with little extra cost.

Our algorithm resembles the one from Stevens [Ste06], but we fix the state words in a different order, and this allows us to satisfy more conditions deterministically. Our result is quite similar to the improved algorithm presented later in [Ste07, Algorithm 5.3].

3.4.3 The Collision Search Algorithm

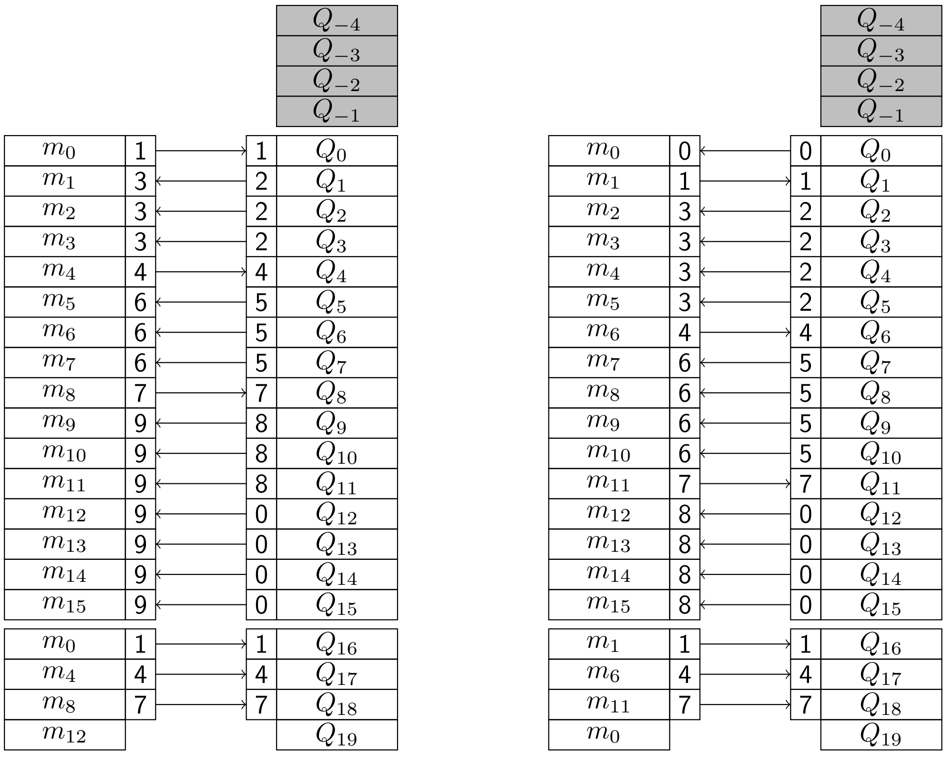
Our algorithm uses two parameters: a *point of choice* p_c of and a *point of verification* p_v. The point of choice denotes the last step to be satisfied deterministically, while the point of verification denotes the last step to be preserved by the tunnels. Between p_c and p_v, the conditions will just be satisfied probabilistically. The value of p_c depends on the message expansion used in the second round: we must have $\pi(16) < \pi(17) < ... < \pi(p_c - 1) < 12$, so we choose $p_c = 19$ in MD4 ($\pi(18) = 8$), and $p_c = 19$ in MD5 ($\pi(18) = 11$).

The key idea of our collision search is to first choose the end of the first round, *i.e.* Q_{12} to Q_{15}. Then we can follow the computations in the first round and in the second round *at the same time*, and choose m_i's that satisfy the conditions in both rounds. There is no difficulty when the first round meets the values we fixed in the beginning: since we only fixed the Q_i's, we just have to compute the corresponding m_i's. More precisely, we will choose the Q_i from step 0 to $\pi(p_c - 1)$, and when we hit a message m_i that is also used in the second round with $i = \pi(j)$, we can modify it to generate a good Q_j since we have already fixed Q_{j-4}, Q_{j-3}, Q_{j-2} and Q_{j-1}. Thus, we can fulfill conditions up to round $p_c - 1$ almost for free. Figure 3.3 shows the order in which the internal words are chosen in MD4 and MD5.

In the end, we will make random choices for the remaining steps ($Q_{\pi(p_c-1)+1}$ to Q_{11}), until we have a message that follows the path up to step $p_v - 1$, and finally we use the tunnels. The algorithm for MD4 is given in Algorithm 3.4.

Since we do not choose the Q_i's in the natural order, we have to modify a little bit the set of sufficient conditions: if we have a condition $Q_{12}^{[k]} = Q_{11}^{[k]}$, we have to use $Q_{11}^{[k]} = Q_{12}^{[k]}$ instead because we choose Q_{12} before Q_{11}.

Figure 3.3 MD4 and MD5 collision search: the numbers show the order in which we choose the different words.



MD4 collision search MD5 collision search

Compared to standard message modifications, our algorithm has an extra cost when we try to satisfy conditions in steps p_c to $p_v - 1$. We only satisfy those conditions randomly while some of them could be fulfilled by advanced message modification, because this gives an easy to describe algorithm without the need to design advanced message modifications. However, one can use advanced message modifications or short tunnels to reduce the cost of this part of the algorithm if needed. Note that the extra cost is quite small anyway, and is amortized through the use of tunnels.

Using Tunnels. Tunnels have been introduced by Klima in [Kli06]. A tunnel is a modification of the message that does not affect the state Q_i of the early steps of the hash function. They are built from local collisions in the first round. Another way to consider a tunnel is that we modify one of the state words in the first round, and we recompute the corresponding message. This does not affect the first round (expect for the word that we modify, of course), but when the message words that we had to modify are used in the second round, this will affect the computation.

Tunnels in MD5 are extensively described in [Kli06], so let us describe a tunnel in MD4. For instance, if we modify Q_2, we have to recompute m_2, m_3, m_4, m_5, and m_6 in the first round. Since m_4 is used at step 17, this message modification will only preserve steps 0 to 16. However, notice the new value of m_4 is computed as:

$$m_4^* = Q_4^{\lll 3} \boxminus Q_0 \boxminus \mathsf{IF}(Q_3, Q_2^*, Q_1) \boxminus k_4 = m_4 \boxplus \mathsf{IF}(Q_3, Q_2, Q_1) \boxminus \mathsf{IF}(Q_3, Q_2^*, Q_1).$$

Therefore as long as the difference in Q_2 is absorbed by the IF function at round 4, m_4 is not actually modified. More precisely, for all k such that $Q_3^{[k]} = 0$ we can modify $Q_2^{[k]}$ without affecting m_4, and this message modification will preserve steps 0 to 20. Similarly, we can study how the new m_5 is computed:

$$m_5^* = Q_5^{\lll 7} \boxminus Q_1 \boxminus \mathsf{IF}(Q_4, Q_3, Q_2^*) \boxminus k_5 = m_5 \boxplus \mathsf{IF}(Q_4, Q_3, Q_2) \boxminus \mathsf{IF}(Q_4, Q_3, Q_2^*).$$

If we also have $Q_4^{[k]} = 1$ we can modify $Q_2^{[k]}$ without affecting m_5, and this message modification will preserve steps 0 to 24.

Therefore, given a message that conforms to the path up to step 22, we can change the value of $Q_2^{[k]}$ for each k satisfying: *(i)* $Q_3^{[k]} = 0$, $Q_4^{[k]} = 1$ and *(ii)* no condition depend on $Q_2^{[k]}$. Then, we can recompute the message in the first round, and we get a new message that still satisfies the path up to step 24 (there are no conditions in step 23 and 24).

To make tunnels more efficient, we can add extra conditions $Q_3^{[k]} = 0$ and $Q_4^{[k]} = 1$ to the path so that more bits of Q_2 can be used.

3.4.4 Choosing a Part of the Message

This method can be extended to allow some message words to be fixed in the collision search. This will make the search for a first message following the path up to the point of verification harder, and it will forbid the use of some tunnels. Actually, we are buying some message freedom with computation time, but we can still find collisions very efficiently. We will show which message words can be chosen, and how to adapt the algorithm to find a first message following the path up to the step $p_v - 1$ with some message words chosen.

Choosing the End. The main advantage of our algorithm is that it allows to choose the end of the message. This is an unsuspected property of Wang's attack, and it will be the core of our attack against `APOP`. Our idea is to split the search in two: first deal with fixed message words, then choose the other internal state variables. This is made possible because our algorithm starts at the end of the first round; the conditions in those steps do not directly fix bits of the message, they also depend on the beginning of the message.

More precisely, if we are looking for collisions where the last t words are chosen, we begin by fixing Q_{12-t}, Q_{13-t}, Q_{14-t} and Q_{15-t}, and we compute Q_{16-t} to Q_{15} using the chosen message words. We can modify Q_{12-t} if the conditions on the first state Q_{16-t} are not satisfied, but for the remaining $t-1$ steps this is impossible because it would break the previous steps. So, these conditions will be fulfilled only statistically, and we might have to try many choices of Q_{12-t}, Q_{13-t}, Q_{14-t}, Q_{15-t} (note that each choice does not cost a full hash computation, but only a few steps).

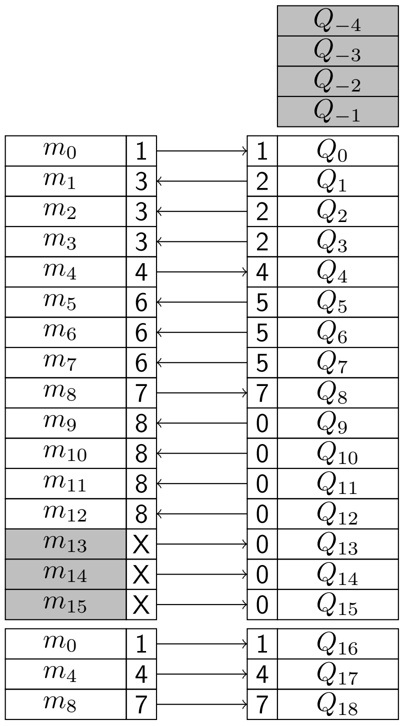
Once we have a partial state that matches the chosen message, we run the same algorithm as in the previous section, but we will be able to deal with fewer steps of the second round due to the extra fixed states Q_{12-t} to Q_{11}. This is illustrated by Figure 3.4, and the full algorithm is given as Algorithm 3.7 in Appendix 3.C.

Choosing the Beginning. Additionally, we can choose the first message words if there are no conditions in the first steps of the compression function. On MD5, this allows to choose m_0. Using the path of [WY05], there are no conditions on Q_0 for the first block, so m_0 is free, but in the second block there are many conditions. On MD4, we can also choose m_0 with the path of [WLF+05]. However, m_0 is used in step 16 at the beginning of the second round, so we have to compute Q_{16} at the beginning of the algorithm, together with $Q_{12-t}, \dots Q_{15}$, which will increase the cost of finding a suitable $Q_{12-t}, \dots Q_{15-t}$.

Algorithm 3.4 MD4 collision finding algorithm

```
 1: function MD4Collision
 2:     ChooseState(12) ChooseState(13) ChooseState(14) ChooseState(15)
 3:     ChooseState(16)
 4:     MD4StepMessage(16)                          ▷ This fixes $m_0$
 5:     MD4StepForward(0)                           ▷ There is no condition on $Q_0$
 6:     repeat
 7:         ChooseState(1) MD4StepMessage(1)
 8:         ChooseState(2) MD4StepMessage(2)
 9:         ChooseState(3) MD4StepMessage(3)
10:         ChooseState(4) MD4StepMessage(4)
11:         MD4StepForward(17) ModifyMessage(17)
12:         MD4StepForward(4)                       ▷ $m_{17} = m_4$ has been modified
13:     until CheckConditions(4)
14:     repeat
15:         ChooseState(5) MD4StepMessage(5)
16:         ChooseState(6) MD4StepMessage(6)
17:         ChooseState(7) MD4StepMessage(7)
18:         ChooseState(8) MD4StepMessage(8)
19:         MD4StepForward(18) ModifyMessage(18)
20:         MD4StepForward(8)                       ▷ $m_{18} = m_8$ has been modified
21:     until CheckConditions(8)
22:     repeat
23:         ChooseState(9) MD4StepMessage(9)
24:         ChooseState(10) MD4StepMessage(10)
25:         ChooseState(11) MD4StepMessage(11)
26:         MD4StepMessage(12)
27:         MD4StepMessage(13)
28:         MD4StepMessage(14)
29:         MD4StepMessage(15)
30:         MD4StepForward(19)
31:         MD4StepForward(20)
32:         MD4StepForward(21)
33:         MD4StepForward(22)
34:     until CheckConditions(19 to 22)
35:     use tunnel $Q_{10}$
36: end function
```

Figure 3.4 MD4 collision search with the last three words fixed



3.4.5 MD4 Message Freedom

Using Wang's EUROCRYPT Path [WLF+05]. To use Wang's EUROCRYPT path, we choose $p_v = 23$ so as to use tunnels only for the third round. Therefore, the tunnels have to preserve the values of m_0, m_4, m_8, m_{12}, m_1, m_5 and m_9 when they modify the Q_i's in the first round.

There are two easy tunnels we can use, in Q_2 and Q_6. $Q_2^{[k]}$ can be used as long as $Q_3^{[k]} = 0$ and $Q_4^{[k]} = 1$, while $Q_6^{[k]}$ can be used if $Q_7^{[k]} = 0$ and $Q_8^{[k]} = 1$. So on the average, we have 8 bits of Q_2 and 8 bits of Q_6 that can be used as a tunnel. If we add some extra conditions on the path we can enlarge these tunnels, but it's not necessary for MD4.

We can use our collision finding algorithm with up to 5 fixed words; Table 3.1 gives the number of conditions we will have to satisfy probabilistically. Of course, the cost of the search increases with t, but with $t = 3$ it should be about 2^9 MD4 computations, which is still very low. Note that this cost is only for the first collision; when computing many collisions, this cost will be shared between all the collisions found through the tunnels, and we expect 2^{14} of them. Another important remark is that this path has a non-zero difference in m_{12}; therefore when choosing more than 3 words, the chosen part in M and M' will have a one bit difference.

Using Yu *et al.*'s CANS Path [YWZW05]. To push this technique to the limit, we will try to use $t = 11$: this leaves only m_0 to m_4 free, which is the minimum freedom to keep a tunnel. In this setting, the conditions in steps 6 to 15 can only be satisfied statistically, which is very expensive with Wang's path [WLF+05] (its goal was to concentrate the conditions in the first round). Therefore we use the path from [YWZW05], which has only 17 conditions in steps 6 to 15.

Table 3.1 Complexity estimates for MD4 collision search with t fixed words, and comparison with Dobbertin's technique [Dob98a]. We assume that a single trial costs 2^{-3} MD4 on average due to early abort techniques.

message words chosen: t	0	1	2	3	4	5	0	11	0	11
path used	[WLF$^+$05]						[YWZW05]		[Dob98a]	
point of choice: p_c	19	19	19	19	18	18	19	17		
point of verification: p_v	23	23	23	23	23	23	23	17		
conditions in steps $17-t$ to 15	0	0	6	**12**	**18**	**24**	0	17		
conditions in steps p_c to p_v-1	**8**	**8**	**8**	8	11	11	11	0		
conditions in steps p_v to N	2	2	2	2	2	2	**17**	**34**		
complexity (MD4 computations $\log_2$)	5	5	5	9	15	21	14	31	20	30

Since we fix almost the full message, the second phase of the search where we satisfy conditions in the first and second round at the same time is very limited, and we have only $p_c = 17$. Then we use the tunnel in Q_0, which is equivalent to iterating over the possible Q_{16}'s, computing m_0 from Q_{16}, and then recomputing Q_0 and m_1, m_2, m_3, m_4. There are 34 remaining conditions, so we will have to use the tunnel about 2^{34} times. Roughly speaking, we break the message search in two: first find $m_0..m_4$ such that the message follows steps 0 to 16, then modify it to follow up to the end by changing Q_0. This path is well suited for this approach, with few conditions well spread over the first two rounds.

This gives us a lot of freedom in MD4 collisions, but the collision search becomes more expensive. Another interesting property of this path is that it only introduces a difference in m_4, so the 11 chosen words will be the same in M and M'.

3.4.6 MD5 Message Freedom

Using Wang's MD5 path [WY05], we will choose $p_v = 24$ so as to use tunnels only for the third round. Therefore, when we modify the Q_i's in the first round to use the tunnels, we have to keep the values of m_1, m_6, m_{11}, m_0, m_5, m_{10}, m_{15} and m_4. We will not describe the available tunnels here, since they are extensively described in Klima's paper [Kli06] (see also [Ste07]).

We use the set of conditions from Stevens [Ste06], which adds the conditions on the rotations that were missing in Wang's paper [WY05]. Note that we had to remove the optimisation condition[2] $Q_{15}^{[17]} \neq Q_{14}^{[17]}$ because it is incompatible with some choices of m_{15}, so we check instead the less restrictive condition on Φ_{15} ($\Phi_{15}^{[31]} = 0$).

As already stated, we can choose m_0 in the first block, and we will see how many words we can choose in the end of the message. With $t = 0$, we set $p_c = 19$, so we have 7 conditions in steps p_c to $p_v - 1$, and after p_v, there are 29 conditions for the first block, and 22 for the second block. With $t = 1$, we use $p_c = 18$, which increases the number of conditions between steps p_c and $p_v - 1$ to 9, but since we will use a lot of tunnels, this has very little impact on the computing time. We can also try to set $t = 2$, but this adds a lot of conditions when we search for the states in the end of the first round. According to our experiments, the conditions on the Q_i's also imply some conditions on m_{14}, so m_{14} could not be chosen freely anyway.

As a summary, in a two-block MD5 collision $(M||N, M'||N')$, we can choose M_0, M_{15} and N_{15}, and the complexity of the collision search is roughly the same as a collision search without extra constraints.

2. Using Stevens notations, it is $Q_{16}[17] = \overline{Q_{15}[17]}$

We only implemented a little number of tunnels, but we find the first block in a few minutes with chosen m_0 and m_{15}, and the second block in a few seconds with chosen m_{15}. This is close to Klima's results in [Kli06].

3.4.7 Applications

Freedom in colliding blocks can be used to break some protocols and to create collisions of special shape. The applications we show here require that the chosen part is identical in M and M', *i.e.* the differential path must not include a difference there.

Fixing the Padding. We can use this technique to find messages with the padding included in the colliding block. For instance, this can be useful to build near-collision of the hash function: if there is a padding block after the near-colliding messages, the near-collision will be completely broken. This will be used in Section 4.4.2.

We can also find collisions on messages shorter than one block, if we fix the rest of the block to the correct padding. An example of a 160-bit MD4 collision is given in Table 3.4 in Appendix 3.B.

Zeroed Collisions. Szydlo and Yin proposed some message preprocessing techniques to avoid collisions attacks against MD5 and SHA-1 in [SY06]. Their idea is to encode the original message in a longer but more structured message. Because the actual message input to the hash function is more constrained, collision attacks should become harder. One of their schemes is called *message whitening*: the message is broken into blocks smaller than 16 words, and the last t words are filled with zeroes. Using our technique we can break this strengthening for MD4 and MD5: in Appendix 3.B we show a 11-whitened MD4 collision in Table 3.6 and a 1-whitened MD5 collision in Table 3.7.

The `APOP` Attack. The main application of this freedom in the colliding block is an attack on the `APOP` authentication scheme. This attack is described in Section 4.6.

3.5 Finding Differential Paths

We now focus on differential paths for the MD4 hash function. We present a new way to search for differential paths, and we hope that this will give a better understanding of the notion of differential path.

The search algorithm has several applications. It allows to improve previous attacks: namely, we have found better paths for MD4 collisions based on [WLF$^+$05, SO06], for second-preimage attacks on MD4 on weak messages [YWZW05], and for the HMAC attack of [WOK08]. More precisely, the new collision paths lead to fewer (or equal) conditions in each of the three rounds of the compression function; the new second-preimage path decreases the total number of conditions, which therefore increases the success probability, and the HMAC path also has fewer conditions, which leads a lower online complexity for the key-recovery attack.

More generally, the search algorithm also allows us to test new message differences, or to search for differential paths with some other specific property. Several new attacks have been proposed by extending Wang *et al.*'s with new differential paths, such as:

- In [SLdW07], Stevens *et al.* used a differential path algorithm for MD5 to create chosen-prefix collisions. The paths are built so that they can iteratively reduce the number of active bits in some special near-collisions.

- In [MRS09], Mendel *et al.* used a differential path algorithm for MD5 to mount an attack on combiners such as MD5$\|$SHA-1. For this application, they used paths that can turn some near-collisions into a collision without using a difference in the message.
- In [SWOK08], Sasaki *et al.* used a new differential path in MD5 to make an IV-bridge in their improved `APOP` attack. The IV-bridge allows to reach a state with a specific difference Δ_{msb} where the most significant bits of each register are active. This allows to use the path of den Boer and Bosselaers [dBB93], which gives a lot of freedom in the second colliding block.
- In [JP07], Joux and Peyrin used the differential path algorithm of [CR06] to build new paths for a more efficient collision attack on SHA-1. In this case, they needed a path with auxiliary differential paths, which are similar to tunnels in MD5.
- In [FLN07b, 4.2], we show how to make a key-recovery attack on HMAC-MD4. For this application we used IV-dependent differential paths whose probability depends on a condition in the IV.

This shows the importance of such tools. In many cases a collision attack is not enough to break a scheme, but if we can tweak the collision attack it can be used more devastatingly.

3.5.1 Notations

In this part we use $\delta(x, y) = y \boxminus x$ to denote the modular difference $y - x \bmod 2^{32}$ and $\partial(x, y) = \left\langle y^{[31]} - x^{[31]}, y^{[30]} - x^{[30]}, \ldots y^{[1]} - x^{[1]}, y^{[0]} - x^{[0]} \right\rangle$ to denote a bitwise signed difference. We will use $\blacktriangle$ and $\blacktriangledown$ to represent $+1$ and -1, and we will give a compact representation by omitting the zeroes, and grouping the bits, *e.g.* $\left\langle \blacktriangle^{[0]}, \blacktriangledown\blacktriangle^{[3,4]}, \blacktriangle\blacktriangle^{[30,31]} \right\rangle$ stands for $\langle 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 1 \rangle$. We use $x^{[k]}$ to represent the $k+1$-st bit of x, that is $x^{[k]} = x^{\ggg k} \bmod 2$ (note that we count bits and steps starting from 0).

We consider two messages M and M', and we use a prime to represent any variable related to the message M' (*e.g.* Q_i', m_i'). As a shortcut, we sometimes use δX (resp. ∂X) to represent $\delta(X, X')$ (resp. $\partial(X, X')$), and Φ_i for $\Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3})$.

When we are given a differential path, we denote it with $\langle \partial_i \rangle_{i=0}^{48}$, and a message pair follows the path if $\partial Q_i = \partial_i$ holds for every step i. We also use δ_i as the desired value of δQ_i.

Before giving the description of the algorithm itself, let us study some useful properties of the different operations used in MD4. The security of MD4 is based on the interaction of incompatible operations, so we have to study how they interact in order to build a differential path. Most of these results are already in the literature (*e.g.* [Dau05], [SNY$^+$06]), but we recall them using our notations.

3.5.2 Mathematical Toolbox

The first tool to study MD4 operations is Wang's ∂ difference. It contains both the modular difference δ and the XOR-difference, and we will often have to switch between these representations.

Relation Between the Modular Difference δ and the Signed Difference ∂. If the value of $\partial(x, y)$ is known, then we know the value of $\delta(x, y)$, but a given $\delta(x, y)$ can be satisfied with several $\partial(x, y)$, with some carry extensions. For instance, if $\delta(x, y) = 2^{27}$, we can have $\partial(x, y) = \left\langle \blacktriangle^{[27]} \right\rangle$ or $\left\langle \blacktriangledown\blacktriangle^{[27,28]} \right\rangle = 2^{28} - 2^{27}$ or $\left\langle \blacktriangledown\blacktriangledown\blacktriangle^{[27,28,29]} \right\rangle = 2^{29} - 2^{28} - 2^{27}$... up to $\left\langle \blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown\blacktriangle^{[27\ldots31]} \right\rangle$ and $\left\langle \blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown^{[27\ldots31]} \right\rangle$.

However, $\delta(x, y)$ can be uniquely determined if we add some extra conditions:

Theorem 3.1. *Let $x, y \in \mathbb{Z}_{2^{32}}$. Then:*

$$\partial(x,y) = \langle \varepsilon_{31}, \varepsilon_{30}, ...\varepsilon_0 \rangle \Leftrightarrow \begin{cases} \sum_{j=0}^{31} \varepsilon_j 2^j = \delta(x,y) \\ \forall j, \varepsilon_j \in \{-1, 0, +1\} \\ \forall j : \varepsilon_j = +1 \Rightarrow x^{[j]} = 0 \\ \forall j : \varepsilon_j = -1 \Rightarrow x^{[j]} = 1 \end{cases}$$

Proof. The "⇒" direction is easy. Reciprocally, let $\langle \varepsilon_j \rangle_{j=0}^{31}$ and $\langle \varepsilon'_j \rangle_{j=0}^{31}$ be two sequences which satisfy the right-hand side conditions. Then we have $\sum(\varepsilon_j - \varepsilon'_j)2^j = 0$, and every $\varepsilon_j - \varepsilon'_j$ is in $\{-1, 0, +1\}$ because of the last two conditions. By reducing this sum modulo two, one sees that $\varepsilon_0 - \varepsilon'_0 = 0 \bmod 2$, therefore $\varepsilon_0 - \varepsilon'_0 = 0$. By iterating, one sees that $\forall j, \quad \varepsilon_j = \varepsilon'_j$. Hence the sequence is unique, and since $\partial(x,y)$ is one candidate, it is the only one. □

Note that some of the conditions depend on x; if only $\delta(x,y)$ is known, we have many possible $\partial(x,y)$, but when bits of x are known, the set of possible $\partial(x,y)$ gets smaller (if x is completely known, we know y and therefore $\partial(x,y)$).

Interactions between the Modular Difference δ and Rotations. To build a differential path, we need to know how a modular difference is affected by a rotation. This turns out to be rather easy, due to the following result:

Theorem 3.2. *Let $a, b \in \mathbb{Z}_{2^{32}}$, $0 \leq s < 32$ and $\alpha = a^{\lll s}, \beta = b^{\lll s}$. Then we may compute $v = \delta(\alpha, \beta)$ from $u = \delta(a,b)$ and a, as follows:*

$$v = \begin{cases} v_1 = u^{\lll s} & \text{if } a + u < 2^{32} \text{ and} \\ & (a \bmod 2^{32-s}) + (u \bmod 2^{32-s}) < 2^{32-s} \\ v_2 = u^{\lll s} \boxplus 1 & \text{if } a + u < 2^{32} \text{ and} \\ & (a \bmod 2^{32-s}) + (u \bmod 2^{32-s}) \geq 2^{32-s} \\ v_3 = u^{\lll s} \boxminus 2^s & \text{if } a + u \geq 2^{32} \text{ and} \\ & (a \bmod 2^{32-s}) + (u \bmod 2^{32-s}) < 2^{32-s} \\ v_4 = u^{\lll s} \boxminus 2^s \boxplus 1 & \text{if } a + u \geq 2^{32} \text{ and} \\ & (a \bmod 2^{32-s}) + (u \bmod 2^{32-s}) \geq 2^{32-s} \end{cases}$$

So, if we know $\delta(a,b)$, we can choose a value for $\delta(a^{\lll s}, b^{\lll s})$, and with some extra conditions on a, this will be the correct one.

Proof. Let $a, b \in \mathbb{Z}_{2^{32}}$, $0 \leq s < 32$ and $\alpha = a^{\lll s}, \beta = b^{\lll s}$. We want to compute $v = \delta(\alpha, \beta)$ from $u = \delta(a,b)$ and a. We will use the integer addition $+$ (in $\mathbb{Z}$) and the modular addition $\boxplus$ (in $\mathbb{Z}_{2^{32}}$), and we express the rotation in the following way:

$$x^{\lll s} = (2^s x \bmod 2^{32}) + \left\lfloor \frac{x \bmod 2^{32}}{2^{32-s}} \right\rfloor.$$

We will use the following result on integer part:

$$\lfloor x + y \rfloor = \begin{cases} \lfloor x \rfloor + \lfloor y \rfloor & \text{if } x + y < \lfloor x \rfloor + \lfloor y \rfloor + 1 \\ \lfloor x \rfloor + \lfloor y \rfloor + 1 & \text{if } x + y \geq \lfloor x \rfloor + \lfloor y \rfloor + 1 \end{cases}$$

If $a + u < 2^{32}$ (in $\mathbb{Z}$), then:

$$\begin{aligned}
\beta - \alpha &= \left(\left\lfloor \frac{a+u}{2^{32-s}} \right\rfloor + 2^s(\cancel{a} + u) \right) - \left(\left\lfloor \frac{a}{2^{32-s}} \right\rfloor + \cancel{2^s a} \right) \\
&= \left\lfloor \frac{a+u}{2^{32-s}} \right\rfloor - \left\lfloor \frac{a}{2^{32-s}} \right\rfloor + 2^s u \\
&= \left\lfloor \frac{u}{2^{32-s}} \right\rfloor + 2^s u \quad \text{or} \quad \left\lfloor \frac{u}{2^{32-s}} \right\rfloor + 2^s u + 1 \\
\beta \boxminus \alpha &= u^{\lll s} \quad \text{or} \quad u^{\lll s} \boxplus 1
\end{aligned}$$

Otherwise, $2^{32} \leq a + u < 2^{33}$:

$$\begin{aligned}
\beta - \alpha &= \left(\left\lfloor \frac{a+u-2^{32}}{2^{32-s}} \right\rfloor + 2^s(a + u) \right) - \left(\left\lfloor \frac{a}{2^{32-s}} \right\rfloor + 2^s a \right) \\
&= -2^s + \left(\left\lfloor \frac{a+u}{2^{32-s}} \right\rfloor + 2^s(a + u) \right) - \left(\left\lfloor \frac{a}{2^{32-s}} \right\rfloor + 2^s a \right) \\
\beta \boxminus \alpha &= u^{\lll s} \boxminus 2^s \quad \text{or} \quad u^{\lll s} \boxminus 2^s \boxplus 1
\end{aligned}$$

This gives the announced result. $\square$

Interactions between the Signed Difference ∂ and the Boolean Functions. The main advantage of the signed difference ∂ over the modular difference δ is to handle the Boolean function. The Φ_i's are bitwise functions and we know for each bit how the input and output are supposed to change between M and M'; if we add some conditions to restrict the inputs, we can make sure the output follows the path. See Table 3.9 in Appendix 3.E for the full conditions.

Based on these tools, we will first show how to compute a set of sufficient conditions once a differential path is given. As such, it can be used to check a given differential path, and it will be the basis of our differential path search algorithm.

3.5.3 Computing a Set of Sufficient Conditions

The technique used here is rather simple, and is the same as used in [SNY+06]. This algorithm (referred to as SC algorithm) will take as input a message difference Δ and a differential path $\langle \partial_i \rangle_{i=0}^{48}$.

The SC algorithm will follow the path backwards from Q_{48} to Q_0, and will recursively compute a set of conditions: at each step i, we assume that the current set is sufficient to ensure that the path is followed from step Q_{i+1} to Q_{48}, and we will add some conditions to extend it to step i. If we look at step $i + 4$ for messages M and M', we have:

$$\begin{aligned}
Q_i &= Q_{i+4}^{\ggg s_{i+4}} \boxminus \phi_{i+4}(Q_{i+3}, Q_{i+2}, Q_{i+1}) \boxminus m_{i+4} \boxminus k_{i+4} \\
Q'_i &= Q'^{\ggg s_{i+4}}_{i+4} \boxminus \phi_{i+4}(Q'_{i+3}, Q'_{i+2}, Q'_{i+1}) \boxminus m'_{i+4} \boxminus k_{i+4}
\end{aligned}$$

We know how to compute $\delta^{\ggg}_{i+4} = \delta(Q_{i+4}^{\ggg s_{i+4}}, Q'^{\ggg s_{i+4}}_{i+4})$ from δ_{i+4} (see Section 3.5.2), and this will add a few conditions on Q_{i+4} (we call these conditions $\lll$-conditions). Then we will add some extra conditions so that the path is followed:

1. We have $\delta Q_i = \delta^{\ggg}_{i+4} \boxminus \delta\Phi_{i+4} \boxminus \delta m_{i+4}$, so we have to make sure that $\delta\Phi_{i+4} = \delta^{\ggg}_{i+4} \boxminus \delta m_{i+4} \boxminus \delta_i$. For this, we select a $\partial\Phi_{i+4}$ corresponding to the desired $\delta\Phi_{i+4}$, and we can ensure that it is followed by adding a few extra conditions on the inputs Q_{i+1}, Q_{i+2}, Q_{i+3} of Φ_{i+4} (see Table 3.9). We call these conditions Φ-conditions.

2. Once we have $\delta Q_i = \delta_i$, we only need a few extra conditions on Q_i to get $\partial Q_i = \partial_i$ by Theorem 3.1. We call these conditions ∂-conditions.

3.5.4 The Differential Path Search Algorithm

Our algorithm is based on the sufficient conditions (SC) algorithm. The basic idea is to run the SC computation, but since we do not know δ_i nor $\delta\Phi_i$, we will assume that $\delta\Phi_i = 0$, which gives $\delta_i = \delta_{i+4}^{\ggg} \boxminus \Delta_{i+4}$: the differences will only appear every 4 steps, and will not propagate in between. This is possible in the first two rounds, because the Boolean functions IF and MAJ can absorb one input difference. Using this basic idea, we find a path with a non-zero difference in $Q_{-4}...Q_{-1}$, that is, a path leading to pseudo-collisions. We denote this trivial path by ϵ.

Then we will run another pass of the algorithm, but we will try to modify the path so as to lower the number of differences in the IV. In fact, we maintain a set of paths $\mathcal{P}$, and each run of the algorithm selects a path, tries to enhance it, and inserts new paths in the set. This basic structure is described in Algorithm 3.5: we will make an extensive use of recursivity to explore the path space. This algorithm will be referred to as the DP algorithm.

Algorithm 3.5 Overview of the differential path search algorithm

```
 1: function PathFind
 2:     $\mathcal{P} \leftarrow \{\epsilon\}$                                ▷ $\epsilon$ is the path with $\delta\Phi_i = 0$
 3:     loop
 4:         Select $P$ in $\mathcal{P}$
 5:         PathStep($P$,$\epsilon$,48)                          ▷ start search from last step
 6:     end loop
 7: end function
 8: function PathStep($P_0$,$P$,$i$)                   ▷ Extend path $P$ to step $i$, following $P_0$
 9:     if $i < 0$ then
10:         Add $P$ to $\mathcal{P}$
11:     else
12:         for all possible choice $P'$ do
13:             PatchTarget($P_0$,$P'$,$i$)
14:         end for
15:     end if
16: end function
17: function PatchTarget($P_0$,$P$,$i$)                ▷ Modify $P$ to fix $IV$ differences in the end
18:     for all possible choice $P'$ do
19:         PatchCarries($P_0$,$P'$,$i$)
20:     end for
21: end function
22: function PatchCarries($P_0$,$P$,$i$)               ▷ Extend some carries to help the next steps
23:     for all possible choice $P'$ do
24:         PathStep($P_0$,$P'$,$i-1$)
25:     end for
26: end function
```

Path Representation. During the computation of a path, we represent the path as $\langle \partial Q_i \rangle_{i=0}^{48}$, where each ∂Q_i is given as 32 values in $\{-1, 0, +1\}$. However, between two passes, this representation is almost useless: when we apply a local modification to a ∂Q_i, the ∂Q_j's for the rest of the path will become quite different.

Therefore we propose a new representation of the path: we will store $\langle \delta\Phi_i \rangle_{i=0}^{48}$. The ∂Q_i's can be efficiently computed from the $\delta\Phi_i$'s, even if there is a little loss of information: a given $\langle \delta\Phi_i \rangle_{i=0}^{48}$ can correspond to many $\langle \partial Q_i \rangle_{i=0}^{48}$ (for instance using different carry extensions), but the algorithm quickly finds a good one. The main advantage of this representation is that a local modification of $\delta\Phi_i$ will not modify the other $\delta\Phi_j$, and we recompute the full path $\langle \partial Q_i \rangle_{i=0}^{48}$ with the new $\partial\Phi_i$'s. In fact, since $\partial\Phi_i = 0$ most of the time, this is a natural description of the path: it shows where something unusual is happening.

Overview of the Algorithm. The function PATHSTEP extends the path one step further, using the same ideas as the SC algorithm at step $i+4$. It assumes the ∂Q_j's and $\delta\Phi_j$'s are chosen for $j > i$. Then, for every possible choice of $\delta^{\lll}_{i+4}$, it computes δQ_i from ∂Q_{i+4} and $\partial\Phi_{i+4}$ and adds the $\lll$-conditions and Φ-conditions. It has to choose a $\partial\Phi_{i+4}$ matching $\delta\Phi_{i+4}$ that is feasible given ∂Q_{i+1}, ∂Q_{i+2}, and ∂Q_{i+3}; if none is available, this branch of the search is aborted. Here we also set $\delta\Phi_i$ to the value it had in the path P_0, so that the new path is similar the old one.

The function PATCHTARGET will then modify $\delta\Phi_i$ so as to remove some unwanted differences in the IV (trying to turn a pseudo-collision path into a collision path).

To finish the step i, the function PATCHCARRIES selects a ∂Q_i corresponding to δQ_i, and extends some carries according to the values $\delta\Phi_{i+1}$, $\delta\Phi_{i+2}$ and $\delta\Phi_{i+3}$. This step is important because we need a non-zero bit in a ∂Q_{j-1}, ∂Q_{j-2} or ∂Q_{j-3} for every non-zero bit in $\partial\Phi_j$. Finally it adds the ∂-conditions.

Correcting Differences. The critical part of the algorithm is the computation of the bits to modify in step i so as to correct a difference in the IV. To change directly a bit $Q_{i_0}^{[k]}$, we can set a non-zero difference in $\Phi_{i_0}^{[k \boxminus s_{i_0}]}$. However, we detect the differences in the IV, and we cannot fix them here; we have to act on a different step and see how the difference affects the IV. The simplest way to do so is to keep $\delta\Phi_i$ unmodified in the rest of the path, which is possible if the difference is absorbed by the other Φ_i's. Therefore we can try to use bit $Q_{i_0+4}^{[k+s_{i_0}]}$ to modify bit $Q_{i_0}^{[k]}$, and so on until we find a bit of Q_{i_0+4t} which can be changed using Φ_{i+4t+4}.

When such a modification succeeds, it will remove one difference in the IV. This simple correction method is already quite efficient: it finds the path from [YWZW05], but not the one from [WLF$^+$05].

Indirect Correction. When searching for more complex paths, we will have some differences in the IV which cannot be dealt with this way. In this case we introduce a difference which does not directly cancel a difference in the IV, but so that the new difference can be used to remove the target difference using the previous method. More precisely, to fix $Q_{i_0}^{[k]}$, we want a difference in some Q_{i_0+4t}, but this requires a difference in the inputs of Φ_{i_0+4t+4}; therefore we try to introduce a difference in Q_{i_0+4t+a}, where $a \in \{1,2,3\}$, using $\Phi_{i_0+4t+a+4t'}$.

When this succeeds, it removes the target difference, but it introduces a new unwanted difference. Hopefully, the new difference will be easier to correct. This method works rather well, and finds many paths using the message difference from [WLF$^+$05].

To include the difference corrections in our algorithm, the function PATCHTARGET has to compute the bits of $\delta\Phi_i$ that can be used to remove a difference in the IV. This is done by considering each IV difference, and computing recursively how it can be corrected at each step of the compression function. See Algorithm 3.6 for a pseudo-code description.

Algorithm 3.6 Details of the bit correcting part of the algorithm

```
function PatchTarget($P_0$,$P$,$i$)
    for all $Q_{i_0}^{[k]}$ bit to fix in $P$ do                ▷ We try each difference, one by one
        PatchTargetBit($P_0$,$P$,$i$,$i_0$,$k$,$\eta_0$)
    end for
end function
function PatchTargetBit($P_0$,$P$,$i$,$i_0$,$k$,$\eta$)         ▷ $\eta$ indirect modifications allowed
    if $i < i_0$ then return
    else if $i = i_0$ then                                        ▷ Direct correction
        modify $P$ on bit $k$ of step $i$                         ▷ Use bit $k$ of $\delta\Phi_i$
        PatchCarries($P_0, P, i$)                                 ▷ Next step of the algorithm
    else
        PatchTargetBit($P_0, P, i, i_0 + 4, k + s_{i_0} \bmod 32, \eta$)
    end if
    if $\eta > 0$ then                                            ▷ Indirect correction
        modify $P_0$ on bit $k$ of step $i_0$
        for all $a \in \{1, 2, 3\}$ do PatchTargetBit($P_0$,$P$,$i$,$i_0 + a$,$k$,$\eta - 1$)
        end for
    end if
end function
```

Impossible Paths. As we compute the differential path and the sufficient conditions at the same time, we do not have to deal with impossible paths, during the execution of the algorithm: if a modification of the paths leads to an impossibility, the search will abort and look for other modifications. However, if the path with $\delta\Phi_i = 0$ is impossible — and this is the case if there are some differences in the third round[3] — the first pass of the algorithm will abort with an incomplete path. Therefore we also add incomplete paths to the set $\mathcal{P}$, and we correct their errors in the same ways we correct differences in the IV.

Exploring the Search Space. In order to avoid spending too much time on uninteresting paths, we have to select an interesting path in the set $\mathcal{P}$. As the indirect corrections are much more expensive than direct ones, we only search for them on paths that have already been run without indirect corrections, and we favour runs without indirect corrections. We implemented the set $\mathcal{P}$ as a priority queue, and our priority function is based on:

– the number of differences in the IV
– the number of conditions
– the number of indirect corrections allowed
– the depth in the tree (*i.e.* the number of runs between the first path and the current path)

To restrict the search space, we also set some limits on the path we are looking for. The difficulty here is to keep enough paths to find the good ones, while cutting enough branches in the search tree to finish in reasonable time. In our algorithm, we limit the length of the carries chains, and the number of total conditions in a path. We also limit the total number of runs of the algorithm, which allows to limit the set $\mathcal{P}$ to a fixed size.

3. for Wang's EUROCRYPT path, the differences in the third round form a local collision, so we can as well run the algorithm only for the first two rounds.

Example. See Table 3.8 in Appendix 3.D for an example of how the algorithm modifies the paths until it has no IV difference. For this path, there is no need for indirect correction.

3.5.5 Comparison with Existing Algorithms

Compared to other algorithms, our algorithm seems very good at finding path with a small number of total conditions. This is mainly possible because it works on the path as a whole, and can optimize it globally, instead of only doing only local modifications. This is specially useful to design attacks where one cannot use message modification techniques, such as HMAC attacks. Unfortunately, it seems difficult to extend the algorithm to work with MD5 or SHA-1, because the diffusion cannot be limited as much as in MD4.

Schläffer and Oswald [SO06]. Our algorithm bears some similarity with the algorithm of Schläffer and Oswald's (*e.g.* the computation of the bits to modify when we have some difference in the IV is very similar to their computation of target differences), but we think the basic idea that rules the algorithm is quite different. Schläffer and Oswald basically try to cancel the differences introduced in the message, while we basically try to compute Q_i from $Q_{i+1}...Q_{i+4}$. Our approach computes the path and the sufficient conditions at the same time, whereas Schläffer and Oswald performed these two steps separately, and had to deal with impossible paths. A more important innovation from our algorithm comes from the possibility of indirect corrections: Oswald and Schläffer had to manually introduce *disturbance difference* which seems to play the same role. It seems that the general structure of their algorithm is not well suited to automate this part. As a result our algorithm finds better ways to choose the indirect corrections, which results in a much better path.

De Cannière and Rechberger [CR06]. This work introduces some important new ideas, and some of them could be used to enhance our algorithm (*e.g.* the generalised differential ∇). However their algorithm as such does not seem really suitable for MD4. It seems that they are only doing local modification to the path, and they cannot correct a difference far from where it was introduced. This feature is well adapted to MD5, SHA-0 and SHA-1 because the step update function will duplicate a difference in the internal state. In MD4, a difference can be absorbed by the Φ_i's, and we can correct it many steps further. Furthermore, the basic idea of iteratively adding conditions seems incompatible with our indirect modification scheme.

Sasaki [SWOK07]. Sasaki introduces an interesting idea in FSE '07 [SWOK07]: he combines forward search and backward search with a meet-in-the-middle approach. More importantly, Sasaki introduces a new message difference and a corresponding path. We have been able to make our algorithm work with this message difference.

Stevens [Ste07]. In his master's thesis, Stevens describe an algorithm to find differential paths in MD5. This algorithm is also based on a meet-in-the-middle approach, with a forward phase and a backward phase. This technique seems quite efficient, but the middle part tends to have long carry chains and a lot of conditions. This is not an issue for collision attacks, but such paths are less efficient for other applications.

3.5.6 Applications

The algorithm was implemented in the C language, and we ran it with different message differences on a desktop computer. We used it to check the paths given by Wang *et al.* in [WLF$^+$05] and [YWZW05].

Yu *et al.*'s CANS Path [YWZW05]. By applying our algorithm to the path of Yu *et al.* [YWZW05], we found that

– This path is rather easy to find and does not require any indirect modifications. Our algorithm finds it in about 0.1 s.
– In [YWZW05], the authors claim that the path can be rotated and gives 32 similar paths using a message difference on the different bits of Q_4, but only 28 paths are actually correct [4].
– If the difference is applied to bit 25 instead of bit 22, the path has only 58 conditions instead of 62. This is a welcomed improvement for applications where one only needs one path with the smallest possible number of conditions, such as attacks against NMAC-MD4 [KBPH06, CY06].

Wang's EUROCRYPT Path [WLF$^+$05]. We also ran our algorithm with the message difference of Wang's EUROCRYPT path [WLF$^+$05], and we found many paths with fewer conditions; the best two are detailed in Path 2 and Path 3. These paths are much harder to find: they need some indirect modifications, and our algorithm takes a few hours to find them (however, a first solution is found in a few minutes, and it already has only 19 conditions on the second round). Our path is also better than the one found by Oswald and Schläffer, see Table 3.2 for a quick comparison. The number of conditions in a path determines the complexity of the collisions finding phase: conditions in the first round cost almost nothing (because message modification in the first round always succeeds); in the beginning of the second round, they cost a little bit more; and in the end of the second round and in the last round they can only be fulfilled statistically, so they have an exponential cost.

Table 3.2 Comparison of MD4 paths

	Conditions			
Path	round 1	round 2	round 3	total
With Wang et al.*'s message difference:*				
Wang *et al.* [WLF$^+$05]	96	25	2	123
Schläffer and Oswald [SO06]	122	22	2	146
Our path [FLN07a]	72	16	2	90
With Sasaki et al.*'s message difference:*				
Sasaki *et al.* [SWOK07]	167	9	1	177

As far as MD4 collisions are concerned, the best path currently known is due to Sasaki *et al.* [SWOK07] and uses another message difference. Unfortunately, we have not been able to make our algorithm work with this message difference.

***IV*-dependent Differential Path.** Using the DP algorithm, we can search for paths with message differences on the first message word m_0, which is used in the first step of the compression function. In this case, the Φ-conditions for the first step will involve Q_{-1} and Q_{-2}, that is, the IV. This kind of IV-dependent path can be used to recover some bits of the IV: if the condition on the

4. it fails if the difference is on bit 17,20,26 or 28.

IV is fulfilled, we will find collisions if we try enough message pairs with the prescribed difference; but if the condition is not fulfilled, that will not happen. This gives us a distinguisher which learns one bit of the IV. If we have several paths, we can recover one bit of IV per path, and do an exhaustive search over the remaining IV bits.

Our algorithm found 22 IV-dependent paths with a one-bit difference $\Delta_0 = 2^k$. Path 5 in Appendix 3.D shows one of them with $k = 0$, and the other ones are obtained with a bit rotation of the whole path. They have one condition on the IV: $Q_{-1}^{[k+s_0]} = Q_{-2}^{[k+s_0]}$, and 79 conditions on the other internal state variables. In Section 4.2, we show a key-recovery attack on HMAC-MD4 based on these paths.

Improved HMAC attack. In [WOK08] L. Wang *et al.* described an improved attack on HMAC/NMAC-MD4 based on pseudo-collision paths. In Appendix 3.D, Path 6 describes a new differential path that can be used to further improve this attack. It has less conditions than the original path, and can be used on different bit positions. Therefore an optimal attack would achieved by using both paths.

Conclusion and Outlook

In this chapter, we described the original attacks of Wang *et al.*, and gave several improvements over the basic attacks. The first improvement is a modified collision search algorithm, allowing some freedom inside the collision blocks. This contradicts the common assumption that the collision blocks look random. The second improvement is an algorithm to build differential paths for MD4. The algorithm can improve some of the previously known paths, and generate a new path with specific constraints.

In the next chapter, we use those improvements to adapt the collision attack into more powerful attacks against new applications.

3.A Collisions with a High Number of Chosen Bits

In the Section 3.4.4, we considered the problem of finding collisions with some chosen *words* but another interesting question involves choosing *bits*, as studied in [YWZW05]. The applications we give in Section 3.4.7 need to choose words and cannot be used with this kind of attack, but choosing many bits of the message could allow to build a meaningful collision.

Following the ideas of Yu *et al.* [YWZW05], we will use a collision path with very few conditions. Such paths are only known in the case of MD4, and we found out that the path from [YWZW05] can be slightly enhanced: if we put the difference in the bit 25 instead of the bit 22, we get only 58 conditions (instead of 62). Now the basic idea is to take a message M, and apply message modifications in the first round: this will give a message M^* that differs in about 10 bit from M (there are 20 conditions in the first round) and it gives a collision $(M^*, M^* + \Delta)$ with probability 2^{-38}. Therefore we will generate about 2^{38} messages M_i close to M and the corresponding M_i^*, and one of them will give a collision.

Little detail is given in Yu *et al.* paper, but we can guess from their collision example that they generated the M_i's by changing m_{14} and m_{15}. This makes the attack more efficient since the M_i^* will all have the same first 14 words, but it will modify about 32 extra bits. Actually, one only needs to iterate over 38 bits, which gives on the average 19 modified bits, but Yu *et al.* used the whole 64 bits.

In fact, if the goal is to have a high number of chosen bits, it is better to choose the M_i in another way: instead of iterating over some bits, we will flip a few bits in the whole message, and iterate over the positions of the differences. We have $\binom{512}{5} \approx 2^{38}$, so it should be enough to select 5 positions, but we will have to run the full message modifications in the first round for every message, which is quite expensive (about 2^{37} MD4 computations). Instead, one can choose 4 positions in the first 480 bits, and two in the last 32 bits: we have $\binom{480}{4}\binom{32}{2} \approx 2^{40}$, and the message modification on the first 15 words will only be run every 2^9 messages; the main cost will be that of testing 2^{38} messages for the second and third rounds: using early abort techniques, this cost will be about 2^{33} MD4 [5].

On average, we expect to have 6 bit differences coming from this iteration, plus 10 coming from the message modification in the first round. An example of such a message is given in Table 3.3, it has 18 bit differences from the target (a block consisting only of 1's), which is much better than achieved by Yu *et al.* (43 bit differences).

5. There are two conditions on step 15, three on step 16 and one on step 17: so $3 \cdot 2^{36}$ messages will stop after 1 step, $7 \cdot 2^{33}$ after 2 steps, 2^{32} after 3 steps, and the remaining 2^{32} messages will need at most 16 steps. This gives less than $95 \cdot 2^{32}$ MD4 steps, that is less than 2^{33} full MD4.

3.B Collision Examples

Table 3.3 A MD4 collision close to 1^{512}.

Message M			
ff ff ff ff	bf ff ff ff	ff f7 ff ff	ff ff df ff
ff ff ff fd	ff ff df ff	ff ff fd ff	ff ef ff ff
ff ff ff ef	ff ff ff fe	ff ff ef 7f	ff 7f ff ff
7f ff fd 7f	ff bf ff ff	ff ff ff ff	ff bf ff fd
Message M'			
ff ff ff ff	bf ff ff ff	ff f7 ff ff	ff ff df ff
ff ff ff ff	ff ff df ff	ff ff fd ff	ff ef ff ff
ff ff ff ef	ff ff ff fe	ff ff ef 7f	ff 7f ff ff
7f ff fd 7f	ff bf ff ff	ff ff ff ff	ff bf ff fd
MD4 without padding			
ff a3 b5 2d	51 63 59 36	11 e5 9a d0	a6 cf 8b 33
MD4 with padding			
59 93 19 84	d0 6f 55 9f	f3 d0 87 4b	c6 24 f4 8d

Table 3.4 A 160-bit MD4 collision.

Message M			
42 79 2d 65	f0 f8 4f d8	d5 7d 86 bf	78 54 9d 67
3f b3 8c aa			
Message M'			
42 79 2d 65	f0 f8 4f d8	d5 7d 86 bf	78 54 9d 67
3f b3 8c ac			
Message M padded to one block			
42 79 2d 65	f0 f8 4f d8	d5 7d 86 bf	78 54 9d 67
3f b3 8c aa	80 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	a0 00 00 00	00 00 00 00
Message M' padded to one block			
42 79 2d 65	f0 f8 4f d8	d5 7d 86 bf	78 54 9d 67
3f b3 8c ac	80 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	a0 00 00 00	00 00 00 00
MD4			
46 6d cb bd	04 66 2c 43	75 12 18 f6	f4 e5 68 71

Table 3.5 `APOP` MD5 collision. These two msg-id's collide if padded with "`bar`"

Message M				
<xxxÑÕç\HSØºä4PD<HO	3c 78 78 78	d1 d5 e7 5c	88 d8 ba e4	34 8b 3c 81
mXÀACn]EBØ\4UBSµiQP1	6d 58 c0 9f	6e 5d 17 d8	5c 34 55 08	b5 69 51 91
Åì"AKSXxô!¿ïFsc'áÒ	c5 ec 22 06	02 78 f4 21	bf ef 46 73	63 27 e1 d2
ÝìVSôGECû49¾VHJxxx@	dd ec 8a f4	47 1b fb 34	39 be 56 89	78 78 78 40
ÚcCòßPºEQÙÖHTNHRIRéý	da 63 43 f2	df 50 ba 05	d9 d6 09 83	8d 52 e9 fd
ÄCC4$½MH-:y6ÇÊ©ÜFS	c4 94 34 24	bd 4d 48 2d	3a 79 36 c7	ca a9 dc 1c
-0:ÞñMWenPUë5{DCþ¾	2d 30 3a de	f1 95 7c 65	6e 8c eb 35	7b 90 fe be
ïL´ïªØ#»ì])ü>	ef 4c b4 ef	aa d8 23 bb	ec 5d 29 fc	3e
Message M'				
<xxxÑÕç\HSØºä4PD<HO	3c 78 78 78	d1 d5 e7 5c	88 d8 ba e4	34 8b 3c 81
mXÀUSn]EBØ\4UBSµiQP1	6d 58 c0 1f	6e 5d 17 d8	5c 34 55 08	b5 69 51 91
Åì"AKSXxô!¿ïFsc§áÒ	c5 ec 22 06	02 78 f4 21	bf ef 46 73	63 a7 e1 d2
ÝìVSôGECû49¾VHTxxx@	dd ec 8a f4	47 1b fb 34	39 be 56 09	78 78 78 40
ÚcCòßPºEQÙÖHTNHRIRéý	da 63 43 f2	df 50 ba 05	d9 d6 09 83	8d 52 e9 fd
ÄCC4¤½MH-:y6ÇÊ©ÜFS	c4 94 34 a4	bd 4d 48 2d	3a 79 36 c7	ca a9 dc 1c
-0:ÞñMWenPUë5{DLþ¾	2d 30 3a de	f1 95 7c 65	6e 8c eb 35	7b 10 fe be
ïL´ïªØ#»ì])|>	ef 4c b4 ef	aa d8 23 bb	ec 5d 29 7c	3e
MD5(M"`bar`")	b8 98 53 57	f8 06 8c 23	72 cf f8 c2	4c 22 c3 81
MD5(M'"`bar`")	b8 98 53 57	f8 06 8c 23	72 cf f8 c2	4c 22 c3 81
MD5(M"`ban`")	40 c6 ee cc	6f e1 e5 2b	53 74 a0 e8	3e f7 4f 54
MD5(M'"`ban`")	02 c1 8c 29	49 91 04 99	8f 88 33 77	a1 eb 81 be

Table 3.6 A 11-whitened MD4 collision.

Whitened Message M			
b9 39 4f 51	3b 43 68 dd	d6 1d 6f 1c	5d b6 a0 b2
44 d4 69 18	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
Whitened Message M'			
b9 39 4f 51	3b 43 68 dd	d6 1d 6f 1c	5d b6 a0 b2
44 d4 69 1a	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
MD4 without padding			
1d ba e9 89	02 22 9f a6	a9 bb 88 f8	30 c1 38 ab
MD4 with padding			
e1 54 1e 65	46 d8 4b 79	db b3 5b b2	13 00 06 9b

Table 3.7 A 1-whitened MD5 collision.

Whitened Message M			
00 00 00 00	23 f9 5a 1c	c8 4f 18 59	1b ef 74 a9
02 7a b6 bf	ff 47 53 be	c3 29 a9 dd	b1 1e 62 94
d1 2c 24 05	07 5e b4 42	1b e2 58 72	25 83 b2 52
12 97 d8 24	ca 8c ae 13	e1 e9 34 77	00 00 00 00
54 01 42 6f	b4 b5 4a 77	d2 15 90 5a	7a 42 cf dd
9f 76 5b 37	90 dd 7e 3d	0a fd 77 d7	d1 4c 55 de
49 ff 3e f2	f5 52 b8 86	72 c0 49 7e	80 ac d1 1f
c7 38 b4 96	a8 d3 73 f0	4b 5c d8 f2	00 00 00 00
Whitened Message M'			
00 00 00 00	23 f9 5a 1c	c8 4f 18 59	1b ef 74 a9
02 7a b6 3f	ff 47 53 be	c3 29 a9 dd	b1 1e 62 94
d1 2c 24 05	07 5e b4 42	1b e2 58 72	25 03 b3 52
12 97 d8 24	ca 8c ae 13	e1 e9 34 f7	00 00 00 00
54 01 42 6f	b4 b5 4a 77	d2 15 90 5a	7a 42 cf dd
9f 76 5b b7	90 dd 7e 3d	0a fd 77 d7	d1 4c 55 de
49 ff 3e f2	f5 52 b8 86	72 c0 49 7e	80 2c d1 1f
c7 38 b4 96	a8 d3 73 f0	4b 5c d8 72	00 00 00 00
MD5 without padding			
98 28 90 7a	75 75 ae 7a	25 f9 80 94	62 ea 52 76
MD5 with padding			
7c c4 3c db	a7 a6 f6 9e	c5 10 8e 46	95 00 fd 82

3.C Algorithms

All the algorithms take the set of conditions as an implicit input, and will modify a shared state containing the m_i's and the Q_i's.

Algorithm 3.7 Our collision finding algorithm

```
 1: procedure FindMessage($p_v, p_c, t$)
 2:     repeat
 3:         choose $Q_{12-t}$, $Q_{13-t}$, $Q_{14-t}$, $Q_{15-t}$
 4:         if $t \neq 0$ then
 5:             StepForward($16 - t$)
 6:             FixState($16 - t$)
 7:             StepBackward($16 - t$)
 8:             if not CheckConditions($12 - t$) then
 9:                 goto 3
10:             for $17 - t \leq i < 16$ do
11:                 StepForward($i$)
12:                 if not CheckConditions($i$) then
13:                     goto 3
14:         $i \leftarrow 0$
15:         for $16 \leq j < p_c$ do
16:             while $i < \pi(j)$ do
17:                 choose $Q_i$
18:                 StepMessage($i$)
19:                 $i \leftarrow i + 1$
20:             StepForward($j$)
21:             FixState($j$)
22:             StepMessage($j$)
23:             StepForward($i$)
24:             if not CheckConditions($i$) then
25:                 goto 16
26:         for $\pi(p_c - 1) + 1 \leq i < 12 - t$ do
27:             choose $Q_i$
28:             StepMessage($i$)
29:         StepMessage(12-t ... 15-t)
30:         for $p_c \leq i < p_v$ do
31:             StepForward($i$)
32:             if not CheckConditions($i$) then
33:                 goto 26
34:         for all tunneled message do
35:             for $p_v \leq i < N$ do
36:                 StepForward($i$)
37:                 if not CheckConditions($i$) then
38:                     use the next message
39:     until all conditions are fulfilled
```

Algorithm 3.8 Klima's message finding algorithm

```
procedure FindMessageKlima
    repeat
        for $0 \leq i < 16$ do
            choose $Q_i$
            StepMessage($i$)
        end for
        for $16 \leq i < p_v$ do
            StepForward($i$)
            if not CheckConditions($i$) then
                modify the message
            end if
        end for
        for all tunneled message do
            for $p_v \leq i < N$ do
                StepForward($i$)
                if not CheckConditions($i$) then
                    use the next message
                end if
            end for
        end for
    until all conditions are fulfilled
end procedure
```

3.D Differential Paths

All the paths given in this section will use the notations defined in Section 3.5. Moreover, there are two extra differences with Wang's tables:

– The ∂-conditions are not included in the table, since they can be easily be deduced from ∂_i (*e.g.* $Q_1^{[6]} = 1$ in step 1 of Path 2).

– The Φ-conditions and $\lll$-conditions are listed in the step where they are needed, rather than in the step were the message modification will be done. This makes the paths easier to read, but must be taken into account if one wants to count the conditions in each round.

Path 1 Original path of Wang *et al.* [WLF+05].

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	Φ-conditions and ⋘-conditions
0	3				
1	7	$\langle \blacktriangle^{[31]} \rangle$		$\langle \blacktriangle^{[6]} \rangle$	
2	11	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$		$\langle \blacktriangledown^{[7]}, \blacktriangle^{[10]} \rangle$	${}^{[6]}_{0} = {}^{[6]}_{-1}$
3	19		$\langle \blacktriangle^{[6]} \rangle$	$\langle \blacktriangle^{[25]} \rangle$	${}^{[6]}_{2} = 1, {}^{[7]}_{1} = {}^{[7]}_{0}, {}^{[10]}_{1} = {}^{[10]}_{0}$
4	3				${}^{[6]}_{3} = 1, {}^{[7]}_{3} = 0, {}^{[10]}_{3} = 0, {}^{[25]}_{2} = {}^{[25]}_{1}$
5	7			$\langle \blacktriangle^{[13]} \rangle$	${}^{[7]}_{4} = 1, {}^{[10]}_{4} = 1, {}^{[25]}_{4} = 0$
6	11			$\langle \blacktriangle\blacktriangle\blacktriangledown^{[18..20]}, \blacktriangle^{[21]} \rangle$	${}^{[13]}_{4} = {}^{[13]}_{3}, {}^{[25]}_{5} = 1$
7	19			$\langle \blacktriangledown\blacktriangledown\blacktriangle^{[12..14]} \rangle$	${}^{[13]}_{6} = 0, {}^{[18]}_{5} = {}^{[18]}_{4}, {}^{[19]}_{5} = {}^{[19]}_{4}, {}^{[20]}_{5} = {}^{[20]}_{4}, {}^{[21]}_{5} = {}^{[21]}_{4}$
8	3		$\langle \blacktriangle^{[13]} \rangle$	$\langle \blacktriangle^{[16]} \rangle$	${}^{[12]}_{6} = {}^{[12]}_{5}, {}^{[13]}_{6} = 0, {}^{[14]}_{6} = {}^{[14]}_{5}, {}^{[18]}_{7} = 0, {}^{[19]}_{7} = 0, {}^{[20]}_{7} = 0, {}^{[21]}_{7} = 0$
9	7		$\langle \blacktriangledown\blacktriangledown\blacktriangle^{[12..14]}, \blacktriangle\blacktriangle\blacktriangledown^{[18..20]} \rangle$	$\langle \blacktriangle^{[19]}, \blacktriangledown\blacktriangledown\blacktriangle^{[20..22]}, \blacktriangledown^{[25]} \rangle$	${}^{[12]}_{8} = 1, {}^{[13]}_{8} = 1, {}^{[14]}_{8} = 1, {}^{[16]}_{7} = {}^{[16]}_{6}, {}^{[18]}_{8} = 0, {}^{[19]}_{8} = 0, {}^{[20]}_{8} = 0, {}^{[21]}_{8} = 1$
10	11		$\langle \blacktriangledown^{[21]} \rangle$	$\langle \blacktriangledown^{[29]} \rangle$	${}^{[12]}_{9} = 1, {}^{[13]}_{9} = 1, {}^{[14]}_{9} = 1, {}^{[16]}_{9} = 0, {}^{[19]}_{8} = {}^{[19]}_{7}, {}^{[20]}_{8} = {}^{[20]}_{7}, {}^{[21]}_{8} = 1, {}^{[21]}_{7} = 0, {}^{[22]}_{8} = {}^{[22]}_{7}, {}^{[25]}_{8} = {}^{[25]}_{7}$
11	19			$\langle \blacktriangle^{[31]} \rangle$	${}^{[16]}_{10} = 1, {}^{[19]}_{10} = 0, {}^{[20]}_{10} = 0, {}^{[21]}_{10} = 0, {}^{[22]}_{10} = 0, {}^{[25]}_{10} = 0, {}^{[29]}_{9} = {}^{[29]}_{8}$
12	3	$\langle \blacktriangledown^{[16]} \rangle$	$\langle \blacktriangle^{[19]}, \blacktriangle^{[22]} \rangle$	$\langle \blacktriangle^{[22]}, \blacktriangle^{[25]} \rangle$	${}^{[19]}_{11} = 0, {}^{[20]}_{11} = 1, {}^{[21]}_{11} = 1, {}^{[22]}_{11} = 0, {}^{[25]}_{11} = 1, {}^{[29]}_{11} = 0, {}^{[31]}_{10} = {}^{[31]}_{9}$
13	7		$\langle \blacktriangle^{[25]} \rangle$	$\langle \blacktriangledown^{[26]}, \blacktriangledown\blacktriangle^{[28,29]} \rangle$	${}^{[22]}_{11} = {}^{[22]}_{10}, {}^{[25]}_{10} = 0, {}^{[25]}_{11} = 1, {}^{[29]}_{12} = 1, {}^{[31]}_{12} = 0$
14	11		$\langle \blacktriangle^{[29]} \rangle$		${}^{[22]}_{13} = 0, {}^{[25]}_{13} = 0, {}^{[26]}_{12} = {}^{[26]}_{11}, {}^{[28]}_{12} = {}^{[28]}_{11}, {}^{[29]}_{11} = 0, {}^{[29]}_{12} = 1, {}^{[31]}_{13} = 1$
15	19			$\langle \blacktriangle^{[18]} \rangle$	${}^{[22]}_{14} = 1, {}^{[25]}_{14} = 1, {}^{[26]}_{14} = 0, {}^{[28]}_{14} = 0, {}^{[29]}_{14} = 0$
16	3		$\langle \blacktriangledown^{[26]}, \blacktriangledown^{[28]} \rangle$	$\langle \blacktriangledown\blacktriangle^{[25,26]}, \blacktriangledown^{[28]}, \blacktriangledown^{[31]} \rangle$	${}^{[18]}_{14} = {}^{[18]}_{13}, {}^{[26]}_{15} \neq {}^{[26]}_{14}, {}^{[28]}_{15} \neq {}^{[28]}_{14}, {}^{[29]}_{15} = {}^{[29]}_{14}$
17	5		$\langle \blacktriangle^{[26]}, \blacktriangledown^{[28]} \rangle$		${}^{[18]}_{16} = {}^{[18]}_{14}, {}^{[25]}_{15} = {}^{[25]}_{14}, {}^{[26]}_{15} \neq {}^{[26]}_{14}, {}^{[28]}_{15} \neq {}^{[28]}_{14}, {}^{[31]}_{15} = {}^{[31]}_{14}$
18	9				${}^{[18]}_{17} = {}^{[18]}_{16}, {}^{[25]}_{17} = {}^{[25]}_{15}, {}^{[26]}_{17} = {}^{[26]}_{15}, {}^{[28]}_{17} = {}^{[28]}_{15}, {}^{[31]}_{17} = {}^{[31]}_{15}$
19	13	$\langle \blacktriangledown^{[16]} \rangle$		$\langle \blacktriangledown^{[29]}, \blacktriangle^{[31]} \rangle$	${}^{[25]}_{18} = {}^{[25]}_{17}, {}^{[26]}_{18} = {}^{[26]}_{17}, {}^{[28]}_{18} = {}^{[28]}_{17}, {}^{[31]}_{18} = {}^{[31]}_{17}$
20	3	$\langle \blacktriangle^{[31]} \rangle$		$\langle \blacktriangledown\blacktriangle^{[28,29]}, \blacktriangledown^{[31]} \rangle$	${}^{[29]}_{18} = {}^{[29]}_{17}, {}^{[31]}_{18} = {}^{[31]}_{17}$
21	5				${}^{[28]}_{19} = {}^{[28]}_{18}$
22	9				${}^{[28]}_{21} = {}^{[28]}_{19}$
23	13		$\langle \blacktriangle^{[29]}, \blacktriangledown^{[31]} \rangle$		${}^{[28]}_{22} = {}^{[28]}_{21}, {}^{[29]}_{22} \neq {}^{[29]}_{21}, {}^{[31]}_{22} \neq {}^{[31]}_{21}$
24	3	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$			
25	5				
26	9				
27	13				
28	3				
29	5				
30	9				
31	13				
32	3				
33	9				
34	11				
35	15	$\langle \blacktriangledown^{[16]} \rangle$		$\langle \blacktriangledown^{[31]} \rangle$	
36	3	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$	$\langle \blacktriangledown^{[31]} \rangle$	$\langle \blacktriangledown^{[31]} \rangle$	
37	9				
38	11				
39	15		$\langle \blacktriangle^{[31]} \rangle$		
40	3	$\langle \blacktriangle^{[31]} \rangle$			
					123 conditions: $96 + 25 + 2$

Path 2 First of the two best paths we found with the same message difference as [WLF+05].

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	Φ-conditions and $\lll$-conditions
0	3				
1	7	⟨▲$^{[31]}$⟩		⟨▼$^{[6]}$⟩	
2	11	⟨▼$^{[28]}$, ▲$^{[31]}$⟩		⟨▲▲▲$^{[7\ldots9]}$⟩	${}^{[6]}_{0} = {}^{[6]}_{-1}$
3	19				${}^{[6]}_{2} = 0$, ${}^{[7]}_{1} = {}^{[7]}_{0}$, ${}^{[8]}_{1} = {}^{[8]}_{0}$, ${}^{[9]}_{1} = {}^{[9]}_{0}$
4	3		⟨▼$^{[6]}$⟩	⟨▲▼$^{[9,10]}$⟩	${}^{[6]}_{3} = 0$, ${}^{[7]}_{3} = 0$, ${}^{[8]}_{3} = 0$, ${}^{[9]}_{3} = 0$
5	7		⟨▲$^{[7]}$⟩	⟨▲$^{[13]}$⟩	${}^{[7]}_{4} = 0$, ${}^{[8]}_{4} = 1$, ${}^{[9]}_{3} = 0$, ${}^{[10]}_{3} = {}^{[10]}_{2}$
6	11		⟨▼$^{[10]}$⟩	⟨▼$^{[18]}$⟩	${}^{[9]}_{5} = 0$, ${}^{[10]}_{5} = 1$, ${}^{[13]}_{4} = {}^{[13]}_{3}$
7	19				${}^{[9]}_{6} = 1$, ${}^{[10]}_{6} = 1$, ${}^{[13]}_{6} = 0$, ${}^{[18]}_{5} = {}^{[18]}_{4}$
8	3		⟨▲$^{[13]}$⟩	⟨▼$^{[12]}$, ▼▲$^{[16,17]}$⟩	${}^{[13]}_{7} = 0$, ${}^{[18]}_{7} = 0$
9	7		⟨▼$^{[12]}$⟩	⟨▲$^{[19]}$⟩	${}^{[12]}_{7} = 1$, ${}^{[12]}_{6} = 0$, ${}^{[16]}_{7} = {}^{[16]}_{6}$, ${}^{[17]}_{7} = {}^{[17]}_{6}$, ${}^{[18]}_{8} = 1$
10	11		⟨▲$^{[17]}$⟩	⟨▼$^{[28]}$⟩	${}^{[12]}_{9} = 0$, ${}^{[16]}_{9} = 0$, ${}^{[17]}_{9} = 1$, ${}^{[19]}_{8} = {}^{[19]}_{7}$
11	19				${}^{[12]}_{10} = 1$, ${}^{[16]}_{10} = 1$, ${}^{[17]}_{10} = 1$, ${}^{[19]}_{10} = 0$, ${}^{[28]}_{9} = {}^{[28]}_{8}$
12	3	⟨▼$^{[16]}$⟩	⟨▲$^{[19]}$⟩	⟨▼$^{[15]}$, ▲$^{[22]}$⟩	${}^{[19]}_{11} = 0$, ${}^{[28]}_{11} = 0$
13	7			⟨▼▼▲$^{[26\ldots28]}$⟩	${}^{[15]}_{11} = {}^{[15]}_{10}$, ${}^{[22]}_{11} = {}^{[22]}_{10}$, ${}^{[28]}_{12} = 1$
14	11		⟨▲$^{[28]}$⟩		${}^{[15]}_{13} = 0$, ${}^{[22]}_{13} = 0$, ${}^{[26]}_{12} = {}^{[26]}_{11}$, ${}^{[27]}_{12} = {}^{[27]}_{11}$, ${}^{[28]}_{12} = 1$, ${}^{[28]}_{11} = 0$
15	19		⟨▲$^{[28]}$⟩	⟨▲$^{[15]}$⟩	${}^{[15]}_{14} = 1$, ${}^{[22]}_{14} = 1$, ${}^{[26]}_{14} = 0$, ${}^{[27]}_{14} = 0$, ${}^{[28]}_{14} = 1$
16	3		⟨▲$^{[15]}$⟩	⟨▲$^{[25]}$⟩	${}^{[15]}_{14} \neq {}^{[15]}_{13}$, ${}^{[26]}_{15} = {}^{[26]}_{14}$, ${}^{[27]}_{15} = {}^{[27]}_{14}$, ${}^{[28]}_{15} = {}^{[28]}_{14}$
17	5			⟨▲$^{[31]}$⟩	${}^{[15]}_{16} = {}^{[15]}_{14}$, ${}^{[25]}_{15} = {}^{[25]}_{14}$
18	9				${}^{[15]}_{17} = {}^{[15]}_{16}$, ${}^{[25]}_{17} = {}^{[25]}_{15}$, ${}^{[31]}_{16} = {}^{[31]}_{15}$
19	13	⟨▼$^{[16]}$⟩		⟨▼$^{[28]}$⟩	${}^{[25]}_{18} = {}^{[25]}_{17}$, ${}^{[31]}_{18} = {}^{[31]}_{16}$
20	3	⟨▲$^{[31]}$⟩	⟨▼$^{[28]}$, ▲$^{[31]}$⟩	⟨▲$^{[28]}$, ▼$^{[31]}$⟩	${}^{[28]}_{18} \neq {}^{[28]}_{17}$, ${}^{[31]}_{19} \neq {}^{[31]}_{18}$
21	5		⟨▼$^{[31]}$⟩		${}^{[31]}_{19} \neq {}^{[31]}_{18}$
22	9				${}^{[31]}_{21} = {}^{[31]}_{19}$
23	13		⟨▲$^{[28]}$⟩		${}^{[28]}_{22} \neq {}^{[28]}_{21}$, ${}^{[31]}_{22} = {}^{[31]}_{21}$
24	3	⟨▼$^{[28]}$, ▲$^{[31]}$⟩			
25	5				
26	9				
27	13				
28	3				
29	5				
30	9				
31	13				
32	3				
33	9				
34	11				
35	15	⟨▼$^{[16]}$⟩		⟨▼$^{[31]}$⟩	
36	3	⟨▼$^{[28]}$, ▲$^{[31]}$⟩	⟨▼$^{[31]}$⟩	⟨▼$^{[31]}$⟩	
37	9				
38	11				
39	15		⟨▲$^{[31]}$⟩		
40	3	⟨▲$^{[31]}$⟩			

90 conditions: $72 + 16 + 2$

Path 3 Second of the two best paths found with the same message difference as [WLF+05].

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	Φ-conditions and $\lll$-conditions
0	3				
1	7	$\langle \blacktriangle^{[31]} \rangle$		$\langle \blacktriangle^{[6]} \rangle$	
2	11	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$		$\langle \blacktriangledown^{[7]}, \blacktriangle^{[10]} \rangle$	${}^{[6]}_{0} = {}^{[6]}_{-1}$
3	19				${}^{[6]}_{2} = 0,\ {}^{[7]}_{1} = {}^{[7]}_{0},\ {}^{[10]}_{1} = {}^{[10]}_{0}$
4	3		$\langle \blacktriangle\blacktriangledown^{[6,7]} \rangle$	$\langle \blacktriangle\blacktriangle\blacktriangledown^{[9\ldots11]} \rangle$	${}^{[6]}_{3} = 0,\ {}^{[7]}_{3} = 1,\ {}^{[10]}_{3} = 0$
5	7			$\langle \blacktriangle^{[13]} \rangle$	${}^{[7]}_{4} = 1,\ {}^{[9]}_{3} = {}^{[9]}_{2},\ {}^{[10]}_{3} = 0,\ {}^{[11]}_{3} = {}^{[11]}_{2}$
6	11		$\langle \blacktriangle\blacktriangledown^{[10,11]} \rangle$	$\langle \blacktriangledown^{[18]} \rangle$	${}^{[9]}_{5} = 0,\ {}^{[10]}_{5} = 1,\ {}^{[11]}_{5} = 1,\ {}^{[13]}_{4} = {}^{[13]}_{3}$
7	19				${}^{[9]}_{6} = 1,\ {}^{[10]}_{6} = 1,\ {}^{[11]}_{6} = 1,\ {}^{[13]}_{6} = 0,\ {}^{[18]}_{5} = {}^{[18]}_{4}$
8	3		$\langle \blacktriangle^{[13]} \rangle$	$\langle \blacktriangledown^{[12]}, \blacktriangle^{[16]} \rangle$	${}^{[13]}_{7} = 0,\ {}^{[18]}_{7} = 0$
9	7		$\langle \blacktriangledown^{[12]} \rangle$	$\langle \blacktriangle^{[19]} \rangle$	${}^{[12]}_{7} = 1,\ {}^{[12]}_{6} = 0,\ {}^{[16]}_{7} = {}^{[16]}_{6},\ {}^{[18]}_{8} = 1$
10	11			$\langle \blacktriangledown^{[29]} \rangle$	${}^{[12]}_{9} = 0,\ {}^{[16]}_{9} = 0,\ {}^{[19]}_{8} = {}^{[19]}_{7}$
11	19				${}^{[12]}_{10} = 1,\ {}^{[16]}_{10} = 1,\ {}^{[19]}_{10} = 0,\ {}^{[29]}_{9} = {}^{[29]}_{8}$
12	3	$\langle \blacktriangledown^{[16]} \rangle$	$\langle \blacktriangle^{[19]} \rangle$	$\langle \blacktriangledown^{[15]}, \blacktriangle^{[22]} \rangle$	${}^{[19]}_{11} = 0,\ {}^{[29]}_{11} = 0$
13	7			$\langle \blacktriangledown\blacktriangledown\blacktriangledown\blacktriangle^{[26\ldots29]} \rangle$	${}^{[15]}_{11} = {}^{[15]}_{10},\ {}^{[22]}_{11} = {}^{[22]}_{10},\ {}^{[29]}_{12} = 1$
14	11		$\langle \blacktriangle^{[29]} \rangle$		${}^{[15]}_{13} = 0,\ {}^{[22]}_{13} = 0,\ {}^{[26]}_{12} = {}^{[26]}_{11},\ {}^{[27]}_{12} = {}^{[27]}_{11},\ {}^{[28]}_{12} = {}^{[28]}_{11},\ {}^{[29]}_{12} = 1,\ {}^{[29]}_{11} = 0$
15	19		$\langle \blacktriangledown\blacktriangle^{[28,29]} \rangle$	$\langle \blacktriangle^{[15]} \rangle$	${}^{[15]}_{14} = 1,\ {}^{[22]}_{14} = 1,\ {}^{[26]}_{14} = 0,\ {}^{[27]}_{14} = 0,\ {}^{[28]}_{14} = 1,\ {}^{[29]}_{14} = 1$
16	3		$\langle \blacktriangle^{[15]} \rangle$	$\langle \blacktriangle^{[25]} \rangle$	${}^{[15]}_{14} \neq {}^{[15]}_{13},\ {}^{[26]}_{15} = {}^{[26]}_{14},\ {}^{[27]}_{15} = {}^{[27]}_{14},\ {}^{[28]}_{15} = {}^{[28]}_{14},\ {}^{[29]}_{15} = {}^{[29]}_{14}$
17	5			$\langle \blacktriangle^{[31]} \rangle$	${}^{[15]}_{16} = {}^{[15]}_{14},\ {}^{[25]}_{15} = {}^{[25]}_{14}$
18	9				${}^{[15]}_{17} = {}^{[15]}_{16},\ {}^{[25]}_{17} = {}^{[25]}_{15},\ {}^{[31]}_{16} = {}^{[31]}_{15}$
19	13	$\langle \blacktriangledown^{[16]} \rangle$		$\langle \blacktriangledown^{[28]} \rangle$	${}^{[25]}_{18} = {}^{[25]}_{17},\ {}^{[31]}_{18} = {}^{[31]}_{16}$
20	3	$\langle \blacktriangle^{[31]} \rangle$	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$	$\langle \blacktriangle^{[28]}, \blacktriangledown^{[31]} \rangle$	${}^{[28]}_{18} \neq {}^{[28]}_{17},\ {}^{[31]}_{19} \neq {}^{[31]}_{18}$
21	5		$\langle \blacktriangledown^{[31]} \rangle$		${}^{[31]}_{19} \neq {}^{[31]}_{18}$
22	9				${}^{[31]}_{21} = {}^{[31]}_{19}$
23	13		$\langle \blacktriangle^{[28]} \rangle$		${}^{[28]}_{22} \neq {}^{[28]}_{21},\ {}^{[31]}_{22} = {}^{[31]}_{21}$
24	3	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$			
25	5				
26	9				
27	13				
28	3				
29	5				
30	9				
31	13				
32	3				
33	9				
34	11				
35	15	$\langle \blacktriangledown^{[16]} \rangle$		$\langle \blacktriangledown^{[31]} \rangle$	
36	3	$\langle \blacktriangledown^{[28]}, \blacktriangle^{[31]} \rangle$	$\langle \blacktriangledown^{[31]} \rangle$	$\langle \blacktriangledown^{[31]} \rangle$	
37	9				
38	11				
39	15		$\langle \blacktriangle^{[31]} \rangle$		
40	3	$\langle \blacktriangle^{[31]} \rangle$			

90 conditions: $72 + 16 + 2$

Path 4 Improved version of the path from Yu *et al.* [YWZW05].

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	conditions
0	3				
1	7				
2	11				
3	19				
4	3	$\langle \blacktriangle^{[25]} \rangle$		$\langle \blacktriangle^{[28]} \rangle$	
5	7				${}^{[28]}_{3} = {}^{[28]}_{2}$
6	11				${}^{[28]}_{5} = 0$
7	19				${}^{[28]}_{6} = 1$
8	3			$\langle \blacktriangle^{[31]} \rangle$	
9	7				${}^{[31]}_{7} = {}^{[31]}_{6}$
10	11		$\langle \blacktriangle^{[31]} \rangle$	$\langle \blacktriangledown^{[10]} \rangle$	${}^{[31]}_{9} = 1$
11	19				${}^{[10]}_{9} = {}^{[10]}_{8}$, ${}^{[31]}_{10} = 1$
12	3			$\langle \blacktriangle^{[2]} \rangle$	${}^{[10]}_{11} = 0$
13	7				${}^{[2]}_{11} = {}^{[2]}_{10}$, ${}^{[10]}_{12} = 1$
14	11			$\langle \blacktriangledown^{[21]} \rangle$	${}^{[2]}_{13} = 0$
15	19				${}^{[2]}_{14} = 1$, ${}^{[21]}_{13} = {}^{[21]}_{12}$
16	3			$\langle \blacktriangle^{[5]} \rangle$	${}^{[21]}_{15} = {}^{[21]}_{13}$
17	5	$\langle \blacktriangle^{[25]} \rangle$	$\langle \blacktriangle^{[5]} \rangle$	$\langle \blacktriangle^{[10]}, \blacktriangle^{[30]} \rangle$	${}^{[5]}_{15} \neq {}^{[5]}_{14}$, ${}^{[21]}_{16} = {}^{[21]}_{15}$
18	9			$\langle \blacktriangledown^{[30]} \rangle$	${}^{[5]}_{17} = {}^{[5]}_{15}$, ${}^{[10]}_{16} = {}^{[10]}_{15}$, ${}^{[30]}_{16} = {}^{[30]}_{15}$
19	13				${}^{[5]}_{18} = {}^{[5]}_{17}$, ${}^{[10]}_{18} = {}^{[10]}_{16}$
20	3			$\langle \blacktriangle^{[8]} \rangle$	${}^{[10]}_{19} = {}^{[10]}_{18}$
21	5		$\langle \blacktriangledown^{[30]} \rangle$	$\langle \blacktriangle^{[15]} \rangle$	${}^{[8]}_{19} = {}^{[8]}_{18}$, ${}^{[30]}_{20} \neq {}^{[30]}_{19}$
22	9			$\langle \blacktriangle\blacktriangledown^{[7,8]} \rangle$	${}^{[8]}_{21} = {}^{[8]}_{19}$, ${}^{[15]}_{20} = {}^{[15]}_{19}$
23	13				${}^{[7]}_{21} = {}^{[7]}_{20}$, ${}^{[15]}_{22} = {}^{[15]}_{20}$
24	3		$\langle \blacktriangledown^{[8]} \rangle$		${}^{[7]}_{23} = {}^{[7]}_{21}$, ${}^{[8]}_{23} \neq {}^{[8]}_{21}$, ${}^{[15]}_{23} = {}^{[15]}_{22}$
25	5			$\langle \blacktriangle^{[20]} \rangle$	${}^{[7]}_{24} = {}^{[7]}_{23}$, ${}^{[8]}_{24} = {}^{[8]}_{23}$
26	9			$\langle \blacktriangledown^{[16]} \rangle$	${}^{[20]}_{24} = {}^{[20]}_{23}$
27	13				${}^{[16]}_{25} = {}^{[16]}_{24}$, ${}^{[20]}_{26} = {}^{[20]}_{24}$
28	3				${}^{[16]}_{27} = {}^{[16]}_{25}$, ${}^{[20]}_{27} = {}^{[20]}_{26}$
29	5			$\langle \blacktriangle^{[25]} \rangle$	${}^{[16]}_{28} = {}^{[16]}_{27}$
30	9			$\langle \blacktriangledown^{[25]} \rangle$	${}^{[25]}_{28} = {}^{[25]}_{27}$
31	13				
32	3				
33	9		$\langle \blacktriangledown^{[25]} \rangle$		${}^{[25]}_{32} = {}^{[25]}_{31}$
34	11	$\langle \blacktriangle^{[25]} \rangle$			

58 conditions: $20 + 37 + 1$

Path 5 An IV-dependent path with the message difference on the first word.

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	Φ-conditions and $\lll$-conditions
0	3	$\langle \blacktriangle^{[0]} \rangle$		$\langle \blacktriangle^{[3]} \rangle$	
1	7				${}^{[3]}_{-1} = {}^{[3]}_{-2}$
2	11				${}^{[3]}_{1} = 0$
3	19				${}^{[3]}_{2} = 1$
4	3			$\langle \blacktriangledown\blacktriangle^{[6,7]} \rangle$	
5	7				${}^{[6]}_{3} = {}^{[6]}_{2},\ {}^{[7]}_{3} = {}^{[7]}_{2}$
6	11				${}^{[6]}_{5} = 0,\ {}^{[7]}_{5} = 0$
7	19		$\langle \blacktriangle^{[7]} \rangle$	$\langle \blacktriangle^{[26]} \rangle$	${}^{[6]}_{6} = 1,\ {}^{[7]}_{6} = 0$
8	3		$\langle \blacktriangledown^{[26]} \rangle$	$\langle \blacktriangle^{[9]}, \blacktriangledown^{[29]} \rangle$	${}^{[26]}_{5} = 1,\ {}^{[26]}_{6} = 0$
9	7				${}^{[9]}_{7} = {}^{[9]}_{6},\ {}^{[26]}_{8} = 0,\ {}^{[29]}_{7} = {}^{[29]}_{6}$
10	11				${}^{[9]}_{9} = 0,\ {}^{[26]}_{9} = 1,\ {}^{[29]}_{9} = 0$
11	19			$\langle \blacktriangle^{[13]} \rangle$	${}^{[9]}_{10} = 1,\ {}^{[29]}_{10} = 1$
12	3			$\langle \blacktriangledown^{[0]}, \blacktriangle^{[12]} \rangle$	${}^{[13]}_{10} = {}^{[13]}_{9}$
13	7				${}^{[0]}_{11} = {}^{[0]}_{10},\ {}^{[12]}_{11} = {}^{[12]}_{10},\ {}^{[13]}_{12} = 0$
14	11		$\langle \blacktriangledown^{[0]} \rangle$	$\langle \blacktriangle\blacktriangle\blacktriangledown^{[11\ldots13]} \rangle$	${}^{[0]}_{13} = 1,\ {}^{[12]}_{13} = 0,\ {}^{[13]}_{13} = 1$
15	19		$\langle \blacktriangledown^{[13]} \rangle$		${}^{[0]}_{14} = 1,\ {}^{[11]}_{13} = {}^{[11]}_{12},\ {}^{[12]}_{13} = 0,\ {}^{[13]}_{13} = 1,\ {}^{[13]}_{12} = 0$
16	3	$\langle \blacktriangle^{[0]} \rangle$	$\langle \blacktriangle\blacktriangledown^{[12,13]} \rangle$		${}^{[11]}_{15} = {}^{[11]}_{13},\ {}^{[12]}_{15} \neq {}^{[12]}_{13},\ {}^{[13]}_{15} \neq {}^{[13]}_{13}$
17	5				${}^{[11]}_{16} = {}^{[11]}_{15},\ {}^{[12]}_{16} = {}^{[12]}_{15},\ {}^{[13]}_{16} = {}^{[13]}_{15}$
18	9			$\langle \blacktriangle\blacktriangle\blacktriangle\blacktriangledown^{[20\ldots23]} \rangle$	
19	13				${}^{[20]}_{17} = {}^{[20]}_{16},\ {}^{[21]}_{17} = {}^{[21]}_{16},\ {}^{[22]}_{17} = {}^{[22]}_{16},\ {}^{[23]}_{17} = {}^{[23]}_{16}$
20	3		$\langle \blacktriangledown^{[23]} \rangle$	$\langle \blacktriangledown^{[26]} \rangle$	${}^{[20]}_{19} = {}^{[20]}_{17},\ {}^{[21]}_{19} = {}^{[21]}_{17},\ {}^{[22]}_{19} = {}^{[22]}_{17},\ {}^{[23]}_{19} \neq {}^{[23]}_{17}$
21	5				${}^{[20]}_{20} = {}^{[20]}_{19},\ {}^{[21]}_{20} = {}^{[21]}_{19},\ {}^{[22]}_{20} = {}^{[22]}_{19},\ {}^{[23]}_{20} = {}^{[23]}_{19},\ {}^{[26]}_{19} = {}^{[26]}_{18}$
22	9			$\langle \blacktriangledown^{[29]} \rangle$	${}^{[26]}_{21} = {}^{[26]}_{19}$
23	13				${}^{[26]}_{22} = {}^{[26]}_{21},\ {}^{[29]}_{21} = {}^{[29]}_{20}$
24	3			$\langle \blacktriangle\blacktriangledown^{[29,30]} \rangle$	${}^{[29]}_{23} = {}^{[29]}_{21}$
25	5				${}^{[30]}_{23} = {}^{[30]}_{22}$
26	9		$\langle \blacktriangle^{[29]} \rangle$		${}^{[29]}_{25} \neq {}^{[29]}_{23},\ {}^{[30]}_{25} = {}^{[30]}_{23}$
27	13				${}^{[29]}_{26} = {}^{[29]}_{25},\ {}^{[30]}_{26} = {}^{[30]}_{25}$
28	3			$\langle \blacktriangledown^{[0]} \rangle$	
29	5				${}^{[0]}_{27} = {}^{[0]}_{26}$
30	9				${}^{[0]}_{29} = {}^{[0]}_{27}$
31	13				${}^{[0]}_{30} = {}^{[0]}_{29}$
32	3	$\langle \blacktriangle^{[0]} \rangle$			
					80 conditions: (1+) $42 + 37 + 0$

Path 6 Improved HMAC path for [WOK08]. This path can be used on 20 different bit positions (it fails on bits 6 to 11 and 17 to 22). When used on bit 12, the path can be modified by changing the sign of the difference in Q_7 and Q_{11}, allowing for a shorter carry chain in Q_{14}, and only 57 conditions.

i	s_i	δm_i	$\partial\Phi_i$	∂Q_i	Φ-conditions and $\lll$-conditions
0	3				
1	7				
2	11				
3	19	$\langle \blacktriangle^{[3]} \rangle$		$\langle \blacktriangle^{[22]} \rangle$	
4	3				${}^{[22]}_{2} = {}^{[22]}_{1}$
5	7				${}^{[22]}_{4} = 0$
6	11		$\langle \blacktriangle^{[22]} \rangle$	$\langle \blacktriangle^{[1]} \rangle$	${}^{[22]}_{5} = 0$
7	19			$\langle \blacktriangle^{[9]} \rangle$	${}^{[1]}_{5} = {}^{[1]}_{4}$
8	3				${}^{[1]}_{7} = 0,\ {}^{[9]}_{6} = {}^{[9]}_{5}$
9	7				${}^{[1]}_{8} = 1,\ {}^{[9]}_{8} = 0$
10	11			$\langle \blacktriangle^{[12]} \rangle$	${}^{[9]}_{9} = 1$
11	19			$\langle \blacktriangle^{[28]} \rangle$	${}^{[12]}_{9} = {}^{[12]}_{8}$
12	3				${}^{[12]}_{11} = 0,\ {}^{[28]}_{10} = {}^{[28]}_{9}$
13	7				${}^{[12]}_{12} = 1,\ {}^{[28]}_{12} = 0$
14	11			$\langle \blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown\blacktriangledown\blacktriangle^{[23\ldots29]} \rangle$	${}^{[28]}_{13} = 1$
15	19		$\langle \blacktriangledown^{[28]} \rangle$		${}^{[23]}_{13} = {}^{[23]}_{12},\ {}^{[24]}_{13} = {}^{[24]}_{12},\ {}^{[25]}_{13} = {}^{[25]}_{12},\ {}^{[26]}_{13} = {}^{[26]}_{12},\ {}^{[27]}_{13} = {}^{[27]}_{12},\ {}^{[28]}_{13} = 1,\ {}^{[28]}_{12} = 0,\ {}^{[29]}_{13} = {}^{[29]}_{12}$
16	3		$\langle \blacktriangledown^{[26]} \rangle$	$\langle \blacktriangledown^{[29]} \rangle$	${}^{[23]}_{15} = {}^{[23]}_{13},\ {}^{[24]}_{15} = {}^{[24]}_{13},\ {}^{[25]}_{15} = {}^{[25]}_{13},\ {}^{[26]}_{15} \neq {}^{[26]}_{13},\ {}^{[27]}_{15} = {}^{[27]}_{13},\ {}^{[28]}_{15} = {}^{[28]}_{13},\ {}^{[29]}_{15} = {}^{[29]}_{13}$
17	5				${}^{[23]}_{16} = {}^{[23]}_{15},\ {}^{[24]}_{16} = {}^{[24]}_{15},\ {}^{[25]}_{16} = {}^{[25]}_{15},\ {}^{[26]}_{16} = {}^{[26]}_{15},\ {}^{[27]}_{16} = {}^{[27]}_{15},\ {}^{[28]}_{16} = {}^{[28]}_{15}$
18	9			$\langle \blacktriangle^{[0]} \rangle$	${}^{[29]}_{17} = {}^{[29]}_{15}$
19	13				${}^{[0]}_{17} = {}^{[0]}_{16},\ {}^{[29]}_{18} = {}^{[29]}_{17}$
20	3			$\langle \blacktriangledown^{[0]} \rangle$	${}^{[0]}_{19} = {}^{[0]}_{17}$
21	5				
22	9		$\langle \blacktriangledown^{[0]} \rangle$		${}^{[0]}_{21} \neq {}^{[0]}_{19}$
23	13				${}^{[0]}_{22} = {}^{[0]}_{21}$
24	3			$\langle \blacktriangledown^{[3]} \rangle$	
25	5				${}^{[3]}_{23} = {}^{[3]}_{22}$
26	9				${}^{[3]}_{25} = {}^{[3]}_{23}$
27	13				${}^{[3]}_{26} = {}^{[3]}_{25}$
28	3	$\langle \blacktriangle^{[3]} \rangle$			
29	5				
30	9				
31	13				

59 conditions: 59 conditions: $40 + 19$

Table 3.8 Example run on the path from [WLF$^+$05]. We show intermediate steps to ease comprehension, but the algorithm actually finds the final path directly from the original one.

			Initial path		Path 1		Path 2		Path 3		Final Path	
step	s_i	δm_i	$\partial\Phi_i$	∂Q_i	$\partial\Phi_i$	∂Q_i	$\partial\Phi_i$	∂Q_i	$\partial\Phi_i$	∂Q_i	$\partial\Phi_i$	∂Q_i
4	3	⟨▲[22]⟩	⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩			⟨▲[25]⟩		⟨▲[25]⟩
5	7		⟨▼[1] ▲[13]⟩	⟨▼[8], ▲[20]⟩	⟨▲[13]⟩	⟨▲[20]⟩	⟨▲[13]⟩	⟨▲[20]⟩	⟨▲[13]⟩	⟨▲[20]⟩		
6	11		⟨▼[17]⟩	⟨▼[28]⟩	⟨▼[17]⟩	⟨▼[28]⟩						
7	19											
8	3									⟨▼▲[28,29]⟩		⟨▼▲[28,29]⟩
9	7			⟨▼[15], ▲[27]⟩		⟨▲[27]⟩		⟨▼▲[27,28]⟩		⟨▲[27]⟩		
10	11			⟨▼[7]⟩		⟨▼[7]⟩	⟨▼[28]⟩	⟨▼[7]⟩	⟨▼[28]⟩	⟨▼[7]⟩	⟨▼[28]⟩	⟨▼[7]⟩
11	19											
12	3									⟨▲[31]⟩		⟨▲[31]⟩
13	7			⟨▲[2], ▼[22]⟩		⟨▲[2]⟩		⟨▲[2]⟩		⟨▲[2]⟩		
14	11			⟨▼[18]⟩		⟨▼[18]⟩		⟨▼[18]⟩		⟨▼[18]⟩		⟨▼[18]⟩
15	19											
16	3									⟨▲[2]⟩		⟨▲[2]⟩
17	5	⟨▲[22]⟩		⟨▲[7]⟩		⟨▲[7], ▲[27]⟩		⟨▲[7], ▲[27]⟩		⟨▲[7], ▲[27]⟩	⟨▲[2]⟩	⟨▲[7], ▲[27]⟩
18	9			⟨▼[27]⟩		⟨▼[27]⟩		⟨▼[27]⟩		⟨▼[27]⟩		⟨▼[27]⟩
19	13											
20	3									⟨▲[5]⟩		⟨▲[5]⟩
21	5			⟨▲[12]⟩	⟨▼[27]⟩	⟨▲[12]⟩	⟨▼[27]⟩	⟨▲[12]⟩	⟨▼[27]⟩	⟨▲[12]⟩	⟨▼[27]⟩	⟨▲[12]⟩
22	9			⟨▼[4]⟩		⟨▼[4]⟩		⟨▼[4]⟩		⟨▲▼[4,5]⟩		⟨▲▼[4,5]⟩
23	13											
24	3								⟨▼[5]⟩		⟨▼[5]⟩	
25	5			⟨▲[17]⟩		⟨▲[17]⟩		⟨▲[17]⟩		⟨▲[17]⟩		⟨▲[17]⟩
26	9			⟨▼[13]⟩		⟨▼[13]⟩		⟨▼[13]⟩		⟨▼[13]⟩		⟨▼[13]⟩
27	13											
28	3											
29	5			⟨▲[22]⟩		⟨▲[22]⟩		⟨▲[22]⟩		⟨▲[22]⟩		⟨▲[22]⟩
30	9			⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩
31	13											
32	3											
33	9		⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩		⟨▼[22]⟩	
34	11	⟨▲[22]⟩										

3.E Conditions for Φ

Table 3.9 Conditions to get the right output difference in Φ_i.
✓: no extra conditions are needed; ✗: input and output requirements are incompatible.

			$F(x,y,z)=\mathsf{IF}(x,y,z)$			$G(x,y,z)=\mathsf{MAJ}(x,y,z)$			$H(x,y,z)=x\oplus y\oplus z$			$I(x,y,z)=y\oplus(x\vee\neg z)$		
∂x	∂y	∂z	$\partial F=0$	$\partial F=1$	$\partial F=-1$	$\partial G=0$	$\partial G=1$	$\partial G=-1$	$\partial H=0$	$\partial H=1$	$\partial H=-1$	$\partial I=0$	$\partial I=1$	$\partial I=-1$
0	0	0	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗
0	0	+1	$x=1$	$x=0$	✗	$x=y$	$x\neq y$	✗	✗	$x=y$	$x\neq y$	$x=1$	$x,y=0,1$	$x,y=0,0$
0	0	−1	$x=1$	✗	$x=0$	$x=y$	✗	$x\neq y$	✗	$x\neq y$	$x=y$	$x=1$	$x,y=0,0$	$x,y=0,1$
0	+1	0	$x=0$	$x=1$	✗	$x=z$	$x\neq z$	✗	✗	$x=z$	$x\neq z$	✗	$x,z=0,1$	$x,y\neq 1,0$
0	−1	0	$x=0$	✗	$x=1$	$x=z$	✗	$x\neq z$	✗	$x\neq z$	$x=z$	✗	$x,y\neq 1,0$	$x,z=0,1$
+1	0	0	$y=z$	$y,z=1,0$	$y,z=0,1$	$y=z$	$y\neq z$	✗	✗	$y=z$	$y\neq z$	$z=0$	$y,z=0,1$	$y,z=1,1$
−1	0	0	$y=z$	$y,z=0,1$	$y,z=1,0$	$y=z$	✗	$y\neq z$	✗	$y\neq z$	$y=z$	$z=0$	$y,z=1,1$	$y,z=0,1$
0	+1	+1	✗	✓	✗	✗	✓	✗	✓	✗	✗	$x=0$	✗	$x=1$
0	−1	+1	✗	$x=0$	$x=1$	✓	✗	✗	✓	✗	✗	$x=0$	$x=1$	✗
0	+1	−1	✗	$x=1$	$x=0$	✓	✗	✗	✓	✗	✗	$x=0$	✗	$x=1$
0	−1	−1	✗	✗	✓	✗	✗	✓	✓	✗	✗	$x=0$	$x=1$	✗
+1	0	+1	$y=0$	$y=1$	✗	✗	✓	✗	✓	✗	✗	✓	✗	✗
−1	0	+1	$y=1$	$y=0$	✗	✓	✗	✗	✓	✗	✗	✗	$y=1$	$y=0$
+1	0	−1	$y=1$	✗	$y=0$	✓	✗	✗	✓	✗	✗	✗	$y=0$	$y=1$
−1	0	−1	$y=0$	✗	$y=1$	✗	✗	✓	✓	✗	✗	✓	✗	✗
+1	+1	0	$z=1$	$z=0$	✗	✗	✓	✗	✓	✗	✗	$z=1$	✗	$z=0$
−1	+1	0	$z=0$	$z=1$	✗	✓	✗	✗	✓	✗	✗	$z=1$	✗	$z=0$
+1	−1	0	$z=0$	✗	$z=1$	✓	✗	✗	✓	✗	✗	$z=1$	$z=0$	✗
−1	−1	0	$z=1$	✗	$z=0$	✗	✗	✓	✓	✗	✗	$z=1$	$z=0$	✗
+1	+1	+1	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓
−1	+1	+1	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗	✗
+1	−1	+1	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
−1	−1	+1	✓	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗
+1	+1	−1	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗	✗
−1	+1	−1	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓
+1	−1	−1	✗	✗	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗
−1	−1	−1	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗

Chapter 4

Applications of Wang *et al.*'s Attacks

4.1 Introduction

Hash functions are used in many cryptographic constructions under various assumptions, and the practical impact of collision attacks is often unclear. In this chapter we study some consequences of Wang *et al.*'s collision attacks. While most applications of hash function are not threatened by collision attacks by themselves, we show that a collision attack can sometimes be extended into a dedicated attack.

In Section 4.2 to 4.5 we look at HMAC and NMAC. HMAC and NMAC are hash-based message authentication codes proposed by Bellare, Canetti and Krawczyk [BCK96a], which are very interesting to study for at least three reasons: HMAC is standardized (by ANSI, IETF, ISO and NIST) and widely deployed (*e.g.* SSL, TLS, SSH, IPsec); both HMAC and NMAC have security proofs [Bel06, BCK96a]; and both are rather simple constructions. In this section we describe the first universal forgery attack, *without related keys*, on HMAC and NMAC instantiated with a real-life hash function, namely MD4. Our main result is an attack on HMAC/NMAC-MD4 which recovers the full NMAC-MD4 secret key after 2^{88} MAC queries and 2^{95} offline computations; for HMAC, we recover information equivalent to the key. We also describe a full key-recovery attack on NMAC-MD5 in the related-key setting, by extending the attack of Contini and Yin [CY06].

In Section 4.6, we study `APOP`, one the authentication schemes of the Post Office Protocol (POP3). `APOP` is built with MD5, using a simple secret-suffix MAC: the server sends a challenge c to the client, and the client responds with MD5($c\|p$) to prove knowledge of the password p. Our contribution is a partial password-recovery attack against `APOP`, in the man-in-the-middle setting. The attack is based on building collisions that allow to test the password characters one by one, and send the collision pair as challenges. This idea was previously used by Preneel and van Oorschot to attack the envelop MAC [PvO96]. In `APOP`, we are able to recover 3 characters of the password, therefore greatly reducing its entropy. Even though we do not achieve full recovery of the password, the missing bits can be found by exhaustive search. In practice it is sufficient to reduce this search to a reasonable time for small passwords, *i.e.* less than 9 characters. This result is interesting because it is a *practical* attack on a *widely used* protocol.

These results use extensions of the basic attack by Wang *et al.* presented in the previous chapter. Namely the HMAC-MD4 attack uses custom differential paths found with the differential path algorithm of Section 3.5, while the `APOP` attack uses freedom in the colliding blocks as explained in Section 3.4. This shows that collision attacks can be a real threat, and the security of constructions based on MD4, MD5 or SHA-1 should be evaluated with regard to collision attacks. However, in order to build such extensions of a collision attack, one needs to get a good understanding of the inner details of collision attacks.

4.2 Application to HMAC/NMAC-MD4

In this section, we study application of Wang *et al.*'s collision attack to HMAC and NMAC. HMAC is the most widely used construction to build a MAC from a hash function, and NMAC is the theoretical foundation of HMAC.

We present the first universal forgery attack, *without related keys*, on HMAC and NMAC instantiated with a real-life hash function, namely MD4. Our main result is an attack on HMAC/NMAC-MD4 which recovers the full NMAC-MD4 secret key after 2^{88} MAC queries and an offline computation of 2^{95}; for HMAC, we do not recover the HMAC-MD4 secret key k, instead we recover both $H(k \oplus \text{ipad})$ and $H(k \oplus \text{opad})$, which is sufficient to compute any MAC. This attack has later been improved by L. Wang *et al.* in [WOK08] using pseudo-collisions instead of collisions. We also describe a full key-recovery attack on NMAC-MD5 in the related-key setting, by extending the attack of Contini and Yin [CY06]. This improvement was independently proposed in [RR06].

Our attacks on MD4 and MD5 are rather different from each other, although both are based on IV-recovery attacks, which allow to recover the IV when one is given oracle access to a hash function whose IV remains secret. Such IV-recovery attacks can be exploited to attack HMAC and NMAC because the oracle can in fact be very weak: we do not need the full output of the hash function; essentially, we only need to detect if two related messages collide under the hash function. The MD5 related-key attack closely follows the Contini-Yin attack [CY06]: the IV-recovery attack is more or less based on message modification techniques. The MD4 IV-recovery attack is based on a new technique: we use differential paths which depend on a condition in the IV. This idea was simultaneously used by Rechberger and Rijmen in [RR07]: using a path that can only be satisfied for a subset of the keys, we can recover key information by testing whether message pairs follow the path. The advantage of this technique is that it can be used to recover the outer key quite efficiently, since we only need to control the *difference* of the inputs and not the *values* themselves. To make this possible *without related keys*, we need a differential path with a message difference only active in the first input words. We found such IV-dependent paths using the automated tool described in Section 3.5. To make this attack more efficient, we also introduce a method to construct cheaply lots of message pairs with a specific hash difference.

Our results are summarized in the following table, together with previous attacks where "Data" means online queries and "Time" is offline computations:

Attacks		Data	Time	Mem	Remark
Generic	E-Forgery	$2^{n/2}$	-	$2^{n/2}$	[PvO99] Collision based
	U-Forgery	$2^{n/2}$	2^{n+1}	$2^{n/2}$	[PvO99] Collision based
		1	$2^{2n/3}$	$2^{2n/3}$	[AH88] TM tradeoff, 2^n precomputation
NMAC-MD4 HMAC-MD4	E-Forgery	2^{58}	-	-	[CY06] Improved with Path 4 (3.D)
	Partial-KR	2^{63}	2^{40}	-	[CY06] Only for NMAC
	U-Forgery	2^{88}	2^{95}	-	**Our result** [FLN07b]
		2^{72}	2^{77}	-	[WOK08] Can be improved with Path 6
NMAC-MD5 *Related keys*	E-Forgery	2^{47}	-	-	[CY06]
	Partial-KR	2^{47}	2^{45}	-	[CY06]
	U-Forgery	2^{51}	2^{100}	-	**Our result** [FLN07b] — Same as [RR07]
		2^{75}	2^{75}	-	[WOK08]

Our attacks have a complexity greater than the birthday paradox, so they are not covered by Bellare's proofs. Note that some MAC constructions have security proof against PRF-attacks up to the birthday bound, but are vulnerable to key-recovery attacks. For instance, the envelope method with a single key, defined as $\text{MAC}_k(M) = H(k \| M \| k)$, was proved to be secure, but a key-recovery attack using 2^{67} known text-MAC pairs, and 2^{13} chosen texts (for a hash function with a 128-bit state) was found by Preneel and van Oorschot [PvO96]. However, in the case of NMAC, we can prove that the security against universal forgery cannot be less than the security of the compression function against a PRF distinguisher (see Appendix 4.A). This shows that NMAC offers good resistance beyond the birthday paradox: a universal forgery attack will have a time complexity of at least 2^n if there is no weakness in the compression function. Conversely, there is a generic attack against any iterated stateless MAC using a collision in the inner hash function [PvO95, PvO99]. In the case of NMAC it allows to search for the two subkeys k_1 and k_2 independently and requires $2^{n/2}$ queries and 2^{n+1} hash computations.

Like [CY06], we stress that our results on HMAC and NMAC do not contradict any security proof; on the contrary they show that when the hypotheses over the hash function are not met, an attack can be built.

Description of HMAC and NMAC. HMAC and NMAC are hash-based message authentication codes proposed by Bellare, Canetti and Krawczyk [BCK96a]. Let H be an iterated Merkle-Damgård hash function. Its HMAC is defined by

$$\text{HMAC}_k(M) = H(\bar{k} \oplus \text{opad} \,||H(\bar{k} \oplus \text{ipad}\,||M)),$$

where M is the message, k is the secret key, $\bar{k}$ is k padded to a full block of the hash function, opad and ipad are two fixed one-block values. Since H is assumed to be based on the Merkle-Damgård paradigm, we denote by H_k the modification of H where the public IV is replaced by the secret key k. Then NMAC with secret key (k_1, k_2) is defined by:

$$\text{NMAC}_{k_1,k_2}(M) = H_{k_1}(H_{k_2}(M)).$$

Thus, HMAC_k is essentially equivalent to $\text{NMAC}_{H(k \oplus \text{opad}), H(k \oplus \text{ipad})}$ [1]. NMAC is easier to analyse, and the security proof of HMAC is based on that of NMAC, while HMAC is easier to use in practice. Attacks on NMAC can usually be adapted to HMAC (pending few modifications), except in the related-key setting [2].

HMAC/NMAC Security. The security of a MAC algorithm is usually measured by the difficulty for an adversary having access to a MAC oracle to forge new valid MAC-message pairs. More precisely, we consider two types of attack: the existential forgery where the adversary must produce a valid MAC for *a message of his choice*, and the universal forgery where the adversary must be able to compute the MAC of *any message*.

The security of HMAC and NMAC was carefully analyzed by its designers. It was first shown in [BCK96a] that NMAC is a pseudorandom function family (PRF) under the two assumptions that (A1) the keyed compression function f_k of the hash function is a PRF, and (A2) the keyed hash function H_k is *weakly collision resistant* (WCR). The WCR assumption states that given oracle access to H_k for an unknown k, it is infeasible to find a collision in H_k. The proof for NMAC was then lifted to HMAC by further assuming that (A3) the key derivation function in HMAC is a

1. There is small difference in the padding: when we use $H(k||\cdot)$ instead of H_k, the length of the input of the hash function (which is included in the padding) is different.

2. If we need an oracle $\text{NMAC}_{k_1,k_2+\Delta}$, we cannot emulate it with any related-key HMAC oracle.

PRF. However, it was noticed that recent collision attacks [WLF$^+$05, WY05, WYY05c, WYY05b] invalidate the WCR assumption (A2) in the case of usual hash function like MD4 or MD5, because one can use the length extension attack to turn a collision attack into a WCR attack. This led Bellare [Bel06] to present new security proofs for NMAC under the PRF assumption (A1) only. As a result, the security of HMAC solely depends on (A1) f_k being a PRF and (A3) the key derivation being a PRF. The security of NMAC as a PRF holds only if the adversary makes less than $2^{n/2}$ NMAC queries (where n is the MAC size), since there is a generic forgery attack using the birthday paradox with $2^{n/2}$ queries [PvO95] (described in Section 1.3.4).

Since recent collision attacks cast a doubt on the validity of (A1), one may wonder if it is possible to exploit collision search breakthroughs to attack HMAC and NMAC instantiated with real-life hash functions. In particular, MD4 is a very tempting target since it is by far the weakest real-life hash function with respect to collision resistance. It is not too difficult to apply collision attacks on MD4 [WLF$^+$05, YWZW05], to obtain distinguishing and existential forgery attacks on HMAC/NMAC-MD4: for instance, this was done independently by Kim *et al.* [KBPH06] and Contini and Yin [CY06]. The situation is more complex with MD5, because the differential path found in the celebrated MD5 collision attack [WY05] is not well-suited to HMAC/NMAC since it uses two blocks: Contini and Yin [CY06] turned instead to the much older MD5 pseudo-collisions of de Boer and Bosselaers [dBB93] to obtain distinguishing and existential forgery attacks on NMAC-MD5 in the related-key setting. It is the use of pseudo-collisions (rather than full collisions) which weakens the attacks to the related-key setting.

Interestingly, universal forgery attacks on HMAC and NMAC seem much more difficult to find. So far, there are only two works in that direction. In [CY06], Contini and Yin extended the previous attacks to *partial* key-recovery attacks on HMAC/NMAC instantiated with MD4, SHA-0, and a step-reduced SHA-1, and related-key *partial* key-recovery attacks on NMAC-MD5. In [RR06], Rechberger and Rijmen improved the data complexity of Kim *et al.* [KBPH06] attacks, and extended them to a partial key recovery against NMAC-SHA-1. These attacks are only partial in the sense that the NMAC attacks only recover the second key k_2, which is not sufficient to compute new MACs of arbitrary messages; and the HMAC attacks only recover $H(k \oplus \text{ipad})$ where k is the HMAC secret key, which again is not sufficient to compute new MACs of arbitrary messages, since it does not give the value of k nor $H(k \oplus \text{opad})$. Note that recovering a single key of NMAC does not significantly improve the generic full key-recovery attack which recovers the keys one by one.

More recently, Rechberger and Rijmen have proposed full key-recovery attacks against NMAC in the related-key setting in [RR07]. They extended the attack of [CY06] to a full key-recovery attack against NMAC-MD5, and introduced a full key-recovery attack against NMAC when used with SHA-1 reduced to 34 rounds.

4.3 Key-Recovery Attacks on HMAC and NMAC

In this section, we give a high-level overview of our key-recovery attacks on HMAC and NMAC instantiated with MD4 and MD5. Detailed attacks will be given in the next two sections: Section 4.4 for MD4 and Section 4.5 for MD5. We will assume that the adversary can request the MAC of messages of his choice, for a fixed secret key, and the goal is to recover that secret key. In the related-key setting, we will assume like in [CY06] that the adversary can request the MAC of messages of his choice, for the fixed secret key as well as for other related secret keys (with a chosen relation). In fact, we will not even need the full output of MAC requests: we will only need to know if the two MACs of messages of our choice collide or not.

To simplify our exposition, we will concentrate on the NMAC case:

$$\mathrm{NMAC}_{k_1,k_2}(M) = H_{k_1}(H_{k_2}(M)).$$

The NMAC-MD4 attack can easily be extended to HMAC-MD4, pending minor modifications. Our NMAC attack will first recover k_2, then k_1. We will collect NMAC collisions of a special shape, in order to disclose hash collisions with first H_{k_2} then H_{k_1}.

4.3.1 Collision Attacks Based on Differential Cryptanalysis

Our attacks on HMAC and NMAC are based on differential collision attack on the underlying hash function. The attacks on MD4 are based Wang *et al.*'s attack, with specific differential paths built by the algorithm described in Section 3.5.

On the other hand, the attacks on NMAC-MD5 are based on the MD5 pseudo-collision by den Boer and Bosselaers [dBB93], like the attacks of [CY06]. Let IV be a 128-bit value satisfying the so-called dBB condition: the most significant bit of the last three 32-bit words of IV are all equal. Clearly, a randomly chosen IV satisfies the dBB condition with probability 1/4. It is shown in [dBB93] that in such a case, a randomly chosen 512-bit message M satisfies with heuristic probability 2^{-46}:

$$\mathrm{cMD5}(IV, M) = \mathrm{cMD5}(IV', M),$$

where IV' is the 128-bit value derived from IV by flipping the most significant bit of each of the four 32-bit words of IV. The probability 2^{-46} is obtained by studying the most likely differences for the internal state Q_i, as usual in differential cryptanalysis.

4.3.2 Extracting Hash Collisions from NMAC Collisions

We will extract hash collisions from NMAC collisions, that is, pairs (M_1, M_2) of messages such that:

$$M_1 \neq M_2 \quad \text{and} \quad \mathrm{NMAC}_{k_1,k_2}(M_1) = \mathrm{NMAC}_{k_1,k_2}(M_2). \tag{C}$$

Our attacks are based on the elementary observation that H-collisions can leak through NMAC. More precisely, two messages M_1 and M_2 satisfy (C) if and only if they satisfy either (C1) or (C2):

(C2) $M_1 \neq M_2$ and $H_{k_2}(M_1) = H_{k_2}(M_2)$: we have a collision in the inner hash function;

(C1) $H_{k_2}(M_1) = N_1 \neq N_2 = H_{k_2}(M_2)$ and $H_{k_1}(N_1) = H_{k_1}(N_2)$: we have a collision in the outer hash function .

If we select M_1 and M_2 uniformly at random, then (C) holds with probability 2^{-128} if NMAC is a random function. However, if we select many pairs (M_1, M_2) in such a way that (C2) holds with a probability significantly higher than 2^{-128}, then whenever $\mathrm{NMAC}_{k_1,k_2}(M_1) = \mathrm{NMAC}_{k_1,k_2}(M_2)$, it will be likely that we also have (C2). More precisely, we have (since $Ci \cap C = Ci$):

$$\frac{\Pr(C2|C)}{\Pr(C1|C)} = \frac{\Pr(C2)}{\Pr(C1)}$$

and we expect that $\Pr(C2) \gg \Pr(C1) \approx 2^{-128}$ for those pairs (M_1, M_2).

Note that the Merkle-Damgård construction used in H leads to a simple heuristic way to distinguish both cases (without knowing the secret keys k_1 and k_2) if M_1 and M_2 have the same length, and therefore the same padding block pad: if $H_{k_2}(M_1) = H_{k_2}(M_2)$, then for any M, we have $H_{k_2}(M_1||\mathsf{pad}||M) = H_{k_2}(M_2||\mathsf{pad}||M)$. In other words, the condition (C2) is preserved if we append $\mathsf{pad}||M$ to both M_1 and M_2 for a randomly chosen M, but that is unlikely for the condition (C1).

To illustrate our point, assume that we know a non-zero Δ such that for all keys k_2, a randomly chosen one-block message M_1 satisfies with probability 2^{-64} the condition $H_{k_2}(M_1) = H_{k_2}(M_2)$ where $M_2 = M_1 \boxplus \Delta$. If we select 2^{64} one-block messages M_1 uniformly at random and call the NMAC oracle on each M_1 and $M_1 \boxplus \Delta$, we are likely to find a pair $(M_1, M_2 = M_1 \boxplus \Delta)$ satisfying (C). By the previous reasoning, we expect that such a pair actually satisfies (C2).

Thus, the NMAC oracle allows us to detect collisions on H_{k_2}, if we are able to select messages which have a non-negligible probability of satisfying (C2). To detect collisions in H_{k_1}, we will use the values of k_2 (recovered using collisions in H_{k_2}): then, we can compute H_{k_2} and directly check whether the NMAC collisions come from (C1). We now explain how to use such collision detections to recover the secret keys k_2 and k_1.

4.3.3 Collision-based IV-recovery Attacks

The previous subsection suggests the following scenario. Assume that a fixed key k is secret, but that one is given access to an oracle which on input M_1 and M_2, answers whether $H_k(M_1) = H_k(M_2)$ holds or not. Can one use such an oracle to recover the secret key k? If so, we have what we call a collision-based IV-recovery attack.

A collision-based IV-recovery attack would clearly reveal the second key k_2 of NMAC, because of (C2): a collision in H_{k_2} gives a collision in NMAC. But it is not clear why this would be relevant to recover the outer key k_1. To recover k_1 thanks to (C1), we would need the following variant of the problem. Namely, one would like to retrieve a secret key k_1 when given access to an oracle which on input M_1 and M_2, answers whether $H_{k_1}(H_{k_2}(M_1)) = H_{k_1}(H_{k_2}(M_2))$ holds or not, where k_2 is known. Since the messages are first processed through a hash function, the adversary no longer chooses the input messages of the keyed hash function, and this oracle is much harder to exploit than the previous one. We call such attacks composite IV-recovery attacks. In the attack on HMAC/NMAC-MD4, we will exploit the Merkle-Damgård structure of H_{k_2} to efficiently extend our basic IV-recovery attacks into composite IV-recovery attacks.

We will present two types of collision-based IV-recovery attacks. The first type is due to Contini and Yin [CY06] and uses related messages, while the second type is novel, based on IV-dependent differential paths.

Using Related Messages. We present the first type of IV-recovery attacks. Assume that we know a specific differential path corresponding to a message difference Δ and with total probability p much larger than 2^{-128}. In other words, a randomly chosen message M will satisfy with probability p:

$$H_k(M) = H_k(M \boxplus \Delta).$$

By making approximately $2/p$ queries to the H_k-oracle, we will obtain a message M such that $H_k(M) = H_k(M \boxplus \Delta)$. Contini and Yin [CY06] then make the heuristic assumption that the pair $(M, M \boxplus \Delta)$ must follow the whole differential path, and not just the first and last steps. Since they do not justify that assumption, let us say a few words about it. The assumption requires a strong property on our specific differential path: that there are no other differential paths with better (or comparable) probability. In some sense, differential cryptanalysis on block ciphers use similar assumptions, and to see how realistic that is, one makes experiments on reduced-round versions of the block cipher. However, one might argue that there are intuitively more paths in hash functions than in block ciphers because of the following facts:

– The message length of a compression function is much larger than the length of the round key in a block cipher;

– A step of the compression function is usually much simpler than a block-cipher step.

Also, because the paths for hash functions have a different shape from those of block ciphers, experiments on reduced-round versions may not be as conclusive.

The paper [CY06] shows that for usual differential paths (like those of MD4), if $(M, M \boxplus \Delta)$ satisfies the whole path, then one can build plenty of messages M^* closely related to M such that:

– If a specific internal register Q_i (during the computation of $H_k(M)$) satisfies certain conditions, then the pair $(M^*, M^* \boxplus \Delta)$ follows the whole path with probability p or larger, in which case $H_k(M^*) = H_k(M^* \boxplus \Delta)$.

– Otherwise, the pair $(M^*, M^* \boxplus \Delta)$ will drift away from the path at some position, and the probability of $H_k(M^*) = H_k(M^* \boxplus \Delta)$ is heuristically 2^{-128}.

Thus, by sending to the oracle many well-chosen pairs $(M', M' \boxplus \Delta)$, one can learn many bits of several internal register Q_i's during the computation of $H_k(M)$. Applying exhaustive search on the remaining bits of such Q_i's, one can guess the whole contents of four consecutive Q_i's. By definition of cMD4 and cMD5, it is then possible to reverse the computation of $H_k(M)$, which discloses $k = (Q_{-4}, Q_{-3}, Q_{-2}, Q_{-1})$. A more detailed description of the related messages is given in Section 4.5.

Using IV-dependent Differential Paths. We now present a new type of IV-recovery attacks, that we will apply against MD4. Assume again that we know a specific differential path corresponding to a message difference Δ and with total probability p much larger than 2^{-128}, but assume this time that the path is IV-dependent: it holds only if the IV satisfies a specific condition (SC). In other words, if k satisfies (SC), then a randomly chosen message M will satisfy with probability p:

$$H_k(M) = H_k(M \boxplus \Delta).$$

But if k does not satisfy (SC), the pair $(M, M \boxplus \Delta)$ will drift away from the differential path from the first step, leading us to assume that $H_k(M) = H_k(M \boxplus \Delta)$ will hold with probability only 2^{-128}.

This leads to the following attack: we submit approximately $2/p$ pairs $(M, M \boxplus \Delta)$ to the H_k-oracle, and conclude that k satisfies (SC) if and only if $H_k(M) = H_k(M \boxplus \Delta)$ for at least one M. This can be repeated with several paths, each disclosing a different bit of the key. When sufficient information on k has been gathered, the rest of k can be guessed by exhaustive search if we have at least one collision.

Notice that in some sense the differential path of the MD5 pseudo-collision [dBB93] is an example of IV-dependent path where (SC) is the dBB condition, but it does not disclose much information about the IV. We would need to find many IV-dependent paths. Our attack on HMAC/NMAC-MD4 will use 22 such paths, which were found by an automated search.

We note that such attacks require an assumption similar to the previous IV-recovery attack. Namely, we assume that for the same message difference Δ, there is no differential paths with better (or comparable) probability, with or without conditions on the IV. To justify this assumption for our HMAC/NMAC-MD4 attack, we have performed experiments which will be explained in Section 4.4.

4.3.4 Subtleties Between the Inner and Outer Keys

Although the recovery of the inner key k_2 and the outer key k_1 both require IV-recovery attacks, we would like to point out important differences between the two cases. As mentioned previously, the recovery of k_2 only requires a basic IV-recovery attack, while the recovery of k_1 requires a composite IV-recovery attack. The composite IV-recovery attacks will be explained in Section 4.4 for MD4, and Section 4.5 for MD5.

When turning a basic IV-recovery attack into a composite IV-recovery, there are two important restrictions to consider:

– We have no direct control over the input of the outer hash function, it is the result of the inner hash function. So IV-recovery attacks using message modifications will become much less efficient when turned into composite IV-recovery. We will see this in Section 4.5 when extending the partial key-recovery from [CY06] into a full key-recovery.

– Since the input of H_{k_1} is a hash, its length is only 128 bits. Any differential path using a message difference Δ with non-zero bits outside these 128 first bits will be useless. This means that the partial key-recovery attacks from [CY06] against MD4, SHA-0 and reduced SHA-1 cannot be extended into a full key-recovery.
  Using related keys one can use a differential path with a difference in the IV and no message difference — such as the one from [dBB93] — and try a given message with both keys. However, if we want to get rid of related keys, we need a differential path with no IV difference and a difference in the beginning of the message.

Later Improvements. After the publication of our attack, L. Wang *et al.* proposed an improvement of the attack using a new kind of composite IV-recovery based on near-collisions in [WOK08]. In this work, they exploit the full output of the hash function, rather than just checking for collisions. The full output is not available when recovering the inner key k_2, but it is available when recovering the outer key k_1. They use a differential path with no state difference in most of the second and third round of MD4 (there is a local collision in the first and second round), and a small difference is introduced by the message at the end of the third round. Message pairs following the path are identified by looking for near-collisions. Then, knowing the message difference and the actual output difference as seen from the NMAC output, they can recover bits of the key. This attack is more efficient than ours because better differential paths can be found for near-collisions.

In Appendix 3.D, Path 6 describes a new differential path that can be used to further improve this attack.

4.3.5 Summary

To summarize, our attacks will have essentially the following structure (the MD5 attack will be slightly different because of the related-key setting):

1. Apply an IV-recovery attack to retrieve k_2, repeating sufficiently many times:
   a) Select many one-block messages M uniformly at random.
   b) Observe if $\text{NMAC}_{k_1,k_2}(M) = \text{NMAC}_{k_1,k_2}(M \boxplus \Delta_1)$ for some M and a well-chosen Δ_1.
   c) Deduce information on k_2.
2. Apply a composite IV-recovery attack to retrieve k_1, repeating sufficiently many times:
   a) Construct carefully many pairs (M_1, M_2).
   b) Observe if $\text{NMAC}_{k_1,k_2}(M_1) = \text{NMAC}_{k_1,k_2}(M_2)$ for some pair (M_1, M_2).
   c) Deduce information on k_1.

4.4 Attacking HMAC/NMAC-MD4

4.4.1 Our IV-recovery Attack against MD4

In order to find differential paths which leak information about the key, we consider differential paths with a message difference in the first word, *e.g.* $\delta m_0 = 1$. Then in the first steps of the compression function, we have:

$$Q_0 = (Q_{-4} \boxplus \mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxplus m_0)^{\lll 3}$$
$$Q'_0 = (Q_{-4} \boxplus \mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxplus m_0 \boxplus 1)^{\lll 3}$$

Hence $Q_0^{[3]} \neq Q_0'^{[3]}$. Then

$$Q_1 = (Q_{-3} \boxplus \mathsf{IF}(Q_0, Q_{-1}, Q_{-2}) \boxplus m_1)^{\lll 7}$$
$$Q'_1 = (Q_{-3} \boxplus \mathsf{IF}(Q'_0, Q_{-1}, Q_{-2}) \boxplus m_1)^{\lll 7}$$

Thus, if $Q_{-1}^{[3]} \neq Q_{-2}^{[3]}$, we will have $\mathsf{IF}(Q_0, Q_{-1}, Q_{-2}) \neq \mathsf{IF}(Q'_0, Q_{-1}, Q_{-2})$ and $Q_1 \neq Q'_1$. On the other hand, if $Q_{-1}^{[3]} = Q_{-2}^{[3]}$ and there is no carry when going from Q_0 to Q'_0, then $Q_1 = Q'_1$. Therefore, collision paths where $Q_{-1}^{[3]} = Q_{-2}^{[3]}$ will be significantly different from collision paths where $Q_{-1}^{[3]} \neq Q_{-2}^{[3]}$. This suggests that the collision probability will be correlated with the condition (SC) : $Q_{-1}^{[3]} = Q_{-2}^{[3]}$, and we expect to be able to detect the bias. More precisely we believe that the case $Q_{-1}^{[3]} \neq Q_{-2}^{[3]}$ will give a much smaller collision probability, since it means that an extra difference is introduced in step 1.

To check this intuition experimentally, we ran one cMD4 round (16 steps) with a random IV on message pairs $(M, M \boxplus 1)$ where M was also picked at random, and we looked for pseudo-collisions in $Q_{12}...Q_{15}$ with the following properties:

– The weight of the non-adjacent form of the difference is lower or equal to 4.
– There is no difference on $Q_{12}^{[12]}$.

The second condition is added to eliminate trivial paths which simply keep the difference introduced in $Q_0^{[3]}$ without modifying it. We ran this with $5 \cdot 10^{11}$ random messages and IVs and found 45624 collisions out of which 45515 respected the condition: this gives a ratio of about 420. This does not prove that we will have such a bias for collisions in the full MD4, but it is a strong evidence.

The same arguments apply when we introduce the message difference in another bit k (*i.e.* $\delta m_0 = 2^k$): we expect to find more collisions if $Q_{-1}^{[k+s_0]} = Q_{-2}^{[k+s_0]}$.

We ran a differential path search algorithm to find such paths, and we did find 22 paths for different values of k with $Q_{-1}^{[k+s_0]} = Q_{-2}^{[k+s_0]}$. The path for $k = 0$ is Path 5 in Appendix 3.D, and the other paths are just rotated versions of this path. We can deduce 22 paths from this one by rotating all the bit differences and bit conditions: it works on bit positions 0, 1, 3, 4, 6-8, 12-17, 19-24, 26, 27, and 29, and fails on other positions due to carry expansions. The corresponding set of sufficient conditions contains 79 conditions on the internal variables Q_i, so we expect that for a random message M:

$$\Pr\left[\mathrm{MD4}(M) = \mathrm{MD4}(M + \Delta)\right] = p \geq 2^{-79} \qquad \text{if } Q_{-1}^{[k+s_0]} = Q_{-2}^{[k+s_0]}$$
$$\ll p \qquad \text{if } Q_{-1}^{[k+s_0]} \neq Q_{-2}^{[k+s_0]}$$

If we try 2^{82} message pairs per path, we will find a collision for every path whose condition is fulfilled with a probability [3] of more than 99%. Then we know 22 bits of the IV ($Q_{-1}^{[k+s_0]} = Q_{-2}^{[k+s_0]}$ or $Q_{-1}^{[k+s_0]} \neq Q_{-2}^{[k+s_0]}$), which leaves only 2^{106} IV candidates. To check if a given IV is the correct one, we just check whether it gives a collision on the pairs colliding with the real IV, so we expect to find the IV after computing 2^{105} pairs of hashes in an offline phase.

3. We have $\left(1 - \left(1 - 2^{-79}\right)^{2^{82}}\right)^{22} > 0.992$

We show in Appendix 4.B.2 how to reduce the search space to 2^{94} keys by extracting more than one bit of information when a collision is found. This gives an IV-recovery attack against MD4 with a data complexity of 2^{88} MD4 oracle queries, and a time complexity of 2^{94} MD4 evaluations.

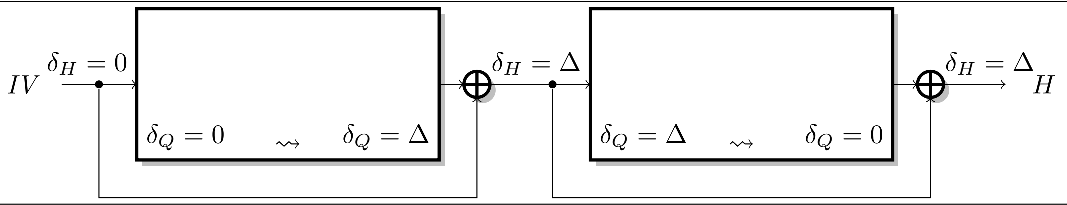
4.4.2 Deriving a Composite IV-recovery Attack against MD4

To turn this into a composite IV-recovery attack, we need to efficiently compute message pairs M, M' such that $H_{k_2}(M) = H_{k_2}(M') \boxplus \Delta$ (we will need 2^{82} such pairs). Since the value of k_2 is known at this point (in the HMAC attack, we first recover it using the basic IV-recovery attack), we can compute $H_{k_2}(M)$ and find such pairs offline. If we do this naively using the birthday paradox, we need to hash about 2^{106} random messages to have all the pairs we need [4]. Then we can use the IV-recovery attack to get k_1.

Actually, we can do much better: we use the birthday paradox to find one pair of one-block messages (R, R') such that $H_{k_2}(R') = H_{k_2}(R) \boxplus \Delta$, and then we extend it to a family of two-block message pairs such that $H_{k_2}(R'||Q') = H_{k_2}(R||Q) \boxplus \Delta$ with very little extra computation. In the end, the cost to generate the messages with $H_{k_2}(M) = H_{k_2}(M') \boxplus \Delta$ will be negligible and the composite IV-recovery attack is as efficient as the basic one. This is the most important part of our work: thanks to our new path, we only need a low level of control on the input of the hash function to extract the IV.

Extending a Pair of Good Messages into a Family of Pairs. Figure 4.1 shows how we will create many message pairs with $H_{k_2}(M) = H_{k_2}(M') \boxplus \Delta$. We use a pair of one-block message (R, R') such that $H_{k_2}(R') = H_{k_2}(R) \boxplus \Delta$. Then we will generate a second block pair (Q, Q') such that $H_{k_2}(R'||Q') \boxminus H_{k_2}(R||Q) = \Delta$. Thanks to the Davies-Meyer construction of the compression function, all that we need is a differential path which starts with a Δ difference and ends with a zero difference in the internal state; then the feed-forward of H will keep $\delta H = \Delta$. Such a path can be found by a differential path search algorithm, or created by hand by slightly modifying a collision path.

Figure 4.1 Generating many pairs of message with a fixed hash difference



In this step, we also have to take care of the padding in MD4. Usually we ignore it because a collision at the end of a block is still a collision after an extra block of padding, but here we want a specific non-zero difference, and this will be broken by an extra block. So we have to adapt our collision finding algorithm to produce a block with a 55-byte message M and the last 9 bytes fixed by the MD4 padding. This can be done with nearly the same complexity as unconstrained MD4 collisions (about 4 MD4 computations per collision) using the technique of Section 3.4.4. Thus, the cost of the message generation in the composite IV-recovery attack drops from 2^{106} using the birthday paradox to 2^{90} and becomes negligible in the full attack.

4. This gives 2^{210} pairs of messages, and each pair has a probability of 2^{-128} to have the correct difference.

4.4.3 MD4 Attack Summary

This attack uses the same IV-recovery attack for the inner key and the outer key, with a complexity of 2^{88} online queries and 2^{94} offline computations. We manage to keep the complexity of the composite IV-recovery as low as the basic IV-recovery because we only need to control the hash differences, and we introduce a trick to generate many messages with a fixed hash difference.

In Appendix 4.B.1 we show how to reduce a little bit the query complexity of the attack, and in the end the NMAC full key-recovery attack requires 2^{88} requests to the oracle, and 2×2^{94} offline computations.

4.5 Attacking NMAC-MD5

In this section, we will describe the attack of Contini and Yin [CY06], and we extend it to a full key recovery. This improved attack was independently found by Rechberger and Rijmen in [RR07].

As for MD4, the IV-recovery attack is based on a specific differential path, and assumes that when a collision is found with the given message difference, the Q_i's follow the path. This gives some bits of the internal state already, and a kind of message modification technique to disclose more bits is proposed in [CY06].

If the path has a condition on the value of a bit $Q_t^{[k]}$ in the step t, then we can extract some extra bits from the register Q_t. To simplify the description of the message modification, let us assume that there are no rotations in MD4 or MD5. We set $\Delta = 2^{k-1}$ and modify the message M into a message M^*:

$$m_j^* = \begin{cases} m_j & \text{if } j < t \\ m_j + \Delta & \text{if } j = t \\ \text{random} & \text{if } j > t \end{cases}$$

Then, if we compute MD4 or MD5 without the rotation, we have:

$$Q_j^* = \begin{cases} Q_j & \text{if } j < t \\ Q_j + \Delta & \text{if } j = t \\ \text{random} & \text{if } j > t \end{cases}$$

We call the MAC oracle with enough such messages (with a different random part) to distinguish if Q_t^* still follows the path. If it does, this means that $Q_t^{*[k]} = Q_t^{[k]}$: there was no carry in $Q_t + \Delta$, therefore $Q_t^{[k-1]} = 0$. On the other hand, if it does not follow the path anymore, then $Q_t^{[k-1]} = 1$.

We can find $Q_t^{[k-2]}$ if we set Δ so that there is a carry up to the bit k if and only if $Q_t^{[k-2]} = 1$:

$$\Delta = \begin{cases} 2^{k-2} & \text{if } Q_j^{[k-1]} = 1 \\ 2^{k-2} + 2^{k-1} & \text{if } Q_j^{[k-1]} = 0 \end{cases}$$

and we can repeat this to get the bits $Q_t^{[k-1]} ... Q_t^{[0]}$. The rotations have little effect over this simplified explanation, they mainly limit the number of bits we can recover (see [CY06] for details).

This attack allows to recover 80 bits of the key with 2^{47} queries. The remaining key bits can be found by exhaustive search.

4.5.1 The IV-recovery Attack against MD5

The IV-recovery attack on MD5 is the same as the one presented in [CY06]. It uses the related-message technique with the pseudo-collision path of de Boer and Bosselaers [dBB93]. Since the differences are in the IV and not in the message, the IV-recovery needs an oracle that answers whether $\text{MD5}_{IV}(M) = \text{MD5}_{IV'}(M)$, instead of the standard oracle that answers whether $\text{MD5}_{IV}(M) = \text{MD5}_{IV}(M')$. To apply this to an HMAC key-recovery, we will have to use the related-key model: we need oracles for NMAC_{k_1,k_2} and $\text{NMAC}_{k_1,k_2'}$ to recover k_2, and oracles for NMAC_{k_1,k_2} and NMAC_{k_1',k_2} to recover k_1.

The IV-recovery attack in the related-key setting requires 2^{47} queries and 2^{45} hash computations, and this translates into a partial key-recovery (which recovers k_2) against NMAC-MD5 in the related-key model with the same complexity.

4.5.2 Deriving a Composite IV-recovery against MD5

To extend this to a composite IV-recovery attack, we run into the problem previously mentioned; to use this attack we need to create many inputs N^* of the hash function related to one input N, but these inputs are the outputs of a first hash function, and we cannot choose them freely: $N = \text{MD5}_{k_2}(M)$. However, we know k_2, so we can compute many $N_R = H_{k_2}(R)$ for random messages R and select those that are related to a particular N; if we want to recover bits of Q_t we will have to choose $32(t+1)$ bits of N_R. We also run into the problem that any N_R is only 128 bits long; the last 384 bits will be fixed by the padding and they are the same for all messages. Therefore, we can only use the related-message technique to recover bits of the internal state in the very first steps, whereas in the simple IV-recovery it is more efficient when used to recover the internal state of later steps (Contini and Yin used step 11 to 14). If we want to recover bits of Q_0 (due to the rotation we can only recover 25 bits of them), we need to produce 24×2^{45} messages N^* with the first 32 bits chosen; this will cost $24 \times 2^{45} \times 2^{32} \approx 2^{82}$ hash computations. Then, we know 25 bits of Q_0, plus the most significant bit of Q_1, Q_2, and Q_3; we still have 100 bits to guess. Thus, we have a related-key composite IV-recovery attack against MD5 with $2 \times 24 \times 2^{45} \approx 2^{51}$ oracle queries and 2^{100} MD5 evaluations.

If we try to guess bits in Q_1, we have to select at least 2^{44} hashes with 64 chosen bits; this costs about 2^{108} MD5, so it does not improve the attack.

4.5.3 MD5 Attack Summary

Thus, the Contini-Yin NMAC-MD5 attack can be extended into a full key-recovery attack in the related-key setting, with a query complexity of 2^{51}, a time complexity of 2^{100} MD5 operations, and success rate of 2^{-4} (due to the dBB condition for k_1 and k_2).

It is a very simple extension of the attack from Contini and Yin: we apply their technique to recover the outer key, but since we cannot choose the value of $H_{k_2}(M)$, we compute it for many random messages until we find a good one. This requires to change the step in which we extract internal bits, and the complexity becomes much higher.

4.6 Application to APOP

The Post Office Protocol Version 3 (POP3) is a standard protocol for remote mailbox access, defined by [RFC1081] in 1988 (current version is [RFC1939]). It is one of most prevalent protocols for e-mail retrieval, together with the Internet Message Access Protocol Version 4 (IMAP4), first described in 1994 by [RFC1730] (current version is [RFC3501]). Virtually all modern e-mail readers support both protocols, as well as most e-mail servers. IMAP is usually recommended because it consumes less resources thanks to its online state, but according to different statistics gathered through the web, between 10% and 50% of users are using POP3: it is still a very widely used protocol.

Initially, POP3 only supported plaintext password through the `USER` and `PASS` commands; `APOP` was added in 1993 in [RFC1460] to provide a simple challenge-response authentication, and avoid passive eavesdropping attacks. The `AUTH` command was added later [RFC1734] and allows to use the IMAP authentication mechanisms, such as Kerberos, GSS-API, S/Key or CRAM-MD5. Moreover POP3 now supports optional SSL encryption which protects the whole session.

Servers implementing the `APOP` command send a challenge (formatted as a message identifier, or msg-id) in their greeting message, and the client authenticates itself by sending the username, and the MD5 of the challenge concatenated with the password: MD5($msg\text{-}id||passwd$). The server performs the same computation on his side, and checks if the digests match. Thanks to this trick, an eavesdropper will not learn the password, provided that MD5 is a partial one-way function.

A typical `APOP` session looks like:

```
S: +OK POP3 ready <171.11776027@pop.somemail.com>
C: APOP alice 99808c6ca2168717c522192f3c3861f3
S: +OK maildrop has 1 message (369 octets)
```

In this example, Alice's password is '`penguin`', and she identifies herself by sending the digest MD5("`<171.11776027@pop.somemail.com>penguin`"). The RFC states that the challenge has to be a message identifier, and that it has to be different for each connection; for instance, it can be generated using a time counter.

`APOP` Security. Since there is no integrity protection and no authentication of the server, this protocol is subject to a man-in-the-middle attack, but a man-in-the-middle should not be able to learn the password or to re-authenticate later. Quoting [RFC1939]:

> It is conjectured that use of the `APOP` command provides origin identification and replay protection for a POP3 session.

This challenge-response can be seen as a MAC algorithm, known as the secret-suffix method: $\text{MAC}_k(M) = \text{MD5}(M||k)$. This construction is weak for at least two reasons: first, it allows off-line collision search so there is a generic forgery attack with $2^{n/2}$ computations and one chosen-text MAC; second, the key-recovery attack against the envelope method of Preneel and van Oorschot [PvO96] can be used on the suffix method with 2^{67} offline computations and 2^{13} chosen text MACs.

This is a hint that `APOP` is weak, but these attacks require more computations than the birthday paradox, and the first one is mostly useless in a challenge-response protocol. However, we can combine these weaknesses with the collision attack against MD5 to build a practical attack against `APOP`.

4.6.1 The `APOP` Attack

In this attack, we act as the server, and we send some specially crafted challenges to the client. We use pairs of challenges, such that the corresponding digests collide only if some part of the password was correctly guessed. Our attack is similar to the attack against the envelop MAC in that it uses the fact that the length of the challenge is variable, so that the password can fall between two MD5 blocks.

Figure 4.2 Testing the first password character. We use a collision M, M' to build the challenges c, c'.

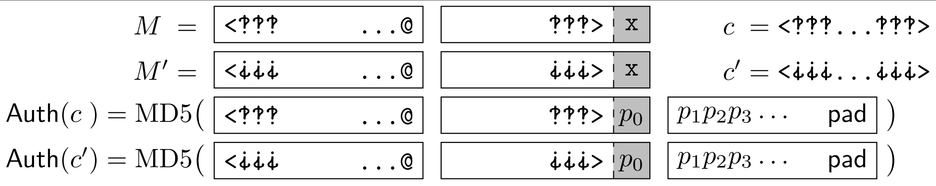
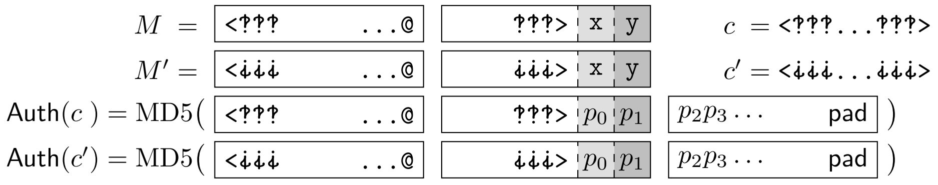


Figure 4.3 Testing the second password character.



Let us assume we can generate a MD5 collision with some specific format: M = "<‽‽‽...‽‽‽>x" and M' = "<⸘⸘⸘...⸘⸘⸘>x", where M and M' have both size 128 bytes (2 MD5 blocks); 'x' is a guess for the first password character, while the '‽' and '⸘' represent any character chosen by the collision finding algorithm. We send "<‽‽‽...‽‽‽>" and "<⸘⸘⸘...⸘⸘⸘>" as challenges, and the client returns MD5("<‽‽‽...‽‽‽>$p_0p_1p_2...p_{n-1}$") and MD5("<⸘⸘⸘...⸘⸘⸘>$p_0p_1p_2...p_{n-1}$"), where "$p_0p_1p_2...p_{n-1}$" is the user's password (the p_i's are the characters of the password).

As we can see in Figure 4.2, the first password character will be in the second MD5 block, while the rest of the password is in the third block. Therefore, if p_0 = 'x', the two hashes will collide after the second block, and since the end of the password and the padding are the same, we can detect the collision in the full hashes (and it is very unlikely that the two hashes collide for $p_0 \neq$ 'x'). Therefore we are able to test the first password character without knowing the others. We construct pairs of challenge to test the 256 possible values, and stop once we have found the first password character.

Then we generate a new collision pair to recover the second character of the password: M = "<‽‽‽...‽‽‽>p_0y" and M' = "<⸘⸘⸘...⸘⸘⸘>p_0y", so as to test if p_1 = 'y' (see Figure 4.3). Thus, we can learn the password characters one by one in linear time.

This motivates the need for message freedom in MD5 collisions: we need to generate collisions with a specific format, and some chosen characters. Using the techniques of Section 3.4, we can indeed choose some part of the colliding messages, and we are able to recover the first three passwords characters. We believe most passwords in real use are short enough to be found by exhaustive search once we know these first characters, and this attack will allow us to re-authenticate later. We stress that this attack *is* practical: it needs less than one hour of computation, and a few hundreds authentications.

Since people often read their mail from different places (including insecure wireless networks and Internet cafés), we believe that this man-in-the-middle setting is rather realistic for an attack against `APOP`. Moreover most mail clients automatically check the mailbox on a regular basis, so it seems reasonable to ask for a few hundred authentications.

Table 4.1 Mail client survey. The attack works on clients doing minimal checks

Mail client	Status in 2007	Status in 2010
Mozilla Thunderbird	**Minimal checks**	Checks ASCII
Microsoft Exchange / Outlook (Express)	*No APOP support*	
Qualcomm Eudora	**Minimal checks**	*No new version*
Apple Mail	*No APOP support*	
Mutt	**Minimal checks**	Checks ASCII
Novell Evolution	**Minimal checks**	Checks ASCII
Fetchmail	**Minimal checks**	Strict checks
KMail	Strict checks	Strict checks

4.6.2 Practical Implementation of the APOP Attack

To implement this attack, we need to efficiently generate MD5 collisions with some chosen parts; we mainly have to fix the last word, which is precisely what we can do efficiently on MD5, using the techniques of Section 3.4.

Additionally, the POP3 RFC [RFC1939] requires the challenge to be a msg-id, which means that:

– It has to begin with '`<`' and end with '`>`'
– It must contain exactly one '`@`', and the remaining characters must be alphanumeric ASCII characters, or a few punctuation signs. Since we cannot find two colliding ASCII messages with the current collision techniques, this means that the attack will use non-RFC-compliant challenges.

In practice, when this attack was discovered, most mail clients were only doing minimal checks of the challenge. More precisely, most software only rejected the following characters:

– `0x00 Null`: used as end-of-string in the C language
– `0x3e Greater-Than Sign` ('`>`'): used to mark the end of the msg-id
– `0x0a Line-Feed`: used for end-of-line (POP is a text-based protocol)
– `0x0d Carriage-Return`: also used for end-of-line

Additionally, some mail clients needed at least one '`@`' in the msg-id, and Qualcomm Eudora truncated the challenge to 256 bytes, but we use challenges shorter than 128 bytes long anyway.

The attack has been implemented and tested with several mail clients, and the results are given in Table 4.1. The attack works on clients that only do minimal checking of the challenge conformity. When this attack was found, we warned the developers of these mail clients and advised them to reject non-RFC compliant challenges so as to thwart the attack. Table 4.1 shows that they now check the conformity of the message-id more thoroughly, which currently prevents the attack.

We also note that some mail clients, such as Mozilla Thunderbird, allow a *downgrade attack*. Thunderbird uses the best authentication method available, but when an attacker impersonates the server, he can claim to support only APOP so as to mount the attack and recover the password.

Generating the Challenges. We build the colliding challenges using Wang *et al.*'s path [WY05], which gives two-block collisions. The first block is more expensive to find, so we use the same block for every msg-id, and we fix the first character as a '`<`', and the last character as a '`@`'. Then we have to generate a second block for every password character test. For each character test, we have to generate a collision where the last word is chosen, and 4 characters are avoided in the message (`0x00`, `0x3e`, `0x0a` and `0x0d`). Using the techniques from Section 3.4.6 we can do this in less than 5 seconds per collision on a standard desktop computer. An example of collision is given in Table 3.5 in Appendix 3.B.

Unfortunately, this path uses a difference $\delta M_{14} = 2^{32}$ and this corresponds to a difference in character 60. Note that in order to learn the i-th password character p_{i-1}, we need to generate a collision where we fix the last $i+1$ characters (i password characters, plus a '`>`' sign to form a correct msg-id). Therefore, we can only retrieve 3 characters of the password with Wang's path.

Attack Complexity. To estimate the complexity of this attack, we assume that the user's password is 8-characters long, and each character has 6 bits of entropy. Under these assumptions, we have to generate 3×2^5 collisions on average, and wait for about 3×2^6 identifications if we test the characters starting with the most likely ones. Each collision takes about 5 seconds to generate; so the limiting factor will be the interval between two identifications attempts from the client. If we assume that he identifies once per minute our attack will take about 3 hours. In a second phase we can do an offline exhaustive search over the missing password characters. We expect to find them after 2^{30} MD5 computations and this is expected to take less than half an hour on a basic PC.

Security Impact. Since `APOP` is not designed to be secure if there is a man-in-the-middle, it is difficult to evaluate the impact of our attack: a man-in-the-middle can easily access the mails of his victim. With our attack, he will learn the password, and the consequences depend on the specific situation:

– if the attacker can only act as a man-in-the-middle for a limited time (*e.g.* for a lunchtime attack, or using an open WiFi network) the password can be used to authenticate later on the mail server;

– if the same password is used for another resource (*e.g.* a computer account), the attacker will gain access to this resource.

4.6.3 Further Improvements

This attack against `APOP` has been independently found by Sasaki *et al.* almost simultaneously; their work is available on the eprint [SYA07].

Later, they improved the attack using a new differential path, which allows to recover more password characters [SWOK08]. In this attack, they use the pseudo-collision path of den Boer and Bosselaers [dBB93] for the last block. This path is well-suited for this attack because it does not have any message difference, and it has relatively few conditions. This allows to recover up to 31 password characters in practice. However, in order to use this pseudo-collision path in a collision attack, one has to find a suitable near-collision with the correct output difference. Sasaki *et al.* built a specific difference path for this purpose, and found a so-called IV-bridge in time 2^{42}. More recently, Xie *et al.* described a new collision attack using a similar path [XLF08]. This path can be used to generate an IV-bridge with complexity only 2^{21}. Note that, so far, all collision paths still need a difference in bit 31 of one of the message words, so one of the challenges will contain a non-ASCII character.

Following the same idea, one can use the birthday paradox to generate the IV-bridge: it requires a precomputation of about 2^{64} MD5, but it needs to be done only once, independently of the user being attacked. Moreover, this approach will allow an attack with RFC-compliant messages, which would work with current mail readers.

Bit-Free Collisions. A recent improvement to the attack suggests to build challenge pairs that can be used to test several password guesses with a single authentication pair. In [WSSO09] L.Wang *et al.* show how to build challenge pairs that can test two password guesses, and reduce by one half the number of authentications needed for the attack.

Application to Other Protocols. The attack can be applied to any challenge-response protocol based on the secret-prefix MAC. For instance, in the HTTP Digest Authentication defined in [RFC2617], the client computes MD5(username‖realm‖password), where the *realm* is sent by the server. The same attack can be used using specially crafted *realms*.

However, this attack would not really work in practice, because the *realm* is not the challenge itself, it is supposed to be an identifier of the server. Most client software will show the *realm* to the user, and ask the user to re-enter the password is the *realm* has changed. Therefore users are very likely to notice something unusual and stop sending authentication information.

4.6.4 Recommendations

We believe `APOP` is to be considered broken, and we advise users to switch to another authentication protocol if possible, such as the `AUTH` command introduced by [RFC1734]. However, the current attack needs non-RFC-compliant challenges, and it is possible to avoid the attack by strictly check if the challenge follows the RFC. This has now been implemented in most mail clients, but future improvements might give an attack with RFC-compliant challenges.

4.A NMAC Security beyond the Birthday Paradox

In this section we study the security of NMAC when the adversary can make enough queries to use the birthday paradox. There is a generic existential forgery attack, but we will show that universal forgery against NMAC requires more resources. Actually, we will prove a much stronger statement: NMAC is secure against short forgery (less than one block) if the underlying compression function is a PRF.

We define the 1-block MAC advantage of a MAC-adversary as (using the same notations as [Bel06], and $\mathcal{B}$ to denote the set of 1-block messages):

$$\mathbf{Adv}_f^{\text{1-MAC}}(A) = \Pr\left[\mathtt{pad}(M) \in \mathcal{B},\ (M,x) \text{ is a forgery} : (M,x) \leftarrow A^{f(K,\cdot),VF_f(K,\cdot,\cdot)}; K \xleftarrow{\$} Keys\right]$$

Note that the adversary is allowed to make any query of any length to the MAC oracle, we only limit the length of the forgery. We require that the padded message fits in one block, so the actual message has to be somewhat smaller; for instance, in the case of MD4/MD5, the forged message has to be 447 bits or less.

The only generic short forgery attack we are aware of is the key-recovery attack using the birthday paradox which require $2^{n/2}$ queries and a time of 2^{n+1} (the generic forgery attack against iterated MAC produces at least a two-block forgery). On the other hand, short message forgeries are possible if the compression function is weak: using the partial key-recovery attacks of Contini and Yin to find k_2, one can compute a pair of short messages (M_1, M_2) such that $H_{k_2}(M_1) = H_{k_2}(M_2)$ and forge a MAC for one of them. This pair can be found using the birthday paradox in time $2^{n/2}$, or in the case of MD4, in time about 2^9 using the results of Section 3.4.4 to include the padding in the one-block message.

Obviously, this kind of attack is stronger than a PRF-attack against HMAC, and weaker than a universal forgery or a key-recovery. The following theorem proves that such short forgeries are as hard as a PRF attack against the compression function, *i.e.* it requires a time of the order of 2^n if there is no weakness in the compression function.

Theorem 4.1. *Let A_{NMAC} be a 1-block MAC-adversary against NMAC instantiated with the compression function h. Then, there exist PRF-adversaries A_1 and A_2 against h such that:*

$$\mathbf{Adv}_{\text{NMAC}}^{\text{1-MAC}}(A_{\text{NMAC}}) \leq \mathbf{Adv}_h^{\text{prf}}(A_1) + \mathbf{Adv}_h^{\text{prf}}(A_2) + 2^{-n+1}$$

Furthermore, A_1 and A_2 have the same time complexity as A_{NMAC}, and just make one extra oracle query.

Proof. We will build the adversary A_1 from A_{NMAC} by simulating the NMAC oracle. We are given a function g, and we try to guess if g is a random function or $h(K,\cdot)$ for some K. First we choose a random k_2, and we will then answer NMAC queries with $g(H_{k_2}(\cdot))$. A_{NMAC} will output some message M and a tag x, and we guess that g is a $f(K,\cdot)$ if and only if $x = g(H_{k_2}(M))$. Thus, we have:

- If g was a random function, $x = g(H_{k_2}(M))$ holds with a very low probability:
  - If we did not query the oracle g with the input $H_{k_2}(M)$ yet, then $g(H_{k_2}(M))$ is random and it will be equal to x with probability 2^{-n}.
  - If M collides under H_{k_2} with one of the q NMAC queries made by A_{NMAC}, this means we can build a PRF-adversary A_2 against h from A_{NMAC}: we answer NMAC queries using a random oracle for H_{k_1} and the given g' to compute the inner hash function (it will require one call to the oracle g' and some computations of f), then we output 1 if and only if $H_{k_2}(M) \in Q$, where Q denotes the set of H_{k_2} outputs computed by A_2. If

g was a random function, there is a negligible probability that we answer 1, and if g is some $h(K,\cdot)$ we will recognize it whenever A_{NMAC} succeeds to forge. Hence:

$$\begin{aligned}\Pr\left[A_2^{\$} \Rightarrow 1\right] &\leq 2^{-n}\\ \Pr\left[A_2^{f(K,\cdot)} \Rightarrow 1\right] &\geq \Pr\left[A_1^{\$} \Rightarrow 1, f(K,M) \in Q\right]\\ \Pr\left[A_1^{\$} \Rightarrow 1, f(K,M) \in Q\right] &\leq \mathbf{Adv}_h^{\text{prf}}(A_2) + 2^{-n}\end{aligned}$$

By combining these two cases, we have:

$$\begin{aligned}\Pr\left[A_1^{\$} \Rightarrow 1\right] &= \Pr\left[A_1^{\$} \Rightarrow 1, f(K,M) \in Q\right] + \Pr\left[A_1^{\$} \Rightarrow 1, f(K,M) \notin Q\right]\\ &\leq \mathbf{Adv}_h^{\text{prf}}(A_2) + 2^{-n} + 2^{-n}\end{aligned}$$

– If g is $h(K,\cdot)$, we will correctly output 1 if A_{NMAC} makes a correct forgery, and we finish the proof by combining the two cases:

$$\begin{aligned}\Pr\left[A_1^{f(K,\cdot)} \Rightarrow 1\right] &\geq \mathbf{Adv}_{\text{NMAC}}^{1-\text{MAC}}(A_{\text{NMAC}})\\ \mathbf{Adv}_{\text{NMAC}}^{1-\text{MAC}}(A_{\text{NMAC}}) &\leq \mathbf{Adv}_h^{\text{prf}}(A_1) + \mathbf{Adv}_h^{\text{prf}}(A_2) + 2^{-n+1}\end{aligned}$$

□

4.B Improving the MD4 IV-recovery

4.B.1 Reducing the Online Cost

First, we can easily lower the number of calls to the NMAC-oracle in the first phase of the IV-recovery. Instead of trying 22×2^{82} random message pairs, we will choose the messages more cleverly so that each message belongs to 22 pairs: we first choose 490 bits of the message at random and then use every possibility for the 22 remaining bits. Thus, we only need 2^{83} calls to the oracle instead of 22×2^{83}.

Note that we cannot use this trick in the composite IV-recovery attack, so the number of queries for the full key-recovery will only be halved (the queries for the basic IV-recovery for k_2 become negligible compared to the queries for the composite IV-recovery that will reveal k_2).

4.B.2 Reducing the Offline Cost

We may also lower the computational cost of the attack, by getting more than one bit of the IV once a collision has been found. This will require the extra assumption the colliding messages follow the differential path in step 1 (previously we only needed step 0), but this seems quite reasonable, for the same reasons. Out of the 22 paths used to learn IV bits, let p be the number of paths for which the condition holds, and a collision is actually found. From each message that collides following the differential path, we can also extract some conditions on the internal states Q_0 and Q_1. These states are not part of the IV, but since we know the message used, we can use these conditions to learn something on the IV. If we have a message pair that collides with $M' \boxminus M = 2^k$, we will call them $M^{(k)}$ and $M'^{(k)}$ and the condition gives us $Q_0^{[k+s_0]}(M^{(k)})$ and $Q_1^{[k+s_0]}(M^{(k)})$. The idea is the following (the symbol '➟' summarizes the number of bits to guess at each step):

1. We guess Q_{-1}. Let $w = |Q_{-1}|$ be the Hamming weight of Q_{-1}.

2. We compute 22 bits of Q_{-2} using the conditions on the IV, and we guess the others ➠ 10 bits.

3. We guess the bits of Q_{-3} used to compute Q_0. Since we have $Q_0 = (Q_{-4} \boxplus \mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxplus k_0 \boxplus m_0) \lll s_0$, we only need $Q_{-3}^{[i]}$ when $Q_{-1}^{[i]} = 0$ ➠ $32 - w$ bits.

4. We have $Q_{-4} = Q_0^{\ggg s_0} \boxminus \mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxminus m_0 \boxminus k_0$. If we use it with the message $M^{(0)}$ and take the equation modulo 2, it becomes: $Q_{-4}^{[0]} = Q_0^{[s_0]}(M^{(0)}) \boxminus (\mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxminus m_0^{(0)} \boxminus k_0) \bmod 2$, and it gives us $Q_{-4}^{[0]}$.

   Then, if we write the equation with $M^{(1)}$ and take it modulo 2, we will learn $Q_0^{[s_0]}(M^{(1)})$ from $Q_{-4}^{[0]}$. Since we know $Q_0(M^{(1)})^{\lll s_0} \bmod 4$ from $Q_0^{[s_0]}(M^{(1)})$ and $Q_0^{[1+s_0]}(M^{(1)})$, we can take the equation modulo 4 to learn $Q_{-4}^{[1]}$.

   By repeating this process, we learn the full Q_{-4}, but we need to guess the bit i when we do not have a message pair $M^{(i)}, M'^{(i)}$ ➠ $32 - p$ bits.

5. We apply the same process to compute the remaining bits of Q_{-3} using bits of Q_1. We already know n bits and we expect to be able to compute a ratio of $p/32$ of the missing ones. ➠ $q \cdot (1 - p/32)$ bits.

So, for each choice of Q_{-1} in step 1, we have to try a number of choices for the other bits that depends on the Hamming weight w of Q_{-1}. In the end, the number of keys to try is for this exhaustive search is:

$$C = \sum_{Q_{-1}} 2^{10+32-w+32-p+w(1-p/32)} = \sum_{Q_{-1}} 2^{74-2p+(32-w)p/32} = 2^{74-2p} \sum_{\overline{Q_{-1}}} 2^{|\overline{Q_1}|\cdot p/32}$$

We can easily compute sums of Hamming weight powers of a real α:

$$S_n(\alpha) = \sum_{x \in \mathbb{Z}_{2^n}} \alpha^{|x|} = \sum_{w=0}^{n} \binom{n}{w} \alpha^w = (1+\alpha)^n$$

This shows that the complexity is:

$$C = 2^{74-2p} \left(1 + 2^{p/32}\right)^{32}$$

With $p = 11$, this becomes a little less than 2^{90}, but the complexity depends on the number of conditions fulfilled by the key. If we assume that every condition has a probability of one half to hold, we can compute the average complexity for a random key:

$$\begin{aligned}
\langle C \rangle &= \frac{1}{\#k} \sum_k 2^{74-2p} \left(1 + 2^{p/32}\right)^{32} \\
&= \frac{2^{74}}{2^{22}} \sum_{k' \in \mathbb{Z}_{2^{22}}} 2^{-2|k'|} \left(1 + 2^{|k'|/32}\right)^{32} \\
&= 2^{52} \sum_{k' \in \mathbb{Z}_{2^{22}}} 2^{-2|k'|} \sum_{i=0}^{32} \binom{32}{i} 2^{|k'|i/32} \\
&= 2^{52} \sum_{i=0}^{32} \binom{32}{i} S_{22}\left(2^{i/32-2}\right) \\
&= 2^{52} \sum_{i=0}^{32} \binom{32}{i} \left(1 + 2^{i/32-2}\right)^{22} \\
&< 2^{93.8}
\end{aligned}$$

Hence, we have an IV-recovery attack requiring less than 2^{88} queries to the NMAC oracle, and less than 2^{94} offline hash computations.

Chapter 5

A Preimage Attack on MD4

5.1 Introduction

The previous chapters covered collision attacks on the MD4 family, and extensions of the collision attack to several applications. In this chapter we consider preimage resistance, and describe a preimage attack on MD4, which is the first preimage attack on a member of the MD4 family.

Preimage resistance is a weaker security notion than collision resistance, and it is expected to hold even for relatively weak functions. In particular MD4 has been broken by collision attacks since 1996 but it is still used in some applications where speed is important and/or a one-way function is needed but collision resistance is not important:

– to "encrypt" passwords in Windows NT and later (as the NTLM hash);
– for password derivation in the S/KEY one time password system [RFC1760];
– to compare file blocks in the incremental file transfer program rsync;
– for file identification and integrity in the eDonkey peer-to-peer network.

S/KEY and rsync even use a truncated MD4 and rely on the partial one-wayness of MD4.

Preimage attacks are rather rare in the world of hash function cryptanalysis; the most notable example is the preimage attack against MD2 by Muller [Mul04], later improved by Knudsen and Mathiassen [KM05]. The last version of this attack has a complexity of 2^{73} [KMMT10]. A preimage attack has much more impact than a collision attack: it can be used to fool integrity checks, to forge signatures using only known messages, to break "encrypted" password files, ... Moreover, when the hash function follows the Merkle-Damgård paradigm (as is the case for MD4) we can add any chosen prefix: given a message M and a target hash value $\overline{H}$, we can actually compute N such that $\text{MD4}(M||N) = \overline{H}$. For instance, this can be used to create a malicious software package with a given signature when trailing garbage is allowed (*e.g.* this is the case with zip, gzip, and bzip2 files).

5.1.1 Our results

Our main result is a preimage attack against MD4 with complexity 2^{102}. This attack uses messages of 18 blocks or slightly more (more precisely 9151 bits, about one kilobyte), and we can add any chosen prefix.

This is based on a partial pseudo-preimage attack on the compression function: we can choose 64 bits of the output (the other bits being randomly chosen by the preimage algorithm) and 32 bits of the input for the cost of 2^{32} compression function (brute force would require 2^{64}).

We also introduce a new technique to turn this attack into an attack on the hash function.

This technique requires some extra properties of the attack on the compression function, but is more efficient than the generic meet-in-the-middle approach.

Our attack uses many ideas from previous cryptanalysis of MD4 [Dob98a, Vau94, Dob98b, WLF+05]. We consider MD4 as a system of equations, we use some kind of differential path and use the Boolean functions to absorb some differences, we fix many values of the internal state using some particularities of the message expansion. The basic strategy used for our attack is the following:

1. Use degrees of freedom to fix some of the variables, in order to simplify the system of equations;
2. Express some output words in terms of the remaining free variables, and solve the equations.

5.1.2 Related work

Following the ideas used by Wang *et al.* for their collision attack on MD4, Yu *et al.* found a kind of second-preimage attack on MD4 [YWZW05]. This attack is based on collision differential paths with a relatively high probability of 2^{-62}. Using the message difference Δ of one those paths, $M + \Delta$ is a second preimage of M with probability 2^{-62}. Using several paths, one can build a second preimage with probability 2^{-56}. However this kind of attack is not what we usually call a second-preimage attack because it only works for a small subset of the message space. The attack has a complexity of one compression function, but for a given message it only works with probability 2^{-56}, and cannot be repeated when it fails. If we want to build an attack that works for any message using this property, we can use a brute-force search when the attack fails: it has a workload of 1 with probability 2^{-56}, and a workload of 2^{127} with probability $1 - 2^{-56}$; the average workload is still extremely close to 2^{127}. More interestingly, we can use this with long messages: if the message is made of 2^{63} blocks (there is no limitation to the size of the message in MD4, as opposed to SHA-1), we will be able to find a second-preimage for at least one of the blocks with a probability of $1 - \exp(-2^{63-56}) \approx 1 - 2^{-184}$. Thus, the cases where we have to run a brute-force search become negligible, and the average workload is just the time needed to test each block until a good one is found, so we expect it to be 2^{56}.

After the publication of our attack, Aoki and Sasaki described a preimage attack on MD4 using a meet-in-the-middle approach [AS08]. In this work, they divide the message expansion in two main chunks, such that some message words only appear in one of the chunks. Then, they start from the middle and they compute the step function independently in the two chunks using the free messages, in order to get a match. This technique is quite different from our attack, and gives a less efficient attack with complexity 2^{107}.

More recently the same technique has been used to attack MD5. In [SA09], Sasaki and Aoki obtained a preimage attack against MD5 with complexity 2^{124} (and 2^{117} for the compression function) using the meet-in-the-middle technique with new refinements.

5.1.3 Road map

In Section 5.2 we describe our attack against the compression function cMD4. Then in Section 5.3 we show how to extend it to an attack against the full MD4, with a better complexity than using the generic meet-in-the-middle attack.

5.1.4 Notations

We use the fact that we can fix some values of the internal state instead of only fixing values of the message words: since the step update function is invertible, when Q_{i-1}, Q_{i-2} and Q_{i-3}

are known, we can compute any one of Q_i, Q_{i-4} or m_i from the two others (see Algorithm 3.1 page 43 for explicit formulas).

In our attack we consider MD4 as a large system of equations over the variables Q_{-4} to Q_{47} and m_0 to m_{15} (we consider only words, and never look at individual bits); we have 48 step update equations, and 4 equations for the output values. We consider the input chaining variables IV_0, IV_1, IV_2, IV_3 as free: this makes the attack on the compression function easier (it is a pseudo-preimage attack), at the cost of some extra work to attack the full hash function. We use $\overline{X}$ to denote the desired value of a variable X, which is given as an input to our attack. The system we are trying to solve can be written as:

$$\begin{cases} Q_i = (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i} & \text{for } i \in \{0...47\} \\ H_0 = Q_{-4} \boxplus Q_{44} = \overline{H}_0 \\ H_1 = Q_{-3} \boxplus Q_{45} = \overline{H}_1 \\ H_2 = Q_{-2} \boxplus Q_{46} = \overline{H}_2 \\ H_3 = Q_{-1} \boxplus Q_{47} = \overline{H}_3 \end{cases}$$

To make the equations more readable, we introduce various colors to show the variables whose value has already been chosen in a previous step of the algorithm (the color itself is not very important).

We use $x^{[k]}$ to represent the $(k+1)$-th bit of x, that is $x^{[k]} = x^{\ggg k} \bmod 2$ (note that we count bits and steps starting from 0). We also use $\mathbb{0}$ and $\mathbb{1}$ to denote the 32-bit words whose bits are all zeroes and all ones (*i.e.* $\mathbb{0} =$ `0x00000000` and $\mathbb{1} =$ `0xffffffff`).

5.2 Pseudo-Preimage on the Compression Function

The general idea of the attack comes from a simple observation: we know that MD4 is very sensitive to differential attacks — there is a collision attack that costs less than 2 calls to the compression function [SWOK07]. However, in a preimage attack, we are looking for a single message and differential tools seem unsuitable for this task. Our idea is to start with an *initial message* (IV, M) with very specific properties, and to compute the digest cMD4(IV, M) of this message. Then we use differential paths to create a family of *related messages* $(IV^{(i)}, M^{(i)})$ so that the computation of cMD4$(IV^{(i)}, M^{(i)})$ differs from the computation of cMD4(IV, M) in a controlled way. This allows us to pick a particular $(IV^{(i_0)}, M^{(i_0)})$ in the family so that some bits of cMD4$(IV^{(i_0)}, M^{(i_0)})$ agree with a target value.

Following this idea, we look for a set of constraints on the initial message and a way to derive the related messages. We managed to easily select a related message that agrees with 32 bits of the target hash. We can compute 2^{32} good related messages for a cost of 2^{32} compression function calls: our attack has an amortized cost of 1. When we run it 2^{32} times, we can test 32 more bits of the target hash, and we expect to find a partial pseudo-preimage on 64 bits. Similarly, we expect to find a full pseudo-preimage after testing 2^{96} messages.

5.2.1 The Initial Message

The key part is to choose a set of constraints that allows to place many differential paths in MD4, in order to have as many related messages as possible. Instead of looking at single bits, we consider the 32-bit words of MD4, and try to have 32 paths at once. We use some properties of the Boolean functions in MD4, and some properties of the message expansion.

The Boolean functions used in the first and second round of MD4 have the nice property to be able to absorb some difference. This key property was used in early cryptanalysis of

Figure 5.1 Absorption of a difference. The step update function is:
$Q_i = (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i}$

1st round: m_4 and m_8 can be modified		2nd round: m_{20} and m_{24} can be modified	
$Q_2 = \mathbb{1}$		$Q_{18} = \mathbb{C}$	
$Q_3 = \mathbb{1}$		$Q_{19} = \mathbb{C}$	
Q_4		Q_{20}	
$Q_5 = \mathbb{0}$	$\mathsf{IF}(Q_4, Q_3, Q_2) = \mathbb{1}$	$Q_{21} = \mathbb{C}$	$\mathsf{MAJ}(Q_{20}, Q_{19}, Q_{18}) = \mathbb{C}$
$Q_6 = \mathbb{1}$	$\mathsf{IF}(Q_5, Q_4, Q_3) = \mathbb{1}$	$Q_{22} = \mathbb{C}$	$\mathsf{MAJ}(Q_{21}, Q_{20}, Q_{19}) = \mathbb{C}$
$Q_7 = \mathbb{1}$	$\mathsf{IF}(Q_6, Q_5, Q_4) = \mathbb{0}$	$Q_{23} = \mathbb{C}$	$\mathsf{MAJ}(Q_{22}, Q_{21}, Q_{20}) = \mathbb{C}$
Q_8	$\mathsf{IF}(Q_7, Q_6, Q_5) = \mathbb{1}$	Q_{24}	$\mathsf{MAJ}(Q_{23}, Q_{22}, Q_{21}) = \mathbb{C}$
$Q_9 = \mathbb{0}$	$\mathsf{IF}(Q_8, Q_7, Q_6) = \mathbb{1}$	$Q_{25} = \mathbb{C}$	$\mathsf{MAJ}(Q_{24}, Q_{23}, Q_{22}) = \mathbb{C}$
$Q_{10} = \mathbb{1}$	$\mathsf{IF}(Q_9, Q_8, Q_7) = \mathbb{1}$	$Q_{26} = \mathbb{C}$	$\mathsf{MAJ}(Q_{25}, Q_{24}, Q_{23}) = \mathbb{C}$
$Q_{11} = \mathbb{1}$	$\ldots$	$Q_{27} = \mathbb{C}$	$\ldots$

MD4 [Vau94, Dob98b] and is the starting point of the construction of differential paths [FLN07a] *à la* Wang. Notably, we have the following properties, for all $\mathbb{C}$, and for all x:

Absorb 1st input	Absorb 2nd input	Absorb 3rd input
$\mathsf{IF}(x, \mathbb{C}, \mathbb{C}) = \mathbb{C}$	$\mathsf{IF}(\mathbb{0}, x, \mathbb{C}) = \mathbb{C}$	$\mathsf{IF}(\mathbb{1}, \mathbb{C}, x) = \mathbb{C}$
$\mathsf{MAJ}(x, \mathbb{C}, \mathbb{C}) = \mathbb{C}$	$\mathsf{MAJ}(\mathbb{C}, x, \mathbb{C}) = \mathbb{C}$	$\mathsf{MAJ}(\mathbb{C}, \mathbb{C}, x) = \mathbb{C}$

In the attack, $\mathbb{C}$ will be set to a fixed constant, while x varies with the related message. Thanks to the absorption property, the outputs of the Boolean functions do not depend on x.

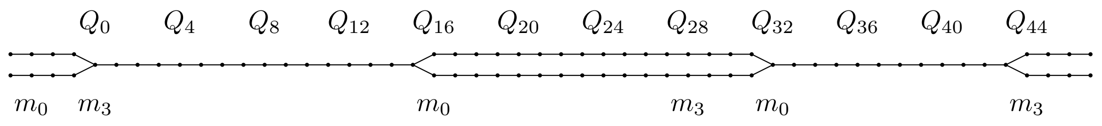
If we fix some internal state values Q_i's to specific values, we can control the propagation of differences in the message. For instance, if we fix Q_2, Q_3, Q_5, Q_6, Q_7, Q_9, Q_{10}, Q_{11} as shown in Figure 5.1, a modification of m_4 will only affect Q_4 and Q_8 (and states after step 12). Moreover, note that we can also modify m_8 while keeping the same state words fixed. Similarly in round 2, we can control the diffusion using absorption properties of the MAJ function, as shown in Figure 5.1.

In our attack, we use this property to create local collisions in the hash function. We introduce a difference at step i_0, and we cancel it step i_1, using constraints on the intermediate steps to avoid diffusion through the Boolean functions. The choice of i_0 and i_1 is guided by the message expansion: we want i_0 and i_1 to be quite far away so as to skip a significant part of the compression function and to maximize the number of free message words in between, but we do not want to use the same message word twice between i_0 and i_1 because that would result in incompatibilities when fixing values of the internal state. This leaves us with 3 possibilities:

– $(i_0, i_1) = (\ 0, 16)$: m_0 , m_4 , m_8 and m_{12} are free (used at steps 0, 4, 8, 12, 16)
– $(i_0, i_1) = (15, 31)$: m_{15}, m_{12}, m_{13} and m_{14} are free (used at steps 15, 19, 23, 27, 31)
– $(i_0, i_1) = (16, 32)$: m_0 , m_1 , m_2 and m_3 are free (used at steps 16, 20, 24, 28, 32)

Note that Vaudenay [Vau94] and Dobbertin [Dob98b] used the same basic idea in their attacks, with $(i_0, i_1) = (15, 31)$. Here we choose $(i_0, i_1) = (16, 32)$ because the free message words are used in the very beginning of the first round. Therefore, when modifying the free message words, the effect on the first round will only involve the first 4 steps of the compression function.

We now have a very good differential path with m_0 and m_3: if we consider the set of 2^{32} pairs that keep Q_{32} constant, their effect on the final hash only involves the first 4 steps and the last 4 steps of MD4. The other free variables m_1 and m_2 will be used to simplify the equations so as to make this path easier to use. Schematically, the differential path looks like this:



We can now choose what will be fixed by the initial message and what will be free for the related messages. The message words m_4 to m_{15} will be fixed by the initial message, while m_0 to m_3 are part of the related message. The internal state variables $Q_{14,15}$, $Q_{17,18,19}$, $Q_{21,22,23}$, $Q_{25,26,27}$, and $Q_{29,30}$ need to be equal and will be part of the initial message. Q_{13} is in the initial message because it is fixed by computing step 17 backwards, while Q_{31} is fixed by computing step 31 forwards. We also add Q_{12} in the initial message to fix the internal state of the first round. See Figure 5.2 for a graphical representation.

Figure 5.2 Initial message and related message

	Q_{-4}		🟥 Q_{12}		Q_{28}
	Q_{-3}		🟥 Q_{13}		🟥 $Q_{29} = \mathbb{C}$
	Q_{-2}		🟥 $Q_{14} = \mathbb{C}$		🟥 $Q_{30} = \mathbb{C}$
	Q_{-1}		🟥 $Q_{15} = \mathbb{C}$		🟥 Q_{31}
m_0	🟥 Q_0	m_0	Q_{16}	m_0	Q_{32}
m_1	🟥 Q_1	🟥 m_4	🟥 $Q_{17} = \mathbb{C}$	🟥 m_8	Q_{33}
m_2	🟥 Q_2	🟥 m_8	🟥 $Q_{18} = \mathbb{C}$	🟥 m_4	Q_{34}
m_3	🟥 Q_3	🟥 m_{12}	🟥 $Q_{19} = \mathbb{C}$	🟥 m_{12}	Q_{35}
🟥 m_4	🟥 Q_4	m_1	Q_{20}	m_2	Q_{36}
🟥 m_5	🟥 Q_5	🟥 m_5	🟥 $Q_{21} = \mathbb{C}$	🟥 m_{10}	Q_{37}
🟥 m_6	🟥 Q_6	🟥 m_9	🟥 $Q_{22} = \mathbb{C}$	🟥 m_6	Q_{38}
🟥 m_7	🟥 Q_7	🟥 m_{13}	🟥 $Q_{23} = \mathbb{C}$	🟥 m_{14}	Q_{39}
🟥 m_8	🟥 Q_8	m_2	Q_{24}	m_1	Q_{40}
🟥 m_9	🟥 Q_9	🟥 m_6	🟥 $Q_{25} = \mathbb{C}$	🟥 m_9	Q_{41}
🟥 m_{10}	🟥 Q_{10}	🟥 m_{10}	🟥 $Q_{26} = \mathbb{C}$	🟥 m_5	Q_{42}
🟥 m_{11}	🟥 Q_{11}	🟥 m_{14}	🟥 $Q_{27} = \mathbb{C}$	🟥 m_{13}	Q_{43}
🟥 m_{12}	🟥 Q_{12}	m_3	Q_{28}	m_3	Q_{44}
🟥 m_{13}	🟥 Q_{13}	🟥 m_7	🟥 $Q_{29} = \mathbb{C}$	🟥 m_{11}	Q_{45}
🟥 m_{14}	🟥 $Q_{14} = \mathbb{C}$	🟥 m_{11}	🟥 $Q_{30} = \mathbb{C}$	🟥 m_7	Q_{46}
🟥 m_{15}	🟥 $Q_{15} = \mathbb{C}$	🟥 m_{15}	🟥 Q_{31}	🟥 m_{15}	Q_{47}

🟥	fixed by the initial message
⬜	modified by the related messages

To select an initial message, we choose random values for $\mathbb{C}$, Q_{12}, Q_{13} and m_{15}; this allows us to compute Q_{31} and m_4 to m_{14} in the second round, and Q_0 to Q_{11} in the first round. Thus we can build 2^{128} different initial messages. Each initial message have 2^{128} related messages (by choosing the values of m_0, m_1, m_2, m_3).

5.2.2 The Related Messages

When an initial message is fixed, we have to choose m_0, m_1, m_2 and m_3 in a way that will give us some control on the hash value. We first isolate the third round from the second round, by choosing the value of Q_{32}. Then the choice of m_2, m_1 and m_3 will give the final state $Q_{44}, ...Q_{47}$, and m_0 will be chosen last: we expect that one value of m_0 will be consistent with the choice of Q_{32} when m_1, m_2, m_3 are fixed. Note that since m_0 is used in step 0, we can compute H_1, H_2

and H_3 without knowing m_0. Thus, we can choose a good value of m_2, m_1 and m_3 by looking only at the last round and the first 4 steps of the first round, and compute m_0 later in order to correct the second round.

We now study the first steps and the last steps. Our goal is to find efficiently a value of m_2, m_1, and m_3 that solves some of the equations. The message words m_2 and m_1 are used quite far for the last steps, so it will be hard to study how they affect the final state $Q_{44}, ...Q_{47}$: we only use m_3 to control the hash, while m_2 and m_1 are used to simplify the equations.

First Steps. We assume that an initial message has been chosen (in pink). Let us first study the initial steps of MD4:

$$Q_0 = (Q_{-4} \boxplus \mathsf{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxplus m_0)^{\lll 3} \tag{5.1}$$
$$Q_1 = (Q_{-3} \boxplus \mathsf{IF}(Q_0, Q_{-1}, Q_{-2}) \boxplus m_1)^{\lll 7} \tag{5.2}$$
$$Q_2 = (Q_{-2} \boxplus \mathsf{IF}(Q_1, Q_0, Q_{-1}) \boxplus m_2)^{\lll 11} \tag{5.3}$$
$$Q_3 = (Q_{-1} \boxplus \mathsf{IF}(Q_2, Q_1, Q_0) \boxplus m_3)^{\lll 19} \tag{5.4}$$

Equation (5.4) shows that Q_{-1} is completely determined by m_3. Additionally, we require than $Q_1 = \mathbb{1}$, as this simplify Equation (5.3) and make Q_{-2} completely determined by m_2, independently of Q_{-1}:

$$Q_2 = (Q_{-2} \boxplus Q_0 \boxplus m_2)^{\lll 11} \tag{5.3'}$$

We will see later how to satisfy the constraint $Q_1 = \mathbb{1}$.

Last Steps. Let us now study the final steps of MD4. We assume that a value has been chosen for Q_{32}, m_2, m_1: we can now compute Q_{32} to Q_{43} in the third round (in yellow), and Q_{-2} by equation (5.3'). This gives us $Q_{46} = \overline{H}_2 \boxminus Q_{-2}$ (in green).

$$Q_{44} = (Q_{40} \boxplus \mathsf{XOR}(Q_{43}, Q_{42}, Q_{41}) \boxplus m_3 \boxplus K_2)^{\lll 3} \tag{5.5}$$
$$Q_{45} = (Q_{41} \boxplus \mathsf{XOR}(Q_{44}, Q_{43}, Q_{42}) \boxplus m_{11} \boxplus K_2)^{\lll 9} \tag{5.6}$$
$$Q_{46} = (Q_{42} \boxplus \mathsf{XOR}(Q_{45}, Q_{44}, Q_{43}) \boxplus m_7 \boxplus K_2)^{\lll 11} \tag{5.7}$$
$$Q_{47} = (Q_{43} \boxplus \mathsf{XOR}(Q_{46}, Q_{45}, Q_{44}) \boxplus m_{15} \boxplus K_2)^{\lll 15} \tag{5.8}$$

Here we see that (5.7) gives the value $Q_{44} \oplus Q_{45}$. Moreover, we require $Q_{41} \boxplus m_{11} \boxplus K_2 = \mathbb{0}$ so as to simplify (5.6) (we explain later how to satisfy this constraint). We let V be $Q_{42} \oplus Q_{43} \oplus Q_{44} \oplus Q_{45}$, which is a known constant, and equation (5.6) becomes:

$$Q_{45} = \mathsf{XOR}(Q_{44}, Q_{43}, Q_{42})^{\lll 9}$$
$$Q_{45} = (Q_{45} \oplus V)^{\lll 9} \tag{5.6'}$$

This last equation is actually a system of linear equations over the bits of Q_{45}; it is easy to check whether it is satisfiable, and to compute the solutions (see Appendix 5.B for an optimization). From Q_{45}, we compute Q_{44} by (5.6) and m_3 by (5.5), and we know that this particular choice of m_3 gives the right value for Q_{46}, so that we have $H_2 = \overline{H}_2$.

Simplifications. We have introduced two extra conditions to simplify the equations:

$$Q_1 = \mathbb{1} \tag{C_1}$$
$$Q_{41} \boxplus m_{11} \boxplus K_2 = \mathbb{0}\,. \tag{C_2}$$

Algorithm 5.1 Partial Pseudo Preimage

```
Input: $\overline{H}_0, \overline{H}_2, \overline{IV}_2$
Output: $M, IV$ st. $H_0 = \overline{H}_0$, $H_2 = \overline{H}_2$ and $IV_2 = \overline{IV}_2$
Running Time: $2^{32}$
 1: loop                                                        ▷ Expect 1 iteration
 2:     Choose an initial message with $Q_1 = \mathbb{1}$                          ▷ $2^{96}$ possibilities
 3:     for all $Q_{32}$ do                                              ▷ $2^{32}$ iterations
 4:         Compute $Q_{33}, Q_{34}, Q_{35}$.
 5:         Choose $m_2$ so that $Q_{-2} = \overline{IV}_2$ (using Eq (5.3')).        ▷ $Q_{-2}$ is $IV_2$
 6:         Compute $Q_{36}, Q_{37}, Q_{38}, Q_{39}$.
 7:         Choose $m_1$ so that $Q_{41} = -m_{11} - K_2$ (using Eq (5.9) and (5.10)).
 8:         Compute $Q_{40}, Q_{41}, Q_{42}, Q_{43}$.
 9:         Choose $m_3$ so that $Q_{46} = \overline{H}_2 \boxminus Q_{-2}$ (solving Eq (5.6')).   ▷ $Q_{46} \boxplus Q_{-2}$ is $H_2$
10:         Compute $Q_{44}, Q_{45}, Q_{46}, Q_{47}$, and $Q_{-1}, Q_{-2}, Q_{-3}$.
11:         Choose $m_0$ so that $Q_{-4} = \overline{H}_0 \boxminus Q_{44}$ (using Eq (5.1)).      ▷ $Q_{44} \boxplus Q_{-4}$ is $H_0$
12:         if $m_0$ matches $Q_{32}$ at step 32 then                     ▷ OK with prob. $2^{-32}$
13:             return
14:         end if
15:     end for
16: end loop
```

(C_1) can only be satisfied statistically, by computing about 2^{32} initial messages and keeping the good ones, but this cost can be amortized when an initial message is used with many different choices of Q_{32}, m_2, and m_1. On the other hand, (C_2) can be satisfied by choosing an appropriate m_1 when Q_{32} and m_2 have been chosen:

$$Q_{40} = (Q_{36} \boxplus \mathsf{XOR}(Q_{39}, Q_{38}, Q_{37}) \boxplus m_1 \boxplus K_2)^{\lll 3} \tag{5.9}$$

$$Q_{41} = (Q_{37} \boxplus \mathsf{XOR}(Q_{40}, Q_{39}, Q_{38}) \boxplus m_9 \boxplus K_2)^{\lll 9} \tag{5.10}$$

The choice of Q_{41} following (C_2) gives Q_{40} by (5.10), which gives m_1 by (5.9). Conversely, with this choice of m_1 (C_2) will be satisfied. Every initial message can now be used with 2^{64} choices of Q_{32} and m_2, so we still have some extra degree of freedom: we can use the freedom of m_2 to choose the value of Q_{-2} using Equation (5.3'). In the end we can choose both Q_{-2} and Q_{46} (hence H_2) for an amortized cost of one compression function.

Partial Pseudo-preimage. When we put this all together, we can compute m_3 so that $H_2 = \overline{H}_2$ for an amortized cost of 1 compression function, and we can also choose IV_2. The full algorithm, given in Algorithm 5.1, is a partial pseudo-preimage attack, which is 2^{32} times more efficient than exhaustive search. It should be repeated about 2^{64} times to find a full pseudo-preimage, and we have enough different initial messages for that. Note that Algorithm 5.1 finds pseudo-preimages on (H_0, H_2), but if we change a little bit the end of the algorithm we can have pseudo-preimages on (H_1, H_2) or (H_2, H_3) just as easily. See Appendix 5.A for an example of a partial pseudo-preimage.

5.3 Preimage of the Full MD4

To extend this attack to the full MD4, we use an idea similar to the unbalanced meet-in-the-middle attack of Lai and Massey [LM92]. We compute many pseudo-preimages of $\overline{H}$, we hash many random messages, and we use the birthday paradox to meet in the middle. If we have a pseudo-preimage attack with complexity 2^s, the generic attack uses the pseudo-preimage attack $2^{(n-s)/2}$ times starting from the target digest $\overline{H}$ (we assume there is no problem with the padding), and hashes $2^{(n+s)/2}$ random blocks, starting from the standard IV. Thanks to the birthday paradox, we expect one match. This gives a preimage attack with complexity $2^{1+(n+s)/2}$. In our case this would be 2^{113} (we have $s = 96$), but we show how to use some specific properties of our pseudo-preimage attack to build a preimage attack with complexity only 2^{102}.

5.3.1 The Padding

First, we need to handle the padding in the last block. When looking for a padded message of b blocks, we use a message length of $512b - 65$ bits so that there is as little padding as possible. The last block is correctly padded if and only if $m_{15} = b/2^{32} (= 0)$, $m_{14} = 512b - 65$, and $m_{13}^{[0]} = \mathtt{1}$. The condition on m_{15} is easy to satisfy since we can choose m_{15} in the initial message, but on the other hand m_{14} depends only on $\mathbb{C}$:

$$\begin{aligned} Q_{27} &= (Q_{23} \boxplus \mathsf{MAJ}(Q_{26}, Q_{25}, Q_{24}) \boxplus m_{14} \boxplus k_{27})^{\lll 13} \\ \mathbb{C} &= (\mathbb{C} \boxplus \mathbb{C} \boxplus m_{14} \boxplus K_1)^{\lll 13} \\ m_{14} &= \mathbb{C}^{\ggg 13} \boxminus \mathbb{C} \boxminus \mathbb{C} \boxminus K_1 \end{aligned}$$

Thus, we just run an exhaustive search over $\mathbb{C}$, and we expect to find one value that gives the correct m_{14}. Similarly, we have $m_{13} = \mathbb{C}^{\ggg 13} \boxminus \mathbb{C} \boxminus \mathbb{C} \boxminus K_1 = m_{14}$, so the condition $m_{13}^{[0]} = 1$ will be satisfied.

Note that we cannot use a message size that is a multiple of 8 this way since $m_{14} = m_{13}$ is used both as the padding and as the length. If we really need to use a message made of bytes and not of bits, we can build a second-preimage attack by reusing the last block of the original message (and keeping the same padding).

When searching for the last block, we only have 2^{32} initial messages available. We will not choose the value of IV_2, but keep m_2 as a degree of freedom. Each initial message can be used to compute 2^{64} related messages with $H_2 = \overline{H}_2$. There is probability of $1 - 1/e \approx 0.63$ that at least one of these messages gives the full correct hash, so we might have to repeat this step a few times. The extra freedom will come from the message length: if we change the number of blocks b, this gives us a new m_{14} and a new $\mathbb{C}$, and we can try again to find a suitable padding block.

We start with $b = 34$ or $b = 18$ (see next section), and increase b until we find a padding block. Note that some values of b gives no suitable value of $\mathbb{C}$, but this is not a problem. Additionally, if one wants to choose a prefix for the preimage attack, one should start with a larger b.

5.3.2 Layered Hash Tree

To improve over the basic meet-in-the-middle attack, we use an extra property of our pseudo-preimage attack on the compression function: we need a workload of 2^s (in our case, $s = 96$) to find a pseudo-preimage of a single target value, but if we have a set of k target values (with some extra conditions on the set), we can find a pseudo-preimage of *one of them* in time $2^s/k$. This is because our pseudo-preimage attack is based on the repetition of a partial pseudo-preimage attack, where the remaining bits are random. Thus, we can find $2k$ pseudo-preimages in time 2^{s+1}, and if we can also make sure that the pseudo-preimage set satisfy the extra conditions, we can iterate this operation. We start with a set $\mathcal{H}_0$ of size 1, and after $n - s$ iterations we have a set $\mathcal{H}_{n-s}$ of size 2^{n-s}, which we use for the unbalanced meet-in-the-middle. The resulting structure is shown in Figure 5.3. In the end, we will find a preimage in time $2(n-s)2^s + 2^s$, using a memory of size $\mathcal{O}(2^{n-s})$.

A similar idea based on multi-target pseudo-preimage was used by Mendel and Rijmen to attack HAS-V [MR07]. In that attack, they could run a multi-target pseudo-preimage attack on a set of size 2^s (this is not possible in our case), and this result in an attack with time complexity 2^{s+1} and a memory requirement of $\mathcal{O}(2^s)$.

Our attack against MD4 can be used as a multi-target pseudo-preimage attack following Algorithm 5.2, if the target set $\mathcal{H}$ satisfies the following extra properties:

- $\{\overline{H}_2 : \overline{H} \in \mathcal{H}\}$ is a singleton: a single m_3 can be used for the whole set;
- $|\mathcal{H}| \leq 2^{64}$: the loop of line 11 is negligible.

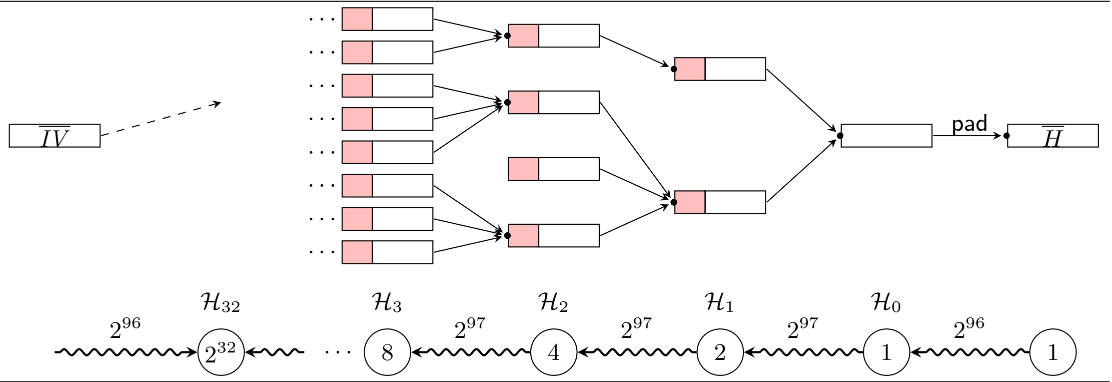
Since our algorithm allows us to choose the value of $\overline{IV}_2$ in the pseudo-preimages, we can build the pseudo-preimage set so that the extra conditions are still satisfied.

Algorithm 5.2 Multi-Target Pseudo Preimage

Input: $\overline{IV}_2$, $\overline{H}_2$ and a target set $\mathcal{H}$ st. $\forall \overline{X} \in \mathcal{H}, \overline{X}_2 = \overline{H}_2$.
Output: preimage set $\mathcal{I}$ st. $\forall (IV, M) \in \mathcal{I},\ F(IV, M) \in \mathcal{H}$ and $IV_2 = \overline{IV}_2$.
Running Time: 2^{97}

```
 1: while $|\mathcal{I}| < 2|\mathcal{H}|$ do                          ▷ Expect $2^{65}$ iterations
 2:     Choose an initial message with $Q_1 = \mathbb{1}$              ▷ $2^{96}$ possibilities
 3:     for all $Q_{32}$ do                                            ▷ $2^{32}$ iterations
 4:         Compute $Q_{33}, Q_{34}, Q_{35}$.
 5:         Choose $m_2$ so that $Q_{-2} = \overline{IV}_2$.           ▷ $Q_{-2}$ is $IV_2$
 6:         Compute $Q_{36}, Q_{37}, Q_{38}, Q_{39}$.
 7:         Choose $m_1$ so that $Q_{41} = -m_{11} - K_2$.
 8:         Compute $Q_{40}, Q_{41}, Q_{42}, Q_{43}$.
 9:         Choose $m_3$ so that $Q_{46} = \overline{H}_2 \boxminus Q_{-2}$.   ▷ $Q_{46} \boxplus Q_{-2}$ is $H_2$
10:         Compute $Q_{44}, Q_{45}, Q_{46}, Q_{47}$, and $Q_{-1}, Q_{-2}, Q_{-3}$.
11:         for all $\overline{H} \in \mathcal{H}$ st. $H_3 = \overline{H}_3$, $H_4 = \overline{H}_4$ do   ▷ Expect $2^{-64}|\mathcal{H}|$ values
12:             Choose $m_0$ so that $Q_{-4} = \overline{H}_0 \boxminus Q_{44}$.   ▷ $Q_{44} \boxplus Q_{-4}$ is $H_0$
13:             if $m_0$ matches $Q_{32}$ then                         ▷ OK with prob. $2^{-32}$
14:                 Add the solution to $\mathcal{I}$
15:             end if
16:         end for
17:     end for
18: end while
```

Figure 5.3 Preimage attack on the full MD4 with a layered hash tree.



The tree costs $32 \times 2^{97} = 2^{102}$ to build, and gives 2^{32} pseudo-preimages of $\overline{H}$; this is significantly better than the generic attack which find 2^t pseudo-preimages in time 2^{96+t}. In the forward phase, we compute the hashes of 2^{96} random messages, and we expect to find a match thanks to the birthday paradox. The full preimage search has time complexity 2^{102}, and require a memory of about 2^{33} message blocks to store the tree (we do not have to store the 2^{96} hashes in the forward phase).

Note that we do not need fast memory to store the tree. Indeed, all the values in $\mathcal{H}_{32}$ share the same value $\overline{H}_2$, so we only have to search 2^{64} values in the set out of the 2^{96} random chaining values of the forward phase.

Tweaking the Tree. We can tweak the parameters of the tree so as to slightly improve the attack. First, instead of doubling the size of the set at each iteration, we can triple it (the length of the preimage starts from 23, and the cost of the attack is about $2^{101.92}$) or quadruple it (the length of the preimage starts from 18, and the cost of the attack is about 2^{102}).

We can also replace the layered tree by another structure. We start with a set of 1 target value, and every time we find a pseudo-preimage of one element of the set, we add it to the set. The first pseudo-preimage costs 2^{96}, the second one $2^{96}/2$, then $2^{96}/3$ and so on... the set will reach a size of 2^{32} after an expected workload of:

$$2^{96} \sum_{k=1}^{2^{32}} \frac{1}{k} \leq 2^{96}(\ln 2^{32} + 1) \leq 2^{100.54}.$$

The total attack ends up with a complexity smaller than $2^{100.6}$ Note that in this case, we do not control the length of the preimage, so we have to use an expendable message [Dea99, KS05] in the forward phase.

Conclusion

Our attack on MD4 is still theoretical due to the high complexity, but it shows that MD4 is even weaker than we thought. Our attack relies on the absorption property of some of the Boolean functions, and exploits the message expansion. It is the first preimage attack on the full MD4 and it is much less efficient than Dobbertin's attack on a two round version [Dob98b].

We did not find any direct application of the attack on the compression function, but constructions relying on the partial one-wayness of cMD4 should be carefully analysed: our attack might be practical depending on the exact assumptions made on cMD4.

This attack reduces the security margin of other members of the MD4, but it is not a direct threat. The features introduced in later members of the family make the attack unsuitable:

- The rounds function of MD5, SHA-1, and SHA-2 have a much better diffusion that MD4 due to the summation of Q_{i-1} to compute Q_i (we cannot absorb a difference);
- The number of rounds is more important;
- The message expansion in the SHA family is much harder to control.

5.A A Partial Pseudo-Preimage of MD4

Here is an example of a partial pseudo-preimage of MD4. We ran our algorithm with $\overline{H}_1 = 0, \overline{H}_2 = 0$, and $\overline{IV}_2 = 0$. It would cost 2^{64} hash evaluations to find this message by exhaustive search, but our algorithm finds it in about 20 minutes on a desktop computer.

IV			
`72 fa 31 aa`	`a0 6e 27 95`	`00 00 00 00`	`13 c9 dc ce`
Message block			
`8e 34 9e ad`	`6c 36 1e 1c`	`21 b7 0e bd`	`14 1e 98 d9`
`79 67 c3 19`	`d7 3c 6a 19`	`d7 3c 6a 19`	`d7 3c 6a 19`
`14 61 85 33`	`14 61 85 33`	`14 61 85 33`	`14 61 85 33`
`58 13 27 05`	`58 13 27 05`	`58 13 27 05`	`17 46 57 27`
MD4			
`34 5e 59 ae`	`c5 6a 3b 8e`	`00 00 00 00`	`00 00 00 00`

The internal state variables for this message are given by:

Q_{-4}=`0xaa31fa72`	Q_{-3}=`0xcedcc913`	Q_{-2}=`0x00000000`	Q_{-1}=`0x95276ea0`
Q_0 =`0x1545809d`	Q_1 =`0xffffffff`	Q_2 =`0xa1bdf692`	Q_3 =`0x1a9925ec`
Q_4 =`0xa8473548`	Q_5 =`0x92125811`	Q_6 =`0x9b4aaa1d`	Q_7 =`0x00a73054`
Q_8 =`0x6ef7f38b`	Q_9 =`0xa3789cab`	Q_{10} =`0x3de0878e`	Q_{11} =`0x2f9cbd24`
Q_{12} =`0x0ffc6391`	Q_{13} =`0x1e2a88f4`	Q_{14} =`0x1e83b396`	Q_{15} =`0x1e83b396`
Q_{16} =`0xb5062a71`	Q_{17} =`0x1e83b396`	Q_{18} =`0x1e83b396`	Q_{19} =`0x1e83b396`
Q_{20} =`0x51547062`	Q_{21} =`0x1e83b396`	Q_{22} =`0x1e83b396`	Q_{23} =`0x1e83b396`
Q_{24} =`0x3b4aa594`	Q_{25} =`0x1e83b396`	Q_{26} =`0x1e83b396`	Q_{27} =`0x1e83b396`
Q_{28} =`0x6f4786bc`	Q_{29} =`0x1e83b396`	Q_{30} =`0x1e83b396`	Q_{31} =`0x24db97dc`
Q_{32} =`0x84d9f63d`	Q_{33} =`0xc9a584fe`	Q_{34} =`0x475e7886`	Q_{35} =`0x508d517f`
Q_{36} =`0x79ca3034`	Q_{37} =`0x3bd701b4`	Q_{38} =`0x980fef11`	Q_{39} =`0x9784cf50`
Q_{40} =`0xc8f3a1b1`	Q_{41} =`0x5da0b34b`	Q_{42} =`0x5fa99919`	Q_{43} =`0x2d166b40`
Q_{44} =`0x042763c2`	Q_{45} =`0x312336ed`	Q_{46} =`0x00000000`	Q_{47} =`0xf913fc25`

Note that MD4 uses a little-endian convention to convert a sequence of bytes to a sequence of words, and that the order of the words in the *IV* and in the hash is not the same as in the internal state.

5.B Solving the Equation $x = (x \oplus V)^{\lll 9}$

In Section 5.2.2 we find that Q_{45} has to be the solution of the following equation:

$$x = (x \oplus V)^{\lll 9}. \tag{5.11}$$

where V is a constant that depends on the choices made on the previous steps of the algorithm. We have to solve this equation 2^{96} times for each pseudo-preimage, so we want to solve it very efficiently (it should cost less that one evaluation of cMD4).

We can write this equation as a linear system over the bits of x and V:

$$(5.11) \iff \begin{cases} x^{[\ 0]} = x^{[23]} \oplus V^{[23]} \\ x^{[\ 1]} = x^{[24]} \oplus V^{[24]} \\ \dots \\ x^{[31]} = x^{[22]} \oplus V^{[22]} \end{cases}$$

And we can express each bit of x as a function of $x^{[0]}$ and V:

$$\iff \begin{cases} x^{[\ 9]} = x^{[0]} \oplus V^{[0]} \\ x^{[18]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \\ \ldots \\ x^{[23]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \cdots \oplus V^{[14]} \\ x^{[\ 0]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \cdots \oplus V^{[14]} \oplus V^{[23]} \end{cases}$$

The system is consistent if and only if the last equation holds, *i.e.* $\bigoplus_{i=0}^{31} V^{[i]} = 0$. In this case we have a first solution x_0 given by

$$\begin{cases} x_0^{[\ 0]} = 0 \\ x_0^{[\ 9]} = V^{[0]} \\ x_0^{[18]} = V^{[0]} \oplus V^{[9]} \\ \ldots \\ x_0^{[23]} = V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \cdots \oplus V^{[14]} \end{cases}$$

and a second solution $x_1 = x_0 \oplus \mathbb{1}$. Note that the expression of the bits of x_0 is linear in the bits of V: $x_0 = \varphi(V)$. This can be used for a very efficient implementation. We split V into 4 bytes, $V = V_3||V_2||V_1||V_0$, and we precompute 4 tables (each one contains 256 words):

$$T_0[u] = \varphi(0||0||0||u) \qquad T_1[u] = \varphi(0||0||u||0) \qquad T_2[u] = \varphi(0||u||0||0) \qquad T_3[u] = \varphi(u||0||0||0)$$

Then we have $x_0 = \varphi(V) = T_0[V_0] \oplus T_1[V_1] \oplus T_2[V_2] \oplus T_3[V_3]$. We can solve the equation, with only:

- the computation of the parity of the Hamming weight of V;
- 4 table look-ups when there is a solution.

Part Two

The SIMD Hash Function

Anyone can design a cipher that he himself can't break. *Bruce Schneier*

Abstract

Following the devastating attacks against many widely used hash functions as described in the previous chapters, NIST has organized the SHA-3 competition to select and standardize a new hash function. This competition is similar to the AES competition held in 1998-2000, and attracts worldwide attention, with a large effort underway to assess the security of the candidates. The competition was launched in 2008 and 64 designs were submitted by research teams from academia and industry, including the most active teams in symmetric key cryptography. 14 candidates have been selected for the second round, and the competition is expected to end in 2012. During my PhD, I designed the hash function SIMD which has been submitted to the competition. In this chapter I describe the design of SIMD and present some security analysis.

SIMD is a new design following the Merkle-Damgård and Davies-Meyer constructions. The most important component of SIMD is its message expansion, which is designed to give a high minimal distance. This allows to prove that SIMD is secure against differential cryptanalysis. The security proof shows that the inner block cipher is resistant to *related key* attacks, which is the most likely kind of attack when used in a hash function. This is in contrast to the security proofs of AES-based designs which are based on analysis of block ciphers as used with a random and unknown key.

SIMD features a small scale parallelism, inside the compression function. This can be used to write an efficient implementation with vector instructions (SIMD). Such instructions are available on many widely used architectures: *SSE* on x86, *Altivec* on PowerPC, *IwMMXt* on ARM. The optimized implementation of SIMD-256 (using SIMD instructions) take 11 cycles per byte on one core of a Core2 processor, which is faster than SHA-2 and about 80% of the throughput of SHA-1. Moreover, the compression function of SIMD can be parallelized on two cores for improved performance.

SIMD has been selected for the second round of the SHA-3 competition.

Contents

Chapter 6

Design of SIMD

6.1 Introduction

This chapter describes the SIMD family of hash functions. This family is based on two functions SIMD-256 and SIMD-512; we define SIMD-n with $n \leq 256$ as a truncation of SIMD-256, and SIMD-n with $256 < n \leq 512$ as a truncation of SIMD-512. Each function SIMD-n takes as input a message of arbitrary size, and outputs a digest of n bits.

6.1.1 Features of SIMD

The SIMD hash function is quite similar to members of the MD/SHA family. It is based on a familiar Merkle-Damgård design, where the compression function is built from a Feistel-like block cipher in Davies-Meyer mode. However there are several innovations in this design: the internal state is twice as large as the output size (following the Chop-MD construction), it uses a strong message expansion, and the compression function uses a modified feed-forward. The main design criterion was to follow the MD/SHA designs principle which are quite well understood, and to add security features to avoid known attacks.

Security. We believe that the internal block cipher in a hash function does not have the same security requirement than a block cipher used to encrypt a message. In particular, the block cipher inside a Davies-Meyer hash function should be secure under related key attacks. In SIMD, we use a message expansion with a very high minimal distance, which can still be computed quite efficiently. Thanks to this proved minimal distance, we can prove that SIMD is secure against differential cryptanalysis.

Our security proof essentially shows that the implicit block cipher in SIMD is resistant to *related key* attacks. Contrary to most security proofs based on block cipher techniques, we do not assume that the key is unknown to the adversary. This is particularly interesting in light of recent attacks on the AES [BK09]: the AES is designed to resist differential attacks, but it can be in fact attacked in a related key setting.

Parallelism. SIMD features a small scale parallelism. The compression function itself can be parallelized to some extent. This can be used to improve hardware efficiency, and allows an efficient software implementation using SIMD instructions. The fact that about half the time required to compute the hash function is spent in the message expansion also allows a second level of parallelism: the message expansion of the message block $i + 1$ can be computed while the Feistel part is compressing the message block i.

Performance. The performances of SIMD are very good on high-end desktop computers. SIMD-256 only needs 11 cycles per byte on one core of a Core2 processor (the reference platform from NIST), and we can go down to 6 cycles per byte if we use two cores. More generally, SIMD is efficient on architectures which include a set of SIMD instructions.

On the other hand, it should also be noted than a fast implementation of the SIMD hash function has to use SIMD instructions, and cannot be written in pure C. Similarly, the performances are not very good if there is no SIMD support on the target platform.

6.2 Algorithm Specification and Rationale

6.2.1 Mathematical Preliminaries and Notations

The design of SIMD uses a number of different operations with useful mathematical properties. In this section, we introduce the operations that will be used through this document, and detail their properties.

The Field $\mathbb{F}_{257}$. Since 257 is a prime, the ring $\mathbb{Z}_{257}$ of the integers modulo 257 is a field $\mathbb{F}_{257}$. The operations in this field are denoted with $\pmod{257}$. This field was chosen because we can easily map a byte to an element of the field, and the operations in $\mathbb{F}_{257}$ can be computed efficiently both in software and in hardware.

The Number-Theoretic Transform. The Number-theoretic transform of size n in $\mathbb{F}_{257}$ is defined as:

$$\begin{aligned} &\mathsf{NTT}_n : \mathbb{F}_{257}^n \to \mathbb{F}_{257}^n \\ &\qquad (x_j)_{j=0}^{n-1} \mapsto (y_i)_{i=0}^{n-1} : \quad y_i = \sum_{j=0}^{n-1} x_j \omega^{ij} \pmod{257}. \end{aligned}$$

where $n \leq 256$, and ω is a n-th root of unity in $\mathbb{F}_{257}$. We can see it as a polynomial evaluation: if the sequence $(x_j)_{j=0}^{n-1}$ is interpreted as a polynomial $P(X) = \sum_{j=0}^{n-1} x_j X^j$, then we have $y_i = P(\omega^i)$. The NTT is a bijection of $\mathbb{F}_{257}^n$.

This transform is identical to the Discrete Fourier Transform but it operates on a finite field instead of the field of complex numbers. It can be computed efficiently by the same algorithm as the Fast Fourier Transform, which has a complexity of $\mathcal{O}(n \log n)$ field operations.

The Rings $\mathbb{Z}_{2^{16}}$ and $\mathbb{Z}_{2^{32}}$. $\mathbb{Z}_{2^{16}}$ denotes the ring of integers modulo 2^{16}, and $\mathbb{Z}_{2^{32}}$ denotes the ring of the integers modulo 2^{32}. We use $\boxplus$ and $\boxtimes$ to represent the modular addition and multiplication in these rings. (Actually, we only use $\boxplus$ in $\mathbb{Z}_{2^{32}}$ and $\boxtimes$ in $\mathbb{Z}_{2^{16}}$).

We can represent elements of $\mathbb{Z}_{2^{16}}$ by 16-bit words, and elements of $\mathbb{Z}_{2^{32}}$ by 32-bit words. Thus, we define the following bitwise Boolean functions on 32-bit words:

$$\begin{aligned} \mathsf{IF}(A, B, C) &= (A \wedge B) \ \vee (\neg A \wedge C) \\ \mathsf{MAJ}(A, B, C) &= (A \wedge B) \ \vee (A \wedge C) \vee (B \wedge C) \end{aligned}$$

where $\vee$ denotes the Boolean OR, $\wedge$ denotes AND, and $\neg$ denotes NOT. We also use $\oplus$ for the exclusive or (XOR). IF acts as a conditional, and MAJ is the majority function. These functions are already used in many hash functions because they have good properties: the output is unbiased, and no input bit has a linear effect on the output.

We also use bitwise rotations on 32-bit words: x rotated to the left by s bits is denoted by $x^{\lll s}$.

Superscripts and Subscripts. Since the compression function consists of the repetition of a simple round function, we use $X^{(i)}$ to denote the variable X associated with round i. Meanwhile, many variables can be seen as vectors, and we use $X_{[0..k]}$ to denote the vector $[X_0, X_1, ...X_k]$ (or its transpose, depending on the context).

6.2.2 Description of the Algorithm

SIMD is an iterative hash function that follows the Merkle-Damgård design. The main component of a Merkle-Damgård hash function F is a compression function $f : \{0,1\}^p \times \{0,1\}^m \to \{0,1\}^p$. To compute $F(M)$, the message M is first divided into k chunks m_i of m bits. The compression function is iterated to update an internal state x by compressing the message chunks and the internal state: $x_i = f(x_{i-1}, m_i)$. There is a padding rule to fill the last m-bit block, and the padding usually includes the message size (this is known as the Merkle-Damgård strengthening). The initial value of the internal state x_{-1} is called IV and is fixed in the description of the hash function. The output of the hash function is given by computing a finalization function $g : \{0,1\}^p \to \{0,1\}^n$ on the last internal state x_{k-1}.

The Davies-Meyer mode is a common way to build a compression function f from a block cipher E: it is defined as $f(x,m) = E_m(x) \oplus x$. Many hash functions use a custom block cipher, designed with a message expansion step, and a Feistel ladder.

The SIMD family uses a similar design, and the size parameters are as follows:

	Output size n	Message block size m	Internal state size p
SIMD-256	256	512	512
SIMD-512	512	1024	1024

The inner state is represented as a matrix of 32-bit words. For SIMD-256, it is a 4×4 matrix, while SIMD-512 has a 8×4 inner state:

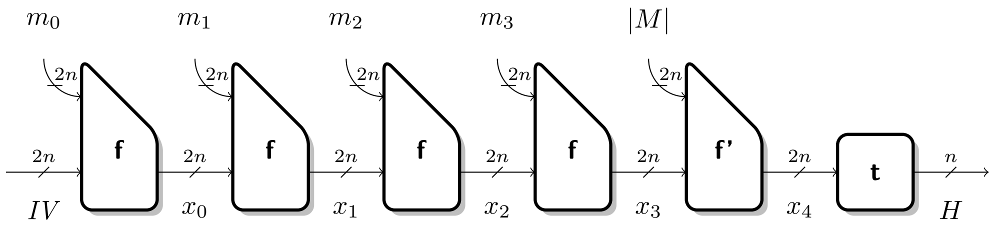
$$\mathcal{S}_{256} = \begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \end{bmatrix} \qquad \mathcal{S}_{512} = \begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \\ A_4 & B_4 & C_4 & D_4 \\ A_5 & B_5 & C_5 & D_5 \\ A_6 & B_6 & C_6 & D_6 \\ A_7 & B_7 & C_7 & D_7 \end{bmatrix}$$

In this section, we will describe more precisely the operating mode of SIMD, and the inside of the compression function: the message expansion and the Feistel ladder.

Mode of Operation.

Iteration. Our mode of operation is similar to the wide-pipe construction of Lucks [Luc05] and to Chop-MD [CDMP05]. The internal state is twice as large as the output, and we use a truncation T to compute the hash value from the last internal state. The padding rule is quite simple: the last message block is filled with zeros if it is smaller than m bits, and an extra block containing the size of the message in bits is added. This extra block is compressed with a slightly modified compression function C', and the output is truncated. This is described by Figure 6.1.

Figure 6.1 The iteration used in SIMD



$x_{-1} = IV$ $\quad x_i = f(x_{i-1}, m_i)$ $\quad H = t(f'(x_{k-1}, |M|))$

The size of the message input to SIMD is not limited, and the number of bits of the message included in the last block is taken modulo 2^m. We believe that it is not necessary to limit the message size in the description of the algorithm, but for all practical purpose it can be considered to be below 2^{64}. Therefore, the message input in the last message block is quite constrained. The last compression function C' acts as kind of blank round, and makes it harder to use the truncation to find a collision.

Modified Davies-Meyer. To build our compression function from a Feistel-like block cipher, we use a technique similar to the well-known Davies-Meyer construction, but with a few variations.

First, we have a message block size that is equal to the internal state size, so we can use $f(x, m) = E_m(x \oplus m) \oplus x$ instead of $f(x, m) = E_m(x) \oplus x$. This is construction 8 from [BRS02] and construction 41 from [PGV93]. It enjoys the same provable security guarantees than the original Davies-Meyer construction. Note that this is to be expected, because the former can be seen as a special case of the later, with a block cipher E' defined as $E'_k(x) = E_k(x \oplus k)$. The fact that $h \oplus m$ goes into the block cipher means that the adversary has to "commit" to a given value of m before starting to evaluate E_m. This prevents, for example, to construct the message m "on the fly", and complicates message modification techniques.

Second, instead of using a simple XOR to combine $E_m(h \oplus m)$ and h, we use a few extra Feistel rounds, with h entering as the key. This makes a function $P : \{0,1\}^p \times \{0,1\}^p \to \{0,1\}^p$, and the compression function is defined as $P(h, E_m(h \oplus m))$. The security proofs of the Davies-Meyer mode can easily be adapted to the modified Davies-Meyer because P is a quasigroup operation:

– for all y, $\Big(x \mapsto P(x, y)\Big)$ is a permutation;

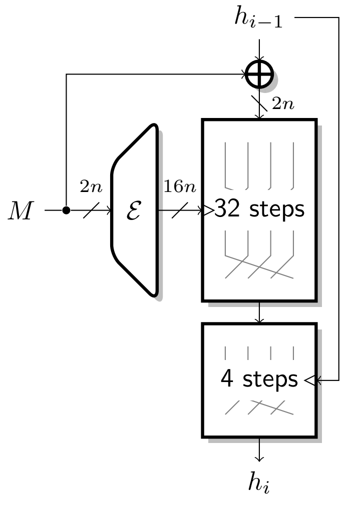
– for all x, $\Big(y \mapsto P(x, y)\Big)$ is a permutation;

– for all z, $\Big(x \mapsto y \text{ s.t. } P(x, y) = z\Big)$ is a permutation;

Moreover, this modified mode prevents some kind of multi-block attacks, and does not allow to find trivial fixed points used in some second preimage attacks [Dea99]. Our modified Davies-Meyer mode is described in Figure 6.2.

The Message Expansion. The message expansion is a very important part of our design. Looking back at the SHA/MD family, most attacks exploit the weak message expansion of these functions. As seen in Part I, collision and preimage attacks use the fact that one can modify a small part of the expanded message without too much effect on other parts of the full expanded message. Therefore, we choose to view the message expansion as an error correcting code, and we try to build a code with a high minimal distance. This is similar to the approach of [JP06], but

Figure 6.2 Modified Davies-Meyer



our message expansion is very different from the ones used in MD/SHA hash functions.

Our message expansion is composed of three layers, which can each be considered as a code in some vector spaces. For SIMD-256 (resp. SIMD-512), it expands a 512-bit (resp. 1024-bit) message block into a 4096-bit (resp. 8192-bit) expanded message, with a minimal distance of 520 (resp. 1032).

	Message block	Expanded message	Minimal distance
SIMD-256	512 bits	4096 bits	520 bits
SIMD-512	1024 bits	8192 bits	1032 bits

First Layer: Number-Theoretic Transform. The first layer of the message expansion is computationally expensive, but it is an important part of our design. The basic idea is to consider the message as a polynomial P of degree 63 (resp. 127) in $\mathbb{F}_{257}[X]$, and to evaluate this polynomial over 128 (resp. 256) points of the field $\mathbb{F}_{257}$ using a Number-Theoretic Transform [1]. This is essentially a punctured Reed-Solomon code, and it has an optimal minimal distance: two different polynomials will match on at most 63 (resp. 127) points. It reaches the Singleton bound, and therefore is a linear MDS code.

However, this code has some unwanted properties, that would allow to build very specific expanded messages:

– The Reed-Solomon code is cyclic: for any polynomial P, if $(y_i)_{i=0}^{n-1} = \mathsf{NTT}_n(P(X))$ and $(z_i)_{i=0}^{n-1} = \mathsf{NTT}(P(\omega X))$ with ω a n-th root of the unity, then $z_i = y_{i+1 \pmod n}$.

– The NTT of a constant polynomial k is uniform ($\forall i,\ y_i = k$). In particular, $\mathsf{NTT}(0) = 0$.

To avoid those properties, we will actually compute the NTT of the polynomial $P + X^{127}$ (resp. $P + X^{255}$). This is equivalent to adding some constants (actually the NTT of X^{127} or X^{255}) to the NTT of P. This makes the code affine, instead of just linear.

1. The NTT is a bijection, but we use it with half of the input set to zero, which makes it an injection.

More precisely, the first message expansion step of SIMD-256 is defined as:

$$\begin{aligned} O : (\mathbb{Z}_{2^8})^{64} &\to (\mathbb{F}_{257})^{128} \\ (x_j)_{j=0}^{63} &\mapsto (y_i)_{i=0}^{127} : \quad y_i = \sum_{j=0}^{63} x_j \alpha^{ij} + \alpha^{127i} \pmod{257}. \end{aligned}$$

where $\alpha = 139$ is a 128th root of unity in $\mathbb{F}_{257}$. To map the input x_j from $\mathbb{Z}_{2^8}$ to $\mathbb{F}_{257}$, we consider them as integers between 0 and 255 (note that we actually use a subset of the Reed-Solomon code because we do not use the value 256).

For SIMD-512, the first message expansion step is defined as:

$$\begin{aligned} O : (\mathbb{Z}_{2^8})^{128} &\to (\mathbb{F}_{257})^{256} \\ (x_j)_{j=0}^{127} &\mapsto (y_i)_{i=0}^{255} : \quad y_i = \sum_{j=0}^{127} x_j \beta^{ij} + \beta^{255i} \pmod{257}. \end{aligned}$$

where $\beta = 41$ is a 256th root of unity in $\mathbb{F}_{257}$, and a square root of α.

This affine code has the same *differential* properties than the Reed-Solomon code: two different messages will match on at most 63 (resp. 127) y_i's. Moreover, we still have a good number of non-zero *values*. Let us consider SIMD-256, and an i such that $y_i = 0$. Then we have

$$\begin{aligned} y_i = 0 &= P(\alpha^i) + \alpha^{127i} \\ 0 &= \alpha^i P(\alpha^i) + 1. \end{aligned}$$

α^i is a root of $X.P + 1$, which is a polynomial of degree 64. Thus there are at least 64 non-zero y_i's in SIMD-256. Similarly, we can show there are at least 128 non-zero y_i's in SIMD-512

Second Layer: Concatenated Code. In order to output a sequence of bytes (rather than elements of $\mathbb{F}_{257}$) and to increase the minimal distance of our message expansion, each symbol of $O(M)$ will be encoded through an inner code $I : \mathbb{F}_{257} \to \mathbb{Z}_{2^{16}}$. We use a class of very efficient codes, implemented with only a single multiplication modulo 2^{16}: $I_C : x \mapsto C \boxtimes x$ for some constant C. We ran an exhaustive search over the constant C, and we found four values that give a minimal Hamming distance of 4 bits: $C = 185$, $C = 233$, and their opposites. Thus, we use the two following inner codes:

$$\begin{aligned} I_{185} : \mathbb{F}_{257} &\to \mathbb{Z}_{2^{16}} \\ x &\mapsto 185 \boxtimes \widetilde{x} \qquad \text{where } -128 \le \widetilde{x} \le 128 \text{ and } \widetilde{x} = x \pmod{257} \\ I_{233} : \mathbb{F}_{257} &\to \mathbb{Z}_{2^{16}} \\ x &\mapsto 233 \boxtimes \widetilde{x} \qquad \text{where } -128 \le \widetilde{x} \le 128 \text{ and } \widetilde{x} = x \pmod{257} \end{aligned}$$

The input x is lifted from $\mathbb{F}_{257}$ to the integers with $-128 \le \widetilde{x} \le 128$ because lifting to $\{-128, ...128\}$ is easier than to $\{0, ...257\}$. We use both $I_{185}(O(M))$ and $I_{233}(O(M))$ in the expanded message (*i.e.*, we have two copies of $O(M)$ encoded through two different inner codes).

Theses codes also have a minimal distance of 4 when we measure the weight of the Non Adjacent Form (NAF) of the modular difference. The NAF is an optimal signed binary representation, so this means that from a modular difference point of view, the code also has a distance of 4. We cannot express the 4 Hamming difference with less than 4 modular differences.

Third Layer : Permutation. The expanded message will be used as a sequence of 32-bit words, so we have to pack pairs of 16-bit words together. The 32-bit word with $I_C(x)$ in his lower 16 bits and $I_C(y)$ in its higher 16 bits is denoted by $I_C(x, y)$. If $I_C(x)$ and $I_C(y)$ are seen as integers between 0 and $2^{16} - 1$, we have $I_C(x, y) = I_C(x) + 2^{16} I_C(y)$.

To make the message expansion stronger we permute the message words so that if an adversary wants to cancel some expanded message words, he has to choose them quite far away. We first define an intermediate 32×4 (resp. 32×8) matrix of 32-bit words. For SIMD-256, we have (with $0 \le j \le 3$):

$$Z_j^{(i)} = \begin{cases} I_{185}\big(y[8i+2j], & y[8i+2j+1]\big) \quad \text{when } 0 \le i \le 15 \\ I_{233}\big(y[8i+2j-128], & y[8i+2j-64]\big) \quad \text{when } 16 \le i \le 23 \\ I_{233}\big(y[8i+2j-191], & y[8i+2j-127]\big) \quad \text{when } 24 \le i \le 31 \end{cases}$$

For SIMD-512, we have (with $0 \le j \le 7$):

$$Z_j^{(i)} = \begin{cases} I_{185}\big(y[16i+2j], & y[16i+2j+1]\big) \quad \text{when } 0 \le i \le 15 \\ I_{233}\big(y[16i+2j-256], & y[16i+2j-128]\big) \quad \text{when } 16 \le i \le 23 \\ I_{233}\big(y[16i+2j-383], & y[16i+2j-255]\big) \quad \text{when } 24 \le i \le 31 \end{cases}$$

Finally, we permute the lines of the matrix Z. Let $W_j^{(i)} = Z_j^{(\Pi(i))}$ with the following permutation:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\Pi(i)$	4	6	0	2	7	5	3	1	15	11	12	8	9	13	10	14

i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\Pi(i)$	17	18	23	20	22	21	16	19	30	24	25	31	27	29	28	26

The full message expansion for SIMD-256 is given in Table 6.1.

The Feistel Ladder. The compression function is based on a Feistel structure, with a step function similar to the step functions of the MD/SHA family:

$$\begin{aligned} A_j^{(i)} &= \left(D_j^{(i-1)} \boxplus W_j^{(i)} \boxplus \phi^{(i)}(A_j^{(i-1)}, B_j^{(i-1)}, C_j^{(i-1)})\right)^{\lll s^{(i)}} \boxplus A_{p^{(i)}(j)}^{(i-1)}{}^{\lll r^{(i)}} \\ B_j^{(i)} &= A_j^{(i-1)}{}^{\lll r^{(i)}} \\ C_j^{(i)} &= B_j^{(i-1)} \\ D_j^{(i)} &= C_j^{(i-1)} \end{aligned}$$

where $\phi^{(i)}$ is a Boolean function, $\boxplus$ is the addition modulo 2^{32} and $\lll s^{(i)}$ denotes rotation to the left by $s^{(i)}$ bits. This step function is shown in Figure 6.5. Note that all the values used to compute the new $A_j^{(i+1)}$'s go through a rotation. That should prevent differential trails active only on the most-significant bit, as was found in MD5 [dBB93].

We basically have 4 parallel Feistel ladders for SIMD-256 (resp. 8 for SIMD-512), and they interact together through the permutations $p^{(i)}$'s. At each round, a new value is computed in each Feistel ladder, and this new value is sent to another Feistel ladder at the following round. The $p^{(i)}$'s are chosen to ensure a good diffusion. For SIMD-256, we define the following permutations:

Figure 6.3 The message expansion of SIMD (sizes are shown for SIMD-256)

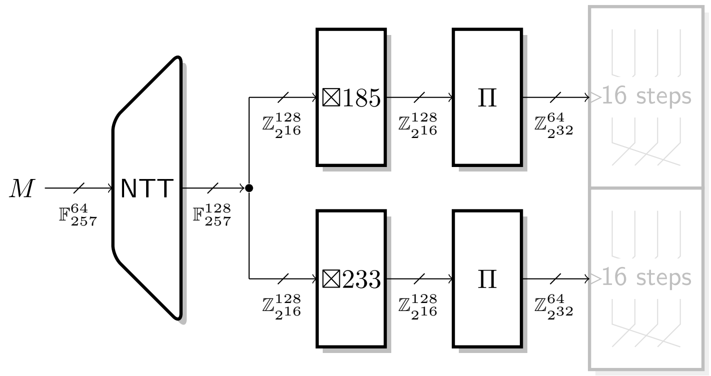
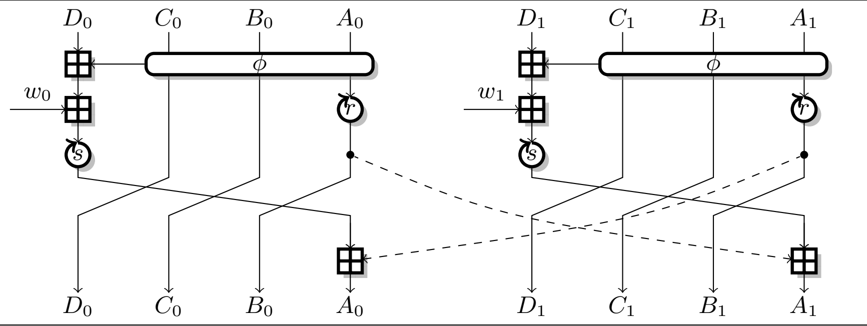


Table 6.1 Full Permutations for SIMD-256

i	$W_0^{(i)}$	$W_1^{(i)}$	$W_2^{(i)}$	$W_3^{(i)}$
0	$I_{185}(y_{32}, y_{33})$	$I_{185}(y_{34}, y_{35})$	$I_{185}(y_{36}, y_{37})$	$I_{185}(y_{38}, y_{39})$
1	$I_{185}(y_{48}, y_{49})$	$I_{185}(y_{50}, y_{51})$	$I_{185}(y_{52}, y_{53})$	$I_{185}(y_{54}, y_{55})$
2	$I_{185}(y_{0}, y_{1})$	$I_{185}(y_{2}, y_{3})$	$I_{185}(y_{4}, y_{5})$	$I_{185}(y_{6}, y_{7})$
3	$I_{185}(y_{16}, y_{17})$	$I_{185}(y_{18}, y_{19})$	$I_{185}(y_{20}, y_{21})$	$I_{185}(y_{22}, y_{23})$
4	$I_{185}(y_{56}, y_{57})$	$I_{185}(y_{58}, y_{59})$	$I_{185}(y_{60}, y_{61})$	$I_{185}(y_{62}, y_{63})$
5	$I_{185}(y_{40}, y_{41})$	$I_{185}(y_{42}, y_{43})$	$I_{185}(y_{44}, y_{45})$	$I_{185}(y_{46}, y_{47})$
6	$I_{185}(y_{24}, y_{25})$	$I_{185}(y_{26}, y_{27})$	$I_{185}(y_{28}, y_{29})$	$I_{185}(y_{30}, y_{31})$
7	$I_{185}(y_{8}, y_{9})$	$I_{185}(y_{10}, y_{11})$	$I_{185}(y_{12}, y_{13})$	$I_{185}(y_{14}, y_{15})$
8	$I_{185}(y_{120}, y_{121})$	$I_{185}(y_{122}, y_{123})$	$I_{185}(y_{124}, y_{125})$	$I_{185}(y_{126}, y_{127})$
9	$I_{185}(y_{88}, y_{89})$	$I_{185}(y_{90}, y_{91})$	$I_{185}(y_{92}, y_{93})$	$I_{185}(y_{94}, y_{95})$
10	$I_{185}(y_{96}, y_{97})$	$I_{185}(y_{98}, y_{99})$	$I_{185}(y_{100}, y_{101})$	$I_{185}(y_{102}, y_{103})$
11	$I_{185}(y_{64}, y_{65})$	$I_{185}(y_{66}, y_{67})$	$I_{185}(y_{68}, y_{69})$	$I_{185}(y_{70}, y_{71})$
12	$I_{185}(y_{72}, y_{73})$	$I_{185}(y_{74}, y_{75})$	$I_{185}(y_{76}, y_{77})$	$I_{185}(y_{78}, y_{79})$
13	$I_{185}(y_{104}, y_{105})$	$I_{185}(y_{106}, y_{107})$	$I_{185}(y_{108}, y_{109})$	$I_{185}(y_{110}, y_{111})$
14	$I_{185}(y_{80}, y_{81})$	$I_{185}(y_{82}, y_{83})$	$I_{185}(y_{84}, y_{85})$	$I_{185}(y_{86}, y_{87})$
15	$I_{185}(y_{112}, y_{113})$	$I_{185}(y_{114}, y_{115})$	$I_{185}(y_{116}, y_{117})$	$I_{185}(y_{118}, y_{119})$
16	$I_{233}(y_{8}, y_{72})$	$I_{233}(y_{10}, y_{74})$	$I_{233}(y_{12}, y_{76})$	$I_{233}(y_{14}, y_{78})$
17	$I_{233}(y_{16}, y_{80})$	$I_{233}(y_{18}, y_{82})$	$I_{233}(y_{20}, y_{84})$	$I_{233}(y_{22}, y_{86})$
18	$I_{233}(y_{56}, y_{120})$	$I_{233}(y_{58}, y_{122})$	$I_{233}(y_{60}, y_{124})$	$I_{233}(y_{62}, y_{126})$
19	$I_{233}(y_{32}, y_{96})$	$I_{233}(y_{34}, y_{98})$	$I_{233}(y_{36}, y_{100})$	$I_{233}(y_{38}, y_{102})$
20	$I_{233}(y_{48}, y_{112})$	$I_{233}(y_{50}, y_{114})$	$I_{233}(y_{52}, y_{116})$	$I_{233}(y_{54}, y_{118})$
21	$I_{233}(y_{40}, y_{104})$	$I_{233}(y_{42}, y_{106})$	$I_{233}(y_{44}, y_{108})$	$I_{233}(y_{46}, y_{110})$
22	$I_{233}(y_{0}, y_{64})$	$I_{233}(y_{2}, y_{66})$	$I_{233}(y_{4}, y_{68})$	$I_{233}(y_{6}, y_{70})$
23	$I_{233}(y_{24}, y_{88})$	$I_{233}(y_{26}, y_{90})$	$I_{233}(y_{28}, y_{92})$	$I_{233}(y_{30}, y_{94})$
24	$I_{233}(y_{49}, y_{113})$	$I_{233}(y_{51}, y_{115})$	$I_{233}(y_{53}, y_{117})$	$I_{233}(y_{55}, y_{119})$
25	$I_{233}(y_{1}, y_{65})$	$I_{233}(y_{3}, y_{67})$	$I_{233}(y_{5}, y_{69})$	$I_{233}(y_{7}, y_{71})$
26	$I_{233}(y_{9}, y_{73})$	$I_{233}(y_{11}, y_{75})$	$I_{233}(y_{13}, y_{77})$	$I_{233}(y_{15}, y_{79})$
27	$I_{233}(y_{57}, y_{121})$	$I_{233}(y_{59}, y_{123})$	$I_{233}(y_{61}, y_{125})$	$I_{233}(y_{63}, y_{127})$
28	$I_{233}(y_{25}, y_{89})$	$I_{233}(y_{27}, y_{91})$	$I_{233}(y_{29}, y_{93})$	$I_{233}(y_{31}, y_{95})$
29	$I_{233}(y_{41}, y_{105})$	$I_{233}(y_{43}, y_{107})$	$I_{233}(y_{45}, y_{109})$	$I_{233}(y_{47}, y_{111})$
30	$I_{233}(y_{33}, y_{97})$	$I_{233}(y_{35}, y_{99})$	$I_{233}(y_{37}, y_{101})$	$I_{233}(y_{39}, y_{103})$
31	$I_{233}(y_{17}, y_{81})$	$I_{233}(y_{19}, y_{83})$	$I_{233}(y_{21}, y_{85})$	$I_{233}(y_{23}, y_{87})$

Figure 6.4 Step update function of SIMD



SIMD-256:
4 parallel
Feistels.
SIMD-512:
8 parallel
Feistels.

$$p^{(0)}(j) = j \oplus 1$$
$$p^{(1)}(j) = j \oplus 2$$
$$p^{(2)}(j) = j \oplus 3$$

j	0	1	2	3
$p^{(0)}(j)$	1	0	3	2
$p^{(1)}(j)$	2	3	0	1
$p^{(2)}(j)$	3	2	1	0

The permutation used at step i is $p^{(i \bmod 3)}$. If a difference is introduced in one Feistel at round i, it will affect all the Feistels at round $i+2$.

For SIMD-512, we define seven permutations:

$$p^{(0)}(j) = j \oplus 1$$
$$p^{(1)}(j) = j \oplus 6$$
$$p^{(2)}(j) = j \oplus 2$$
$$p^{(3)}(j) = j \oplus 3$$
$$p^{(4)}(j) = j \oplus 5$$
$$p^{(5)}(j) = j \oplus 7$$
$$p^{(6)}(j) = j \oplus 4$$

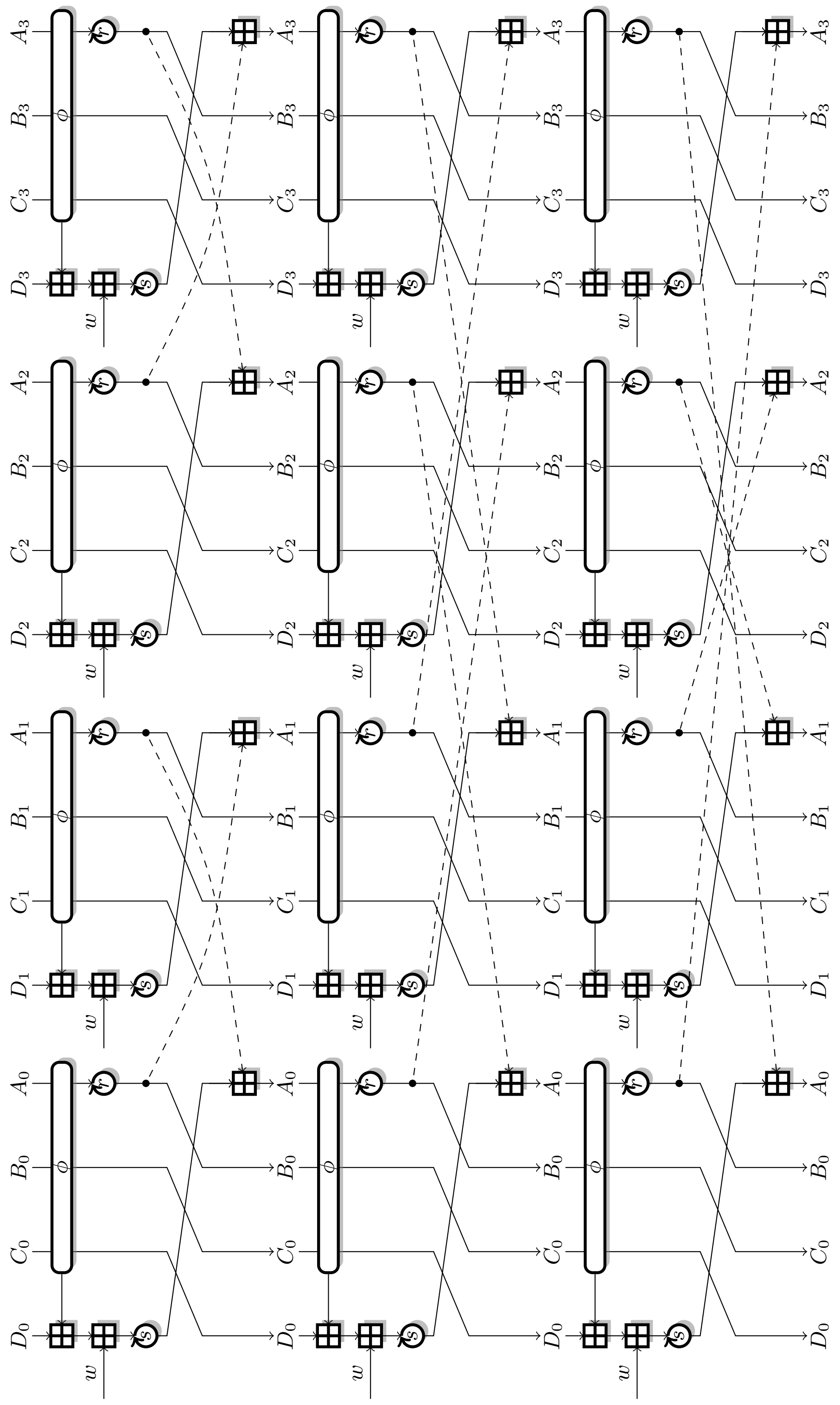
j	0	1	2	3	4	5	6	7
$p^{(0)}(j)$	1	0	3	2	5	4	7	6
$p^{(1)}(j)$	6	7	4	5	2	3	0	1
$p^{(2)}(j)$	2	3	0	1	6	7	4	5
$p^{(3)}(j)$	3	2	1	0	7	6	5	4
$p^{(4)}(j)$	5	4	7	6	1	0	3	2
$p^{(5)}(j)$	7	6	5	4	3	2	1	0
$p^{(6)}(j)$	4	5	6	7	0	1	2	3

The permutation used at step i is $p^{(i \bmod 7)}$. If a difference is introduced in one Feistel at round i, it will affect all the Feistels at round $i+3$. Figure 6.5 shows the repetition of a few round functions, with the permutations of SIMD-256.

Alternatively, we can write an equivalent description of the step update involving only the A_j registers:

$$\begin{aligned}
B_j^{(i)} &= A_j^{(i-1)^{\lll r_{i-1}}} \\
C_j^{(i)} &= A_j^{(i-2)^{\lll r_{i-2}}} \\
D_j^{(i)} &= A_j^{(i-3)^{\lll r_{i-3}}} \\
A_j^{(i)} &= \left(A_j^{(i-4)^{\lll r_{i-4}}} \boxplus W_j^{(i)} \boxplus \phi^{(i)}(A_j^{(i-1)}, A_j^{(i-2)^{\lll r_{i-2}}}, A_j^{(i-3)^{\lll r_{i-3}}})\right)^{\lll s^{(i)}} \boxplus A_{p^{(i)}(j)}^{(i-1)^{\lll r^{(i)}}}
\end{aligned}$$

Figure 6.5 A few step updates of SIMD-256



Formally, the step update function of SIMD-256 is:

$$\mathsf{Step}\left(\begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \end{bmatrix}, \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}, \phi, r, s, p\right)$$
$$= \begin{bmatrix} (D_0 \boxplus W_0 \boxplus \phi(A_0, B_0, C_0))^{\lll s} \boxplus A_{p(0)}^{\lll r} & A_0^{\lll r} & B_0 & C_0 \\ (D_1 \boxplus W_1 \boxplus \phi(A_1, B_1, C_1))^{\lll s} \boxplus A_{p(1)}^{\lll r} & A_1^{\lll r} & B_1 & C_1 \\ (D_2 \boxplus W_2 \boxplus \phi(A_2, B_2, C_2))^{\lll s} \boxplus A_{p(2)}^{\lll r} & A_2^{\lll r} & B_2 & C_2 \\ (D_3 \boxplus W_3 \boxplus \phi(A_3, B_3, C_3))^{\lll s} \boxplus A_{p(3)}^{\lll r} & A_3^{\lll r} & B_3 & C_3 \end{bmatrix} \tag{6.1}$$

and the step update function of SIMD-512 is:

$$\mathsf{Step}\left(\begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \\ A_4 & B_4 & C_4 & D_4 \\ A_5 & B_5 & C_5 & D_5 \\ A_6 & B_6 & C_6 & D_6 \\ A_7 & B_7 & C_7 & D_7 \end{bmatrix}, \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \end{bmatrix}, \phi, r, s, p\right)$$
$$= \begin{bmatrix} (D_0 \boxplus W_0 \boxplus \phi(A_0, B_0, C_0))^{\lll s} \boxplus A_{p(0)}^{\lll r} & A_0^{\lll r} & B_0 & C_0 \\ (D_1 \boxplus W_1 \boxplus \phi(A_1, B_1, C_1))^{\lll s} \boxplus A_{p(1)}^{\lll r} & A_1^{\lll r} & B_1 & C_1 \\ (D_2 \boxplus W_2 \boxplus \phi(A_2, B_2, C_2))^{\lll s} \boxplus A_{p(2)}^{\lll r} & A_2^{\lll r} & B_2 & C_2 \\ (D_3 \boxplus W_3 \boxplus \phi(A_3, B_3, C_3))^{\lll s} \boxplus A_{p(3)}^{\lll r} & A_3^{\lll r} & B_3 & C_3 \\ (D_4 \boxplus W_4 \boxplus \phi(A_4, B_4, C_4))^{\lll s} \boxplus A_{p(4)}^{\lll r} & A_4^{\lll r} & B_4 & C_4 \\ (D_5 \boxplus W_5 \boxplus \phi(A_5, B_5, C_5))^{\lll s} \boxplus A_{p(5)}^{\lll r} & A_5^{\lll r} & B_5 & C_5 \\ (D_6 \boxplus W_6 \boxplus \phi(A_6, B_6, C_6))^{\lll s} \boxplus A_{p(6)}^{\lll r} & A_6^{\lll r} & B_6 & C_6 \\ (D_7 \boxplus W_7 \boxplus \phi(A_7, B_7, C_7))^{\lll s} \boxplus A_{p(7)}^{\lll r} & A_7^{\lll r} & B_7 & C_7 \end{bmatrix} \tag{6.2}$$

A block of eight steps, is called a round, and it is parametrized by a set of rotation constants $\pi_{[0..3]}$. The Boolean functions and the rotation constants are used as follows:

i mod 8	$\phi^{(i)}$	$r^{(i)}$	$s^{(i)}$
0	IF	π_0	π_1
1	IF	π_1	π_2
2	IF	π_2	π_3
3	IF	π_3	π_0
4	MAJ	π_0	π_1
5	MAJ	π_1	π_2
6	MAJ	π_2	π_3
7	MAJ	π_3	π_0

Round	π_0	π_1	π_2	π_3
0	3	23	17	27
1	28	19	22	7
2	29	9	15	5
3	4	13	10	25

The whole compression function is made of 4 rounds, plus four final steps to mix the initial chaining value to the initial state (this is our feed-forward). The feed-forward steps use the Boolean function IF and the same rotation constants as the last round. A pseudo-code description of the full SIMD hash function is given in Algorithm 6.1.

The Final Compression Function. After all the message blocks have been compressed, there is an extra call to the compression function, with the message length as input. The message length is counted in bits, modulo 2^m if needed. It is written as a sequence of bytes using the little endian convention, *i.e.* the low order byte of the counter will be the first message byte.

For this final compression function, we use a slightly different message expansion, with a tweaked outer code. In SIMD-256, instead of using $O(M) = \mathsf{NTT}_{128}(M + X^{127})$, we use $O'(M) = \mathsf{NTT}_{128}(M + X^{127} + X^{125})$. In SIMD-512, instead of using $O(M) = \mathsf{NTT}_{256}(M + X^{255})$, we use $O'(M) = \mathsf{NTT}_{256}(M + X^{255} + X^{253})$. The range of this modified message expansion is distinct from the range of the main message expansion. Alternatively, we can consider that the compression function takes an extra input bit, and that the message is encoded in a prefix-free way by setting the extra bit only in the final block.

6.2.3 Rationale

The SIMD hash function follows the spirit of the MD/SHA family, but it should be protected against known attacks on members of this family.

Iteration Mode. We believe that Merkle-Damgård construction is now well understood, thanks to years of study. Its main problem is the extension attack which is easily fixed with a finalisation function. Moreover, there are several attacks related to the size of the inner state:

- the second preimage attack on long messages of Kelsey and Schneier [KS05];
- the multi-collision attack of Joux [Jou04];
- the herding attack of Kelsey and Kohno [KK06];
- MAC attacks based on collisions [PvO95];
- various meet-in-the-middle attacks: building expandable messages from fixed point[Dea99], preimages on the hash function from preimages on the compression function.

These attacks show than the Merkle-Damgård iteration does not perform as a perfect function above the birthday bound. SIMD uses a wide pipe to avoid these attacks. Moreover, we use modified compression function for the last block, (which is equivalent to a prefix-free encoding of the message).

Modified Davies-Meyer Mode. The Davies-Meyer mode is also well studied, and suffers the following problems:

- It is easy to find fixed-points, which can be used to build expandable messages. For any given message M, $E_M^{-1}(0)$ is a fixed point .
- In collision attacks, the feed-forward makes it quite easy to transform pseudo-collision into collisions. If we have a linear characteristic that gives a message difference Δ, we can use two non-linear characteristic to build a differential path $0 \rightsquigarrow \Delta$ and $\Delta \rightsquigarrow \Delta$ in the Feistel part. As seen in Figure 3.2 page 47, this allows to find a collision when the input difference Δ cancels the output difference Δ in the feed-forward. This property was used by Wang *et al.* to break MD5 [WY05] and SHA-1 [WYY05b].

Our non-linear feedback should avoid these attacks.

However, it was pointed out by Gauravaram and Bagheri [GB09] that the modified Davies-Meyer used in SIMD is still easy to differentiate from a random function. Indeed, one can use a technique very similar to the fixed-point attack on the regular Davies-Meyer:

- Start with a state $\mathcal{S}^{(31)}$ (the output of the block cipher) so that the registers satisfy $A_j^{(31)} = 0$ and $D_j^{(31)} = \boxminus\phi^{(32)}(A_j^{(31)}, B_j^{(31)}, C_j^{(31)})$.

Algorithm 6.1 Pseudo-code description of SIMD.

```
function MessageExpansion($M$, $f$)                ▷ $f$ marks the final compression function
    if $f = 0$ then
        $(y_i) \leftarrow \mathsf{NTT}_{128}(M + X^{127})$              ▷ resp. $X^{255}$ for SIMD512
    else
        $(y_i) \leftarrow \mathsf{NTT}_{128}(M + X^{127} + X^{125})$        ▷ resp. $X^{255} + X^{253}$ for SIMD512
    end if
    Compute the $Z_j^{(i)}$'s by applying the inner codes $I_{185}$ and $I_{233}$ to the $y_i$'s.
    Compute the $W_j^{(i)}$'s by permuting the $Z_i^{(j)}$'s.
    return the $W_j^{(i)}$'s.
end function

function Round($\mathcal{S}$, $i$, $\pi_{[0..3]}$)
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+0)}, \mathsf{IF}, \pi_0, \pi_1)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+1)}, \mathsf{IF}, \pi_1, \pi_2)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+2)}, \mathsf{IF}, \pi_2, \pi_3)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+3)}, \mathsf{IF}, \pi_3, \pi_0)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+4)}, \mathsf{MAJ}, \pi_0, \pi_1)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+5)}, \mathsf{MAJ}, \pi_1, \pi_2)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+6)}, \mathsf{MAJ}, \pi_2, \pi_3)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, W_{[0..3]}^{(8i+7)}, \mathsf{MAJ}, \pi_3, \pi_0)$
    return $\mathcal{S}$
end function

function SIMD-Compress($IV$, $M$, $f$)
    $W \leftarrow$ MessageExpansion$(M, f)$
    $\mathcal{S} \leftarrow IV \oplus M$
    $\mathcal{S} \leftarrow$ Round$(\mathcal{S}, 0, [\ 3, 23, 17, 27])$
    $\mathcal{S} \leftarrow$ Round$(\mathcal{S}, 1, [28, 19, 22,\ 7])$
    $\mathcal{S} \leftarrow$ Round$(\mathcal{S}, 2, [29,\ 9, 15,\ 5])$
    $\mathcal{S} \leftarrow$ Round$(\mathcal{S}, 3, [\ 4, 13, 10, 25])$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, IV_{[0..3]}^{(0)}, \mathsf{IF},\ 4, 13)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, IV_{[0..3]}^{(1)}, \mathsf{IF}, 13, 10)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, IV_{[0..3]}^{(2)}, \mathsf{IF}, 10, 25)$
    $\mathcal{S} \leftarrow \mathsf{Step}(\mathcal{S}, IV_{[0..3]}^{(3)}, \mathsf{IF}, 25,\ 4)$
    return $\mathcal{S}$
end function

function SIMD($M$)
    Split the message $M$ into chunks $M_i, 0 \leq i < k$.
    $M_{k-1}$ is padded with zeros.
    $\mathcal{S} \leftarrow IV$
    for $0 \leq i < k$ do
        $\mathcal{S} \leftarrow$ SIMD-Compress$(\mathcal{S}, M_i, 0)$
    end for
    $\mathcal{S} \leftarrow$ SIMD-Compress$(\mathcal{S}, |M|, 1)$
    return Truncate$(\mathcal{S})$
end function
```

– The computation at step 32 is the first step of the feed-forward permutation P, and the computation output one IV word, up to a rotation:

$$\begin{aligned} A_j^{(32)} &= \left(D_j^{(31)} \boxplus IV_j \boxplus \phi^{(32)}(A_j^{(31)}, B_j^{(31)}, C_j^{(31)})\right)^{\lll s^{(32)}} \boxplus A_{p^{(32)}(j)}^{(31)}{}^{\lll r^{(32)}} \\ &= IV_j^{\lll s^{(32)}}. \end{aligned}$$

– Then, we have in the output of the compression function:

$$D_j^{(35)} = A_j^{(32)\lll r^{(33)}} = IV_j^{\lll s^{(32)}+r^{(33)}}$$

This gives some kind of partial fixed-points, where 128 bits of the output (256 for SIMD-512) are bits of the IV up to some rotation. This property is weaker than the fixed-points of a Davies-Meyer compression function, but it shows that the compression function of SIMD is not a perfect function.

Note that this has no implication on the security of the SIMD hash function: many designs are based on the plain Davies-Meyer which has a stronger distinguisher, but the only known attacks based on fixed-points use this property to build expandable messages. In the case of SIMD, expandable messages cannot be build from fixed points because the chaining value is twice as large as the output size. In fact, the security proofs of Coron *et al.* [CDMP05] show that the iteration of a Davies-Meyer based compression function is indifferentiable from a random oracle under the assumption that the block cipher is an ideal cipher.

The Message Expansion. When using a block cipher in Davies-Meyer mode to build hash function, the key of the block cipher is under control of the adversary. This setting is quite different from the regular use of a block cipher, where the key is random and unknown to the adversary. An attack against the hash function usually translates to a related-key or known key attack on the block cipher. Such attacks on block ciphers are not very well understood, and security proofs of block ciphers do not take them into account. For instance the resistance of the AES to differential cryptanalysis has been extensively studied [DR98, PSLL03], and it has been shown that the maximum differential probability for 4 AES rounds is bounded by $1.14 \cdot 2^{-111}$. However, this does not take into account related key attacks, and recent results show that the AES can be broken in related key setting [BKN09].

In SIMD, we tried to build a block cipher that would be resistant to related key attacks. By using a strong key expansion, we can prove that our construction is resistant to related key differential attacks. Moreover, because the design is wide-pipe, known-key attacks that do not put a difference in the message are not a threat: even a differential path with probability 1 is mostly harmless. The security proofs will be detailed in Chapter 7.

Indeed, most attacks against Davies-Meyer based hash functions take advantage of the weak message expansion. For the members of the MD/SHA family, the message expansion can be seen as a linear code and the minimal distance of this code seems to play a very important role. This minimal distance is only 3 and 4 for MD4 and MD5, so the adversary has a lot of control. In SHA-1, the minimal distance is no more than 44, and is exactly 25 in the last 60 words [JP06].

In our design, we follow the approach of Jutla and Patthak [JP06], who designed a better message expansion for SHA-1 with a minimal distance of 82, and 75 on the last 60 words. In [JP06], the authors used a code with a structure similar to the code of SHA-1 for efficiency reasons, and ruled out various algebraic codes. They consider Reed-Solomon codes over $\mathbb{F}_{2^8}$ which have a very good minimal distance, but they conclude they are unsuitable for a software implementation. In SIMD, we do use a Reed-Solomon code, but we use the field $\mathbb{F}_{257}$ to allow efficient software implementations. The designers of SWIFFT [LMPR08] already used this field for the same reason. Finally using concatenated codes, we can increase the minimal distance without adding much computations.

Our message expansion is designed to avoid related key attacks on the block cipher. It has a minimal distance of at least 520 for SIMD-256, and at least 1032 for SIMD-512. After the NTT layer of SIMD-256, any pair of distinct message are mapped to a sequence of 128 elements in $\mathbb{F}_{257}$, with at least 65 non-equal components (resp. 256 elements with 129 non-equal elements). The concatenated code maps the elements of $\mathbb{F}_{257}$ to 16-bit words, so that distinct elements are mapped to words with a Hamming distance of at least 4. We have two copies of the concatenated code (with two different inner codes), so this makes a minimal Hamming distance of $2 \times 4 \times 65 = 520$ for the message expansion of SIMD-256 (resp. $2 \times 4 \times 129 = 1032$ for SIMD-512).

6.3 Implementation Aspect and Performances

The design of SIMD is highly parallellisable due to the choice of the components: the NTT and the parallel Feistel ladders. This should allow efficient hardware implementations. As far as software is concerned, we can use SIMD instructions (Single Instructions, Multiple Data) to compute most operations in parallel.

6.3.1 Software Implementation

SIMD Instructions. SIMD instructions allow to compute a given operation on multiple data in parallel. Processors that support SIMD instructions usually come with a set of dedicated registers, which can contain a vector of integers or floating point data. For instance the SSE registers in x86 processors are 128-bit wide and can be used to store 16 8-bit values, 8 16-bit values, 4 32-bit values, or 2 64-bit values. The SIMD instruction set allows to compute in parallel some arithmetic operations on those vectors: addition, multiplication, bitwise operations, rotations, ...

SIMD instructions were introduced in personal computers to improve the efficiency of multimedia computations, and are now very widely available. The x86 family supports MMX since 1997 and SSE since 1999 and the PPC family has AltiVec since 1998. For embedded systems, Intel has introduced IwMMXt to its PXA family of ARM processors, and is now promoting the Atom, an x86 processor which supports SSE. Moreover, recent ARM processors such as the Cortex family support a set of vector instructions called NEON. We believe that SIMD support will become even more widespread in the future. We also note that the efficiency of SIMD implementations is constantly improving: the SSE units of Intel Core micro-architecture based processors is much faster than in the older NetBurst micro-architecture. Similarly, the new AMD K10 processors feature a much better SSE units than AMD K8 ones.

Moreover, Intel has announced that it will introduce 256-bit wide SIMD registers with the new AVX instructions in its next CPU generation. AMD has also announced future support for AVX and they will also add new operations with the XOP extension. Two specific instructions in XOP are of particular interest to hash functions: parallel rotations, and the bitwise conditional move which implements the Boolean function IF. This is expected to significantly improve the speed of SIMD on this platform.

Another advantage of SIMD instructions is that they usually come with a relatively large set of registers, even on CISC processors. The x86 architecture has only 8 general purpose 32-bit registers but SSE instructions comes with 8 extra 128-bit registers (on x86-64 we have 16 general purpose 64-bit registers, and 16 128-bit SSE registers). In most cases, the full state of the Feistel ladder can be kept inside those registers, which is good for performances. The NTT can also be computed mostly inside the registers.

Optimized Implementation. The NTT is the main component of the message expansion, and the high minimal distance is mainly due to the NTT. However the NTT is quite expensive and has to be carefully optimized to make an efficient implementation of SIMD.

Field Operations. An element of $\mathbb{F}_{257}$ will be stored in a signed 16-bit word. The choice of the field $\mathbb{F}_{257}$ allows an efficient implementation of the field operations, because 257 is a prime and $256 = -1 \pmod{257}$.

To reduce x modulo 257, we let q and r be the quotient and the remainder of the division of x by 256, and we have $x = r - q \pmod{257}$. This can be written as `(x&0xff) - (x»8)` in C, provided that the integers are represented in a two's-complement arithmetic, and that the right shift operation preserves the sign bit (which is an implementation-defined behaviour). The result is not fully reduced and will be between -127 and 383. When a full reduction is needed, we substract 257 to values over 128, and get a result between -128 and 128. This reduction is quite fast to compute because it only uses bitwise operations, and no division. Moreover, these operations are available in most SIMD instruction sets.

To compute a multiplication in $\mathbb{F}_{257}$, we reduce both operands to $\{-128, ...128\}$, and the result can be computed with a single multiplication modulo 2^{16} with no overflow.

Note that it is not necessary to perform a reduction after each field operation, because we have some extra bits in a 16-bit word. We have to study the NTT algorithm to find out where reductions are needed.

NTT Computation. Because SIMD instructions compute the same operation on each element of the vectors, we will not use the classical radix-2 FFT algorithm. Instead, we rewrite the one-dimensional NTT as a two-dimensional one. In our implementation, we rewrite the NTT of size 64 as a two-dimensional NTT of size 8×8. The input data is seen as a 8×8 matrix, and the computation of the NTT_{64} is done in three steps:

- First we compute 8 NTT_8 on the columns of the matrix using a decimation in time algorithm.
- We multiply by the twiddle factors, transpose the matrix, and permute the row and the columns following the bit reversal order.
- Then we compute 8 NTT_8 on the columns of the matrix using a decimation in frequency algorithm.

The first and the last step are easy to parallelize with SIMD instructions. Moreover, the root of unity used in the NTT_8 is 4, so the multiplications needed for the NTT_8 are only bit shifts. The transposition can be implemented using the merge operation that is available on most SIMD instruction sets (`punpcklwd`/`punpckhwd` in SSE, `vmrghh`/`vmrglh` in Altivec, `wunpckilh`/`wunpckihh` in IwMMXt).

For more details on the NTT and FFT algorithms, see [Nus82].

Computation of the Feistel Ladder. The computation of the Feistel Ladder can also be parallelized with SIMD instruction. We store each row of the state matrix in a vector register, and the step functions can be computed in parallel.

Multi-core. Our design can also exploit multi-core processors: the most expensive part of the algorithm is the message expansion, and it can be done in parallel for different message blocks. When using two cores, we gain a factor 1.8 on the performances.

The synchronisation cost can be made quite low if each core computes the message expansion on many message block before waiting for the chaining value from the other core.

Table 6.2 Performances of SIMD compared to the SHA family, in megabyte per second (MB/s).

Architecture		SHA-1/256/512			Scalar SIMD-256/512		Vector SIMD-256/512	
Core2	32 bits	261	140	45	31	24	245	220
	64 bits	323	176	223	45	33	270	250
K10	32 bits	207	135	39	30	20	145	145
	64 bits	301	147	193	38	29	160	160
P4	32 bits	147	89	19	16	13	85	65
K8	32 bits	174	107	31	23	15	80	?
	64 bits	238	111	148	30	22	78	?
Atom	32 bits	66	35	12	7.2	5.7	64	50
G4	32 bits	102	55	16	10	7.5	78	55
ARM		19	11	3.0	2.1	1.6	9	?

Table 6.3 Performances of SIMD compared to the SHA family, in cycles per byte (c/B).

Architecture		SHA-1/256/512			Scalar SIMD-256/512		Vector SIMD-256/512	
Core2	32 bits	11	21	63	90	118	12	13
	64 bits	9	16	13	63	85	11	12
K10	32 bits	12	18	64	80	125	17	17
	64 bits	9	17	13	65	85	16	16
P4	32 bits	19	89	147	170	210	32	43
K8	32 bits	12	19	65	90	135	25	?
	64 bits	9	18	14	66	88	26	?
Atom	32 bits	24	46	133	220	280	25	32
G4	32 bits	12	23	78	125	166	16	23
ARM		22	38	138	200	260	46	?

Performance. SIMD-512 and SIMD-256 offer comparable performances: one SIMD-512 compression function needs roughly twice the number of operations of one SIMD-256 compression function, but the message block size is also doubled. SIMD-512 is still somewhat slower because of the higher memory requirement, and the slightly more expensive NTT (because of the $\log n$ factor). As a general rule, the message expansion of SIMD takes half of the computing time.

The memory requirement of SIMD is essentially the internal state (64 bytes for SIMD-256 and 128 bytes for SIMD-512) and the output of the NTT ($4 \times 64 = 256$ bytes for SIMD-256 and $4 \times 128 = 256$ bytes for SIMD-512).

Performances on a range of computers are given in Table 6.2 and Table 6.3. We compare two implementations of SIMD, a scalar one written in pure C and a vectorized one written in C using compiler extensions to access the SIMD instructions. Our vector implementation runs on x86 with SSE2, on PowerPC with Altivec, and on ARM with IwMMXt. The performance for SHA-1, SHA-256 and SHA-512 have been obtained using the implementation from sphlib [Sphl]. We used the same compiler for SHA and SIMD. More extensive benchmarks are available from the eBASH project [eBACS].

Software Platforms. Here is a brief description of the test platforms:

Core2:	Intel Xeon E5440 running at 2.83 GHz; compiled with gcc 4.1.2.
K10:	AMD Phenom 9850 running at 2.5 GHz; compiled with gcc 4.2.4.
P4:	Intel Pentium 4 running at 2.8 GHz; compiled with gcc 4.1.2.
K8:	AMD Athlon64 X2 3800+ running at 2 GHz; compiled with gcc 4.2.3.
Atom:	Intel Atom N270 running at 1.6 GHz; compiled with gcc 4.1.3.
G4:	PowerPC 7447 running at 1.25 GHz; compiled with gcc 4.1.2.
ARM:	Intel XScale PXA270 running at 416 MHz; compiled with gcc 4.1.3.

6.3.2 8-bit Implementation

We also tested SIMD on a 8-bit platform. We used gcc to compile the optimized code to an Atmel AVR AtMega8, and we ran it in the simularv simulator. We optimized some part of the code with inline assembly to handle efficiently some 32-bit operations on the 8-bit architecture. Our code ran at approximately 1300 cycles/byte.

6.3.3 Hardware Implementation

We did a preliminary study to implement SIMD on a FPGA. The Feistel part of SIMD can be implemented in the same way as the Feistel part of other hash functions of the MD/SHA family, and we would include the hardware to compute the four Feistels in parallel (resp. eight for SIMD-512). Since SIMD has fewer steps than SHA-1 and SHA-2, this part will run faster, but requires more gates to compute the four Feistels. To compute the NTT, we propose to include the hardware to compute a size 8 NTT, which will be called 32 times to compute the size 128 NTT of SIMD-256. It should run at about the same speed as the Feistel part.

6.4 Expected Strength

We conjecture that no non-random properties of an instance of SIMD-224 or SIMD-256 (indexed by the IV) can be identified with less than 2^{256} calls to the compression function.

Similarly we conjecture that no non-random properties of an instance of SIMD-384 or SIMD-512 can be identified with less than 2^{512} calls to the compression function.

In particular this means that we believe that a collision attack on SIMD-n has a complexity of $2^{n/2}$, and a preimage or second preimage attack has a complexity of 2^n. There should be neither shortcut multi-collision attack nor shortcut second preimage against long messages.

6.4.1 Security of the Compression Function

The compression function of SIMD has a relatively slow diffusion, and it might be possible to build free-start distinguishers that do not introduce differences in the message. This was the case in SIMD 1.0, as shown by [MN09]. However, this kind of distinguisher is very unlikely to be useful against the full hash function because of the wide-pipe design of SIMD. The compression function was designed assuming that the chaining value is hard to control, which is the case when the compression function is iterated to build the hash function.

6.5 Security Analysis

In this Section we briefly discuss the security of SIMD. More specific results will be given in Chapter 7.

6.5.1 Mode of Operation

Mode of Operation for the Hash Function. Since we use a modified message expansion for the final compression function, this is equivalent to having an extra input bit to the message expansion, and using a prefix free encoding. This allows better security proofs of the iteration mode. Alternatively, we can model the compression function C and the final compression C' as two independent random oracles and see our construction as an instance of the wide-pipe design of Lucks [Luc05].

Thus, following proofs from [CN08, MT07, Luc05], our iteration mode is indifferentiable from a random oracle if the compression function is a random oracle. These proofs show that there is no generic attack against the mode of operation. Moreover, the security proved is up to 2^n queries, where n is the output length of the hash function. Consequently, there are no generic attacks for collision, second-preimage or preimage.

Furthermore, Coron *et al.* [CDMP05] showed that the same results can be obtained under the assumption that the underlying block cipher is an ideal cipher. This result shows that the distinguisher based on fixed-point that can be found against the Davies-Meyer mode and our modified Davies-Meyer mode does not affect the security of the hash function.

Moreover, since there is a prefix-free encoding of the message, one can prove that cascading the compression function is a PRF under the assumption that the compression function is a PRF [BCK96b]. This can be used to prove the security of MAC constructions under weaker assumptions that the indifferentiability of the compression function..

Mode of Operation for the Compression Function. The mode of operation for the compression function does not follow directly from the Davies-Meyer mode of operation. This mode presents some weaknesses we want to avoid : fixed points can be easily found for example. The mode we used can be seen as a variant of the construction 8 of paper [BRS02] (and construction 41 from [PGV93]). Finally, the proofs provided in [BRS02] can be extended to our construction in the ideal cipher model.

6.5.2 Security of the Compression Function

Resistance to Differential Cryptanalysis. The SIMD family is provably secure against a class of differential attacks. This is based on the fact that the message expansion has a high minimal distance: any pair of distinct messages gives expanded messages with at least 520 bit differences for SIMD-256, resp. 1032 for SIMD-512.

If we assume that the adversary does not control the positions of the differences in the expanded message, each difference in the expanded message will introduce a difference in the state, and the adversary has to control its propagation. The adversary must at least control the effect of the carry, which can only be good with a probability of 2^{-1}. As a comparison, in SHA-1, it is quite easy to control the error propagation because the perturbation vector can be shifted to correct the errors, but the success probability is only $2^{-2.5}$. We expect that it will actually be more difficult to control the propagation of differences in SIMD.

Even if the adversary can use message-modification techniques to control the non-linearity in one half of the hash function for free, he still has to deal with 260 differences, resp. 516 for SIMD-512. Note that our compression function construction forces the adversary to choose the message from the beginning, so we do not expect message modification techniques to work.

The Step Update Function. The step update function of SIMD is defined as:

$$A_j^{(i)} = \left(A_j^{(i-4)^{\lll r_{i-4}}} \boxplus W_j^{(i)} \boxplus \phi^{(i)}\left(A_j^{(i-1)}, A_j^{(i-2)^{\lll r_{i-2}}}, A_j^{(i-3)^{\lll r_{i-3}}}\right)\right)^{\lll s^{(i)}} \boxplus A_{p^{(i)}(j)}^{(i-1)^{\lll r^{(i)}}}$$

It is quite similar to the step update functions of members of the MD/SHA family, and has been built with previous attacks on these functions in mind.

Our function is of form $A^{(i)} = A^{(i-4)} \boxplus \phi(A^{(i-3)}, A^{(i-2)}, A^{(i-1)}) \boxplus A^{(i-1)}$, like in MD5. This gives a good avalanche effect, since a difference in $A^{(i-1)}$ will most likely be propagated to $A^{(i)}$ and cannot be easily absorbed. Most attacks on MD4 are based on the fact that the step update allows to easily absorb a difference in the internal state.

Den Boer and Bosselaers discovered another kind of weakness in the step update function of MD5 [dBB93]. If there is some differential pattern in $A^{(i-4)}, A^{(i-3)}, A^{(i-2)}, A^{(i-1)}$, that can be cancelled through F, then the addition of $A^{(i-1)}$ will reintroduce this pattern and it will propagate in the compression function. To avoid this kind of attack, we added a rotation on $A^{(i-1)}$ in the design of SIMD.

6.5.3 Use as a MAC

We propose two distinct ways to build a Message Authentication Code from the SIMD hash function.

First, as any Merkle-Damgård based hash function, SIMD can be used with the HMAC construction. We define HMAC-SIMD as:

$$\text{HMAC-}\mathsf{SIMD}_k(M) = \mathsf{SIMD}(\bar{k} \oplus \text{opad} \,||\mathsf{SIMD}(\bar{k} \oplus \text{ipad} \,||M))$$

where $\bar{k}$ is k padded to a full message block (512 bits for SIMD-256, and 1024 bits for SIMD-512), and ipad and opad are defined as in [RFC2104]. Note that this construction uses the truncated output of the inner hash function as input for the outer hash function, in order to use the hash function as-is. The security proof of Bellare in [Bel06] can be used to prove the security of HMAC-SIMD.

Alternatively, we can simply compute $\text{MAC}_k(M) = \mathsf{SIMD}(k\|M)$ where $\|$ denotes the concatenation. The key can optionally be padded to a full block to allow more efficient implementations. Thanks to the security proof in the indifferentiability framework, there are no generic shortcut attack on this construction. This means that one has to find a weakness in the compression function in order to break this MAC.

6.A Test Vectors

We give the output of SIMD on some particular messages as test vectors. We use three messages for each function: the empty message, a message that fills exactly one block with a 8-bit counter, and a message made of a sequence of 1 bits, that fill one block and some part of a second block.

M	SIMD224(M)
Empty message	`43e1d53656d7b85d10d5499e28afdef90bb497730d2853c8609b534b`
0x00, 0x01, 0x02, ...0x3f	`cbb4f8a9304b4b043093a94b7059ee36e43ff94a21dc46611f1a7769`
700 1 bits	`19499a44b7541c7f27b867b207f87269d7351d2cbb405f7c0cb491aa`

M	SIMD256(M)
Empty message	`8029e81e7320e13ed9001dc3d8021fec695b7a25cd43ad805260181c35fcaea8`
0x00, 0x01, 0x02, ...0x3f	`5bebdb816cd3e6c8c2b5a42867a6f41570c4b917f1d3b15aabc17f24679e6acd`
700 1 bits	`e80b4eeb8a370e6ca918e7810400441f6dd0da1eb4559cade791c314f82d524a`

M	SIMD384(M)
Empty message	`5fdd62778fc213221890ad3bac742a4af107ce2692d6112e\`
	`795b54b25dcd5e0f4bf3ef1b770ab34b38f074a5e0ecfcb5`
0x00, 0x01, 0x02, ...0x7f	`5e02e645868ef837f535f44609a268a0a146476584d50f83\`
	`683ce3e7cb355caaf7e8eb81cb28db3ccf40d25313f16950`
1079 1 bits	`e999b35b42301eca3a9c648fef39635b13059b2ac3be16f5\`
	`c9372d3e773a716f1b2a23b784f3c1e231e42d87d2c950f3`

M	SIMD512(M)
Empty message	`51a5af7e243cd9a5989f7792c880c4c3168c3d60c4518725fe5757d1f7a69c63\`
	`66977eaba7905ce2da5d7cfd07773725f0935b55f3efb954996689a49b6d29e0`
0x00, 0x01, 0x02, ...0x7f	`8851ad0a57426b4af57af3294706c0448fa6accf24683fc239871be58ca913fb\`
	`ee53e35c1dedd88016ebd131f2eb0761e97a3048de6e696787fd5f54981d6f2c`
1079 1 bits	`c060fa9aae2414715a3a27c5df22dbd41469e3d09056b093861f0f6b9bdc311a\`
	`24ac743811e88358dc69094ad444036bbbf7708ed8bdeaf1e8ed871dfb79c218`

Chapter 7

Security Analysis of SIMD

7.1 Introduction

In this chapter we provide three important contributions to the security analysis of SIMD. In Section 7.2 we study its resistance against self-similarity attacks [BDFL10, 8]. This class of attack is inspired by the complementation property of DES and includes symmetry-based attacks (more self-similarity attacks will be shown in Chapter 8). In the case of SIMD, we show that it is possible to exploit the symmetry of the design using special messages. This shows that the constants included in the message expansion of SIMD are not sufficient to prevent symmetry relations, and non-symmetric constants should be added in the last steps of the message expansion. The study of this symmetry property shows that it is much weaker that symmetry properties in CubeHash [ABM$^+$09, FLM10] or *Lesamnta* [BDFL10, 8.3]. More precisely, most symmetry properties can be used to generate many symmetric states out of a single state, but this is not the case for SIMD.

In Section 7.3, we show a proof of security for the mode of operation used in SIMD, the truncated prefix-free Merkle-Damgård, in the presence of some efficient distinguishers on the compression function. The class of distinguisher we consider includes the symmetry based distinguisher, and also includes differential paths with a non-zero chaining value difference. This shows that the properties of the compression function of SIMD found so far do not affect the security of the iterated hash function. This part is also of independent interest and applies to other wide-pipe hash functions.

In Section 7.4, we study differential attacks, and bound the probability of paths with a non-zero message difference, *i.e.* related key attacks on the block cipher. We show an upper bound on such paths on the order of $2^{n/2}$, and we argue that the best paths are probably much worse than this bound. We note that there are very few results known regarding resistance to related key attack for block ciphers. In particular, the differential properties of the AES have been extensively studied [PSLL03] but related key differential attacks have been shown recently [BK09]. In many hash function designs (in particular those based on the Davies-Meyer construction), related key attacks are a real concern and should be studied accordingly.

By combining the results of Section 7.3 and 7.4, we show that SIMD is resistant to differential cryptanalysis: a path with a non-zero difference in the chaining value input cannot be used to attack the hash function because it is wide-pipe, while a path a non-zero difference in the message can only have a low success probability.

Finally, in Section 7.5 we express our views on the security of SIMD.

7.1.1 Known Cryptanalysis Results

As far as we know, the following results have been found on SIMD:

– In [GB09], Gauravaram and Bagheri showed that the modified Davies-Meyer construction used in SIMD allows to find partial fixed-points (this is a weaker version of Davies-Meyer's fixed-points). There is no easy way to find full fixed-points as in the original Davies-Meyer construction, but those partial fixed-points give an easy distinguisher of the compression function. Just like the fixed-points of Davies-Meyer, this property does not affect the security of the hash function, and the mode can be proven secure under the assumption that the block cipher is ideal [CDMP05].

– In [MN09], Mendel and Nad showed a differential path with probability 2^{-507} for the compression function of the round-1 version of SIMD-512. They used it to make a distinguishing attack on the compression function with complexity 2^{427}, using IV/message modifications. In this path, no difference is introduced in the message, but a specific difference Δ_{in} in the chaining value can go to a difference Δ_{out}. Because of the need to control the chaining value difference, this path cannot be used to attack the iterated hash function. In Section 7.3, we show that even if there is a path with probability one, we only loose a factor 2 in the indifferentiability proof.
However, this path was using some unwanted properties of the permutations used in the compression function, and it was decided to remove those properties by tweaking the design for the second round of the SHA-3 competition [LBF09].

– In [NPSS10], Nikolić *et al.* applied rotational cryptanalysis to the compression function of SIMD-512. They showed that 24 rounds can be distinguished from a random function with complexity 2^{497} if the constants are removed from the design. In the real design, they can only distinguish 12 rounds (out of 36) because of the non-linear message expansion. This is clearly not a threat for SIMD-512.

7.2 A Distinguisher for the Compression Function of SIMD

Our distinguisher is based on symmetries in the design, and follows the ideas of [BDFL10]. Symmetry based properties have already been found in several hash function designs, such as CubeHash [ABM+09, FLM10] or *Lesamnta* [BDFL10, 8.3]. We describe the distinguisher in the case of SIMD-256, but it applies similarly to SIMD-512.

7.2.1 Building the Symmetric Messages

The basic idea is to build a message so that the expanded message is symmetric. Then, if the internal state is also symmetric, the compression rounds preserve the symmetry. This can also be used with a pair of symmetric messages, and a pair of symmetric states.

The NTT layer of the message expansion is an affine transformation, therefore it is easy to find inputs that satisfy some affine conditions on the output. Since it only doubles the size of the input, we have enough degrees of freedom to force equalities between pairs of output. The next expansion step is a multiplication by a constant, and it will preserve equality relations.

Then if we look at the permutations used in the message expansion, they have the following property [1]: the NTT words used to build the message words $W_0^{(i)}, W_1^{(i)}, W_2^{(i)}, W_3^{(i)}$ are always of the form $(y_{k_1}, y_{k_2}), (y_{k_1+2}, y_{k_2+2}), (y_{k_1+4}, y_{k_2+4}), (y_{k_1+6}, y_{k_2+6})$ for some k_1 and k_2 (with $k_i = 0 \bmod 8$ or $k_i = 1 \bmod 8$). The full permutations are given in Table 6.1, page 126. Because of this property, if we have $y_i = y_{i\oplus 2}$ after the NTT, then we have $W_0^{(i)} = W_1^{(i)}$ and $W_2^{(i)} = W_3^{(i)}$.

1. This design choice was guided by implementation efficiency

This allows to build a symmetric message. An example of such a symmetric message is given in Table 7.1.

More precisely, let us use the notation $\overset{\leftrightarrow}{\bullet}$ to denote this symmetry relation, and $\underset{\leftrightarrow}{\bullet}$ and $\overset{\leftrightarrow}{\underset{\leftrightarrow}{\bullet}}$ to denote the other two possible symmetries:

$$\begin{aligned}
\overleftrightarrow{(X_0, X_1, X_2, X_3)} &= (X_1, X_0, X_3, X_2) \\
\underleftrightarrow{(X_0, X_1, X_2, X_3)} &= (X_2, X_3, X_0, X_1) \\
\underleftrightarrow{\overleftrightarrow{(X_0, X_1, X_2, X_3)}} &= (X_3, X_2, X_1, X_0)
\end{aligned}$$

We now consider two messages M and M'. We use y to denote the NTT output for M, and y' to denote the NTT output for M'. The equality constraints on the NTT output that are necessary to build a pair of symmetric expanded messages are (we use $\mathcal{E}$ to denote the message expansion):

$$\begin{aligned}
y_i = y'_{i\oplus 2} &\Rightarrow \mathcal{E}(M) = \overleftrightarrow{\mathcal{E}(M')} \\
y_i = y'_{i\oplus 4} &\Rightarrow \mathcal{E}(M) = \underleftrightarrow{\mathcal{E}(M')} \\
y_i = y'_{i\oplus 6} &\Rightarrow \mathcal{E}(M) = \underleftrightarrow{\overleftrightarrow{\mathcal{E}(M')}}
\end{aligned}$$

In Appendix 7.A we solve the linear systems involved, and we describe the sets of symmetric messages. For SIMD-256 we have the following results:

Symmetry class			# msg	# pairs
$\overset{\leftrightarrow}{\bullet}$	$y_i = y'_{i\oplus 2}$	$W_i = W'_{i\oplus 1}$	2^8	$256 \cdot 255$
$\underset{\leftrightarrow}{\bullet}$	$y_i = y'_{i\oplus 4}$	$W_i = W'_{i\oplus 2}$	2^{16}	$(256 \cdot 255)^2$
$\overset{\leftrightarrow}{\underset{\leftrightarrow}{\bullet}}$	$y_i = y'_{i\oplus 6}$	$W_i = W'_{i\oplus 3}$	2^8	$256 \cdot 255$

there are about 2^{16} symmetric messages, and less than 2^{32} symmetric pairs.

For SIMD-512 the results are:

Symmetry class		# msg	# pairs
$y_i = y'_{i\oplus 2}$	$W_i = W'_{i\oplus 1}$	2^8	$256 \cdot 255$
$y_i = y'_{i\oplus 4}$	$W_i = W'_{i\oplus 2}$	2^{16}	$(256 \cdot 255)^2$
$y_i = y'_{i\oplus 6}$	$W_i = W'_{i\oplus 3}$	2^8	$256 \cdot 255$
$y_i = y'_{i\oplus 8}$	$W_i = W'_{i\oplus 4}$	2^{32}	$(256 \cdot 255)^4$
$y_i = y'_{i\oplus 10}$	$W_i = W'_{i\oplus 5}$	2^8	$256 \cdot 255$
$y_i = y'_{i\oplus 12}$	$W_i = W'_{i\oplus 6}$	2^{16}	$(256 \cdot 255)^2$
$y_i = y'_{i\oplus 14}$	$W_i = W'_{i\oplus 7}$	2^8	$256 \cdot 255$

there are about 2^{32} symmetric messages, and less than 2^{64} symmetric pairs. An important property of these message classes is that they are all disjoints: it is not possible to use the intersection of two symmetry classes.

7.2.2 Symmetry Property on the Compression Function

Let us consider a pair of symmetric messages for one of the symmetry relations (without loss of generality, we assume it's the $\overset{\leftrightarrow}{\bullet}$ symmetry): $\mathcal{E}(M') = \overleftrightarrow{\mathcal{E}(M)}$. We can take advantage of the symmetry of the Feistel part using those messages. If we have a pair of states $\mathcal{S}^{(i)}, \mathcal{S}'^{(i)}$ with $\mathcal{S}'^{(i)} = \overleftrightarrow{\mathcal{S}^{(i)}}$ and we compute one Feistel step with messages W and W' such that $W' = \overleftrightarrow{W}$, we obtain a new pair of states with $\mathcal{S}'^{(i+1)} = \overleftrightarrow{\mathcal{S}^{(i+1)}}$. The xor-based symmetry classes commute with the xor-based permutations $p^{(i)}$ used to mix the Feistels (and they are the only symmetry classes to do so).

Because the compression function is built using a modified Davies-Meyer mode (Figure 6.2), we need to start with H_{i-1} such that $H_{i-1} \oplus M$ is symmetric: $H'_{i-1} \oplus M' = \overleftrightarrow{H_{i-1} \oplus M}$. Then, in the feed-forward, H_{i-1} is used as the key to a few Feistel rounds, and since H_{i-1} is not symmetric, those rounds will break the symmetry. However, it turns out the symmetric messages are very sparse, so H_i will be almost symmetric, and the feed-forward will mostly preserve the symmetry of the outputs.

This gives a distinguisher on the compression function: an almost symmetric chaining value is transformed into a somewhat symmetric chaining value. See Table 7.1 for a concrete example.

The distinguisher can be used either with a pair of messages and chaining values with $\mathcal{E}(M') = \overleftrightarrow{\mathcal{E}(M)}$ and $H'_{i-1} \oplus M' = \overleftrightarrow{H_{i-1} \oplus M}$, or with a single chaining value and message, with $\mathcal{E}(M) = \overleftrightarrow{\mathcal{E}(M)}$ and $H_{i-1} \oplus M = \overleftrightarrow{H_{i-1} \oplus M}$.

7.2.3 Non-Ideality of the Compression Function

Here we define the bias of the compression function with the notations that will be used in Section 7.3.

For each symmetric message M under a symmetry relation (denoted by $\overleftrightarrow{\bullet}$ without loss of generality), we have a first order relation between the inputs and output of the compression function:

$$\mathcal{R}_1^M(h, m, h') := \left(m = M \wedge h \oplus m = \overleftrightarrow{h \oplus m}\right) \Rightarrow P^{-1}(h', h) = \overleftrightarrow{P^{-1}(h', h)}$$

We use the feed-forward permutation P to define the relation, because it is tricky to describe exactly the somewhat symmetry of h' after the feed-forward. We have about 2^{16} such relations for SIMD-256 and about 2^{32} relations for SIMD-512. We can capture all of them in a single relation:

$$\mathcal{R}_1(h, m, h') := \bigwedge_M \mathcal{R}_1^M(h, m, h').$$

Similarly, for each symmetric message pair M, M', this gives a second order relation (there are about 2^{32} such relations for SIMD-256 and 2^{64} for SIMD-512):

$$\mathcal{R}_2^{M,M'}(h_1, m_1, h_2, m_2, h'_1, h'_2) :=$$
$$\left(m_1 = M \wedge m_2 = M' \wedge h_1 \oplus m_1 = \overleftrightarrow{h_2 \oplus m_2}\right) \Rightarrow P^{-1}(h'_1, h_1) = \overleftrightarrow{P^{-1}(h'_2, h_2)}$$

$$\mathcal{R}_2(h_1, m_1, h_2, m_2, h'_1, h'_2) := \bigwedge_{M,M'} \mathcal{R}_2^{M,M'}(h_1, m_1, h_2, m_2, h'_1, h'_2)$$

The corresponding weak states are:

$$\mathcal{W}_1^M := \{M \oplus x \mid x = \overleftrightarrow{x}\} \qquad \mathcal{W}_1 := \bigcup_M \mathcal{W}_1^M$$

$$\mathcal{W}_2^{M,M'} := \left\{(h, \overleftrightarrow{h} \oplus M' \oplus \overleftrightarrow{M})\right\} \qquad \mathcal{W}_2 := \bigcup_{M,M'} \mathcal{W}_2^{M,M'}$$

Table 7.1 Example of the property. The output is almost symmetric.

Chaining Value = Message				
$A_{0..3}$	00000000	00000000	00000000	00000000
$B_{0..3}$	00000000	00000000	00000000	a2000000
$C_{0..3}$	00000000	00000000	00000000	00000000
$D_{0..3}$	00000000	00000000	00000000	f1000000

Expanded Message				
$W_{0..3}^{(0)}$	0a1ee3d1	0a1ee3d1	bc12531b	bc12531b
$W_{0..3}^{(1)}$	a5abca86	a5abca86	4be14335	4be14335
$W_{0..3}^{(2)}$	e827b082	e827b082	1b761da1	1b761da1
$W_{0..3}^{(3)}$	287848fd	287848fd	aa01d8fa	aa01d8fa
$W_{0..3}^{(4)}$	2fb2e543	2fb2e543	c914c4be	c914c4be
$W_{0..3}^{(5)}$	050f4ec5	050f4ec5	de09ccb1	de09ccb1
$W_{0..3}^{(6)}$	143cc7a2	143cc7a2	31ddec7d	31ddec7d
$W_{0..3}^{(7)}$	50f0d841	50f0d841	0dbbb1f4	0dbbb1f4
$W_{0..3}^{(8)}$	d04e1abd	d04e1abd	36ec3b42	36ec3b42
$W_{0..3}^{(9)}$	ebc4385e	ebc4385e	ce231383	ce231383
$W_{0..3}^{(10)}$	f5e21c2f	f5e21c2f	43eeace5	43eeace5
$W_{0..3}^{(11)}$	17d94f7e	17d94f7e	e48ae25f	e48ae25f
$W_{0..3}^{(12)}$	af1027bf	af1027bf	f2454e0c	f2454e0c
$W_{0..3}^{(13)}$	faf1b13b	faf1b13b	21f7334f	21f7334f
$W_{0..3}^{(14)}$	d788b703	d788b703	55ff2706	55ff2706
$W_{0..3}^{(15)}$	5a55357a	5a55357a	b41fbccb	b41fbccb
$W_{0..3}^{(16)}$	320fcdf1	320fcdf1	624c9db4	624c9db4
$W_{0..3}^{(17)}$	a4135bed	a4135bed	3126ceda	3126ceda
$W_{0..3}^{(18)}$	21adde53	21adde53	4aa2b55e	4aa2b55e
$W_{0..3}^{(19)}$	237fdc81	237fdc81	975568ab	975568ab
$W_{0..3}^{(20)}$	435abca6	435abca6	ab5b54a5	ab5b54a5
$W_{0..3}^{(21)}$	9ccb6335	9ccb6335	409fbf61	409fbf61
$W_{0..3}^{(22)}$	641e9be2	641e9be2	daaf2551	daaf2551
$W_{0..3}^{(23)}$	46feb902	46feb902	1893e76d	1893e76d
$W_{0..3}^{(24)}$	71c58e3b	71c58e3b	a06f5f91	a06f5f91
$W_{0..3}^{(25)}$	1e09e1f7	1e09e1f7	dd6a2296	dd6a2296
$W_{0..3}^{(26)}$	9a1065f0	9a1065f0	eeb5114b	eeb5114b
$W_{0..3}^{(27)}$	c3ee3c12	c3ee3c12	452cbad4	452cbad4
$W_{0..3}^{(28)}$	e684197c	e684197c	c1333ecd	c1333ecd
$W_{0..3}^{(29)}$	f9a1065f	f9a1065f	2ac7d539	2ac7d539
$W_{0..3}^{(30)}$	f3420cbe	f3420cbe	558eaa72	558eaa72
$W_{0..3}^{(31)}$	cd0832f8	cd0832f8	6c4f93b1	6c4f93b1

Output				
$A_{0..3}$	0e0618e6	0ee618e6	ec5a3cee	fbdc48ae
$B_{0..3}$	17bde794	17bddbd4	5a0a59f2	5a2a59f2
$C_{0..3}$	12a9c015	12a9c015	be7d3df1	be775df1
$D_{0..3}$	15f9cb8d	15f9cb8d	2efef45c	2efef45c

The study of the symmetry classes of SIMD, in Appendix 7.A shows that:

$$|\mathcal{W}_1| = 2^{256} \cdot 256^2 + 2 \cdot 256 \approx 2^{256} \cdot 2^{16} \qquad \text{for SIMD-256}$$

$$|\mathcal{W}_1| = 2^{512} \cdot 256^4 + 2 \cdot 256^2 + 4 \cdot 256 \approx 2^{512} \cdot 2^{32} \qquad \text{for SIMD-512}$$

$$|\mathcal{W}_2| = 2^{512} \cdot ((256 \cdot 255)^2 + 2 \cdot 256 \cdot 255) < 2^{512} \cdot 2^{32} \qquad \text{for SIMD-256}$$

$$|\mathcal{W}_2| = 2^{1024} \cdot ((256 \cdot 255)^4 + 2 \cdot (256 \cdot 255)^2 + 4 \cdot 256 \cdot 255) < 2^{1024} \cdot 2^{64} \qquad \text{for SIMD-512}$$

Each chaining value can be used with less than 2^{32} related chaining values (less than 2^{64} for SIMD-512) and each such pair can be used with a single message.

7.2.4 Impact of the Symmetry-based Distinguisher

There are two main classes of attacks based on symmetric properties of the compression function. To attack the compression function, one can use the symmetry property to force the output of the compression function in a small subspace. This allows to find collisions in the compression function more efficiently than brute force, with the efficiency of this attack depending on the size of the symmetry classes. On the other hand, to attack the hash function, one can first try to reach a symmetric state using random messages, and then use symmetric messages to build a large set of symmetric states. To expand the set, the adversary will build a tree, starting with the symmetric state that was reached randomly. The degree and the depth of the tree can be limited depending of the symmetry property. In the case of SIMD, none of these attacks are effective for the following reasons:

– First, the modified Davies-Meyer mode of operation means that the compression function does not transform a symmetric state into a symmetric state, but it transforms an almost symmetric state into a somewhat symmetric state. We show in Appendix 7.A that a "somewhat symmetric" output pair can only be used as an "almost symmetric" input pair with a very small probability. This prevents attacks based on building long chains of symmetric messages, like the attacks on CubeHash [ABM+09, FLM10].
– Second, if a pair of almost symmetric states is reached, there is only a single message pair that can be used to reach a symmetric state in the Feistel rounds. This prevents attacks like the herding attack on *Lesamnta* [BDFL10, 8.3], where one reaches a symmetric state and then uses a lot of different messages in order to explore the subset of symmetric outputs.
– Third, the final transformation of SIMD uses the message length as input. Therefore, the symmetry property can only be seen in the output of the hash function with messages of unrealistic length (almost 2^{512} bits for SIMD-256 and almost 2^{1024} bits for SIMD-512). Note that computing the hash of such a message is vastly more expensive than finding a preimage.
– Moreover the symmetry classes do not intersect. It is not possible to build a smaller symmetry classes in order to show collisions in the compression function, as was done for CubeHash [ABM+09, FLM10]. Finding collisions in the compression function using the symmetry property costs $2^{n/2}$. It is more efficient than generic attacks on the compression function, but cannot be used to find collisions in the hash function faster than the birthday attack. We also note that the initial state of the SIMD hash function is not symmetric.

To summarize, reaching a symmetric state in SIMD is far less interesting than reaching a symmetric state in CubeHash or in *Lesamnta*. Table 7.2 gives a comparison of the symmetry properties found in these functions.

Another very important factor is that SIMD is a wide-pipe design. Therefore reaching a symmetric state is about as hard a finding a preimage for the hash function. In the next section, we provide a formal proof that this distinguisher has only a small effect on the security of SIMD.

Table 7.2 Comparison of symmetry properties in several hash functions.

Function	Reach symm. state	Max. length	Max. degree	Free-start Collisions
Lesamnta-512	2^{256}	1	2^{256}	2^{128} (semi-free-start)
CubeHash (symm $C_1..C_7$)	2^{384}	∞	2^{128}	2^{32} (semi-free-start)
CubeHash (symm $C_8..C_{15}$)	2^{256}	∞	1	2^{64} (semi-free-start)
SIMD-512	2^{480}	1	1	2^{256}

We can prove that the hash function behaves as a random oracle under the assumption that the compression function is a weak perfect function having this symmetry property.

7.3 Security with Free-start Distinguishers

In this section, we discuss the security of the prefix-free iteration of non-ideal compression functions. While our primary objective is to show that the distinguisher for the compression function of SIMD presented in Section 7.2 does not void the security proof of SIMD, the reasoning and the proof presented here are pretty general and could very well be adapted to other functions.

Let $\mathcal{H} = \{0,1\}^p$ denote the set of chaining values, $\mathcal{M} = \{0,1\}^m$ denote the set of message blocks, and $\mathcal{F}$ be the set of all functions $\mathcal{H} \times \mathcal{M} \to \mathcal{H}$. Let $F \in \mathcal{F}$ be a compression function taking as input an p-bit chaining value and an m-bit message block. A mode of operation for a hash function $H^{\cdot}$ combined with a compression function F yields a full hash function H^F.

Following [MRH04, CDMP05], we rely on the notion of indifferentiability of systems to reduce the security of SIMD to that of its compression function. The usual way of establishing the soundness of a mode of operation $H^{\cdot}$ is to show that it is indifferentiable from a random oracle. This is done by constructing a simulator $\mathcal{S}$ such that any distinguisher $\mathcal{D}$ cannot tell apart (H^F, F) and $(RO, \mathcal{S})$ without a considerable effort, where RO is a variable-input-length random oracle (VIL-RO, for short). When this is established, it is shown in [MRH04] that any cryptosystem making use of a VIL-RO is not less secure when the random oracle is replaced by the hash function H^F, where F is an ideal compression function (*i.e.* a fixed-input-length random oracle, FIL-RO for short). Informally, if F is ideal (*i.e.* has no special property that a random function would not have), then H^F is secure up to the level offered by the indifferentiability proof. More precisely, if $H^{\cdot}$ is $(t_{\mathcal{D}}, t_{\mathcal{S}}, q_S, q_0, \epsilon)$-indifferentiable from a VIL-RO when the compression function is assumed to be a FIL-RO, then this means that there exists a simulator running in time $t_{\mathcal{S}}$, such that any distinguisher running in time $t_{\mathcal{D}}$ and issuing at most q_S (resp. q_0) queries to the simulator (resp. to the VIL-RO) has success probability at most ϵ.

A property of this methodology is that as soon as the compression function used in a hash function turns out to be non-ideal, then the security argument offered by the indifferentiability proof becomes vacuous. For instance, distinguishers exhibiting a "non-random" behavior of the compression function are usually advertised by their authors to nullify the security proof of the full hash function.

This problematic situation was first tackled by the designers of Shabal, who provided a security proof taking into account the existence of an efficient distinguisher on the internal permutation of their proposal [BCCM$^+$09]. We will follow their track and demonstrate that the security of SIMD can be proved despite the existence of an efficient distinguisher on its compression function.

The mode of operation of SIMD can be "concisely" described as being the wide-pipe prefix-free [2]

2. this is not explicitly stated in the submission document, but SIMD has a different finalization function that

iteration of the compression function. Let H^F therefore denote the *prefix-free* Merkle-Damgård iteration of F. Formally, $g : \{0,1\}^* \rightarrow \mathcal{M}^*$ is a *prefix-free encoding* if for all x, x', $g(x)$ is not a prefix of $g(x')$. The mode of operation $H^{\cdot}$ simply applies the Merkle-Damgård iteration of F to the prefix-free encoding of the message.

The original security argument was that if the internal state and the hash are both p-bit wide, then prefix-free Merkle-Damgård is indifferentiable from a random oracle up to about $2^{p/2}$ queries [CDMP05]. More precisely, we have:

Theorem 7.1. *Prefix-Free Merkle-Damgård is $(t_{\mathcal{D}}, t_{\mathcal{S}}, q_S, q_O, \epsilon)$-indifferentiable from a VIL-RO when the compression function is modeled by a FIL-RO, for any running time $t_{\mathcal{D}}$ of the distinguisher, and $t_{\mathcal{S}} = \mathcal{O}\left((q_O + \kappa \cdot q_S)^2\right)$ where κ is an upper-bound on the size of the queries sent to the VIL-RO. If $q = q_S + \kappa \cdot q_O + 1$, then the success probability of the distinguisher is upper-bounded by:*

$$\epsilon = 8 \cdot \frac{q^2}{2^p}$$

In SIMD where the internal state is $2n$ bits, this ensures the indifferentiability of the whole function up to roughly 2^n queries (if H is indifferentiable up to q queries, then the composition of a truncation that truncates half of the output and of H is also secure up to q queries).

To restore the security argument damaged by the distinguisher, we will show that the prefix-free iteration of a non-ideal compression function is to some extent still indifferentiable from a VIL-RO.

7.3.1 Deterministic Distinguishers for the Compression Function

Let us consider a non-ideal compression function F.

– For instance, it may have *weak states*, that are such that querying F thereon with a well-chosen message block produces a “special” output allowing to distinguish F from random in one query. Known examples include for instance the symmetry on the compression function of *Lesamnta* [BDFL10, 8.3], CubeHash [ABM+09, FLM10], and SIMD (described in Section 7.2).

– But F can also have *bad second-order properties*, meaning that the output of F on correlated input states (with well-chosen message blocks) produces correlated outputs, allowing to distinguish F from random in two queries. A notable example of this property include the existence of differential paths with probability one in the compression function of Shabal [AMM09]. Symmetry properties also give second order relations, which means that *Lesamnta*, CubeHash and SIMD have bad second-order properties as well.

Following the methodology introduced in [BCCM+09], we model this situation by saying that there are two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ such that:

$$\begin{aligned} \forall (h,m) \in \mathcal{H} \times \mathcal{M} : \quad & \mathcal{R}_1(h, m, F(h,m)) = 1 \\ \forall (h_1, h_2, m_1, m_2) \in \mathcal{H}^2 \times \mathcal{M}^2 : \quad & \mathcal{R}_2(h_1, m_1, h_2, m_2, F(h_1, m_1), F(h_2, m_2)) = 1 \end{aligned}$$

We denote by $\mathcal{R}$ the relation formed by the union of $\mathcal{R}_1$ and $\mathcal{R}_2$, and we will denote by $\mathcal{F}[\mathcal{R}]$ the subset of $\mathcal{F}$ such that the above two equations hold. We require the relations to be efficiently checkable, *i.e.* that given h, m and h', it is efficient to check whether $\mathcal{R}_1(h, m, h') = 1$. The relation can thus be used as an efficient distinguishing algorithm that tells $\mathcal{F}[\mathcal{R}]$ apart from $\mathcal{F}$.

effectively acts as a prefix-free encoding.

A *weak state* is a state on which it is possible to falsify the relation $\mathcal{R}_1$. We formally define the set of weak states for $\mathcal{R}_1$ in the following way:

$$\mathcal{W} = \{h \in \mathcal{H} \mid \exists m, h' \in \mathcal{M} \times \mathcal{H} \text{ such that } \mathcal{R}_1(h, m, h') = 0\}$$

$\mathcal{W}$ should be a relatively small subset of $\mathcal{H}$ because the loss of security will be related to the size of $\mathcal{W}$. Moreover, we require than the IV is not $\mathcal{H}$.

In the same vein, a *weak pair* is a pair of states on which it is possible to falsify the relation $\mathcal{R}_2$. We therefore define the set of *weak pairs* for $\mathcal{R}_2$ by an undirected graph $G_{\mathcal{R}_2} = (\mathcal{H}, \mathcal{WP})$, where $\mathcal{WP}$ is defined by:

$$\mathcal{WP} = \left\{h_1 \leftrightarrow h_2 \mid \exists m_1, m_2, h'_1, h'_2 \in \mathcal{M}^2 \times \mathcal{H}^2 \text{ such that } \mathcal{R}_2(h_1, m_1, h_2, m_2, h'_1, h'_2) = 0\right\}$$

Similarly, $\mathcal{WP}$ should be a relatively small subset of $\mathcal{H}^2$ because the security loss will be related to the size of $\mathcal{WP}$. For the sake of expressing things conveniently, we define a variant of the same graph, $G'_{\mathcal{R}_2} = (\mathcal{H} \times \mathcal{M}, \mathcal{WP}')$, where $\mathcal{WP}'$ is defined by:

$$\mathcal{WP}' = \left\{(h_1, m_1) \leftrightarrow (h_2, m_2) \mid \exists h'_1, h'_2 \in \mathcal{H}^2 \text{ such that } \mathcal{R}_2(h_1, m_1, h_2, m_2, h'_1, h'_2) = 0\right\}$$

To simplify the proof we also require that the connected component of $G'_{\mathcal{R}_2}$ have size at most two. This rules out some second-order relations, but it includes for instance the existence of a differential path with probability one with a non-zero difference in the input chaining value, as well as the symmetry in the compression function of SIMD or *Lesamnta*. We expect a similar result with larger connected components, but there will be a loss of security related to their size.

We also require the existence of the *sampling algorithms* for $\mathcal{R}$, namely of two efficient algorithms $\textbf{Sampler}_1$ and $\textbf{Sampler}_2$ such that:

$$\begin{aligned}\textbf{Sampler}_1(h, m): \quad & h' \xleftarrow{\$} \{f(h, m) \mid f \in \mathcal{F}[\mathcal{R}]\}\,; \text{return } h' \\ \textbf{Sampler}_2(h_1, m_1, h_2, m_2, h'_1): \quad & h'_2 \xleftarrow{\$} \{f(h_2, m_2) \mid f \in \mathcal{F}[\mathcal{R}] \text{ and } F(h_1, m_1) = h'_1\}\,; \text{return } h'_2\end{aligned}$$

Informally, the sampling algorithms should produce an output that looks as if it were produced by a random function constrained to conform to $\mathcal{R}$.

7.3.2 Adapting the Indifferentiability Proof to Non-Ideal Compression Functions

We now assume that the compression function is a public function chosen uniformly at random in $\mathcal{F}[\mathcal{R}]$, and for the sake of convenience we will call it a "biased FIL-RO". We show that the prefix-free iteration of biased FIL-RO is indifferentiable from a VIL-RO. In fact, we extend Theorem 7.1 to the case where the compression function is biased.

Theorem 7.2. *Prefix-Free Merkle-Damgård is $(t_\mathcal{D}, t_\mathcal{S}, q_S, q_O, \epsilon)$-indifferentiable from a VIL-RO, when the compression function is modeled by a biased FIL-RO conforming to the relation $\mathcal{R}$ and when the IV is not in $\mathcal{W}$, for any running time $t_\mathcal{D}$ of the distinguisher, and $t_\mathcal{S} = \mathcal{O}\left((q_O + \kappa \cdot q_S)^2\right)$ where κ is an upper-bound on the size of the queries sent to the VIL-RO. If $q = q_S + \kappa \cdot q_o + 1$, then the probability of success of the distinguisher is upper-bounded by:*

$$\epsilon = 16 \cdot \frac{q^2}{2^p} + 4 \cdot |\mathcal{W}| \cdot \frac{q}{2^p} + 4 \cdot |\mathcal{WP}| \cdot \frac{q^2}{(2^p - q)^2}$$

The first term of the expression of ϵ is similar to the result given in Theorem 7.1, when the compression function is ideal (up to a factor two that could be avoided by making the argument slightly more involved). The two other terms reflect the fact that the compression function is biased. The relation induces a security loss if $|\mathcal{W}|$ is at least of order $2^{p/2}$, or if $|\mathcal{WP}|$ is at least of order 2^p. Informally, it seems possible to iterate compression functions having a relatively high bias in a secure way.

Application to Free-start Differential Attacks. Let us assume that the compression function is weak because of the existence of a good differential path with a non-zero difference in the input chaining value. Even if the probability of the differential path is 1, this has a very limited effect on the security of the hash function: this leads to $\mathcal{W} = \emptyset$ and $|\mathcal{WP}| = 2^{p-1}$. The advantage of the distinguisher is at most twice as high, compared to the iteration of an ideal FIL-RO.

Application to SIMD. In SIMD-256 (resp. SIMD-512), the internal state has $p = 512$ bits (resp. $p = 1024$ bits), and the distinguisher of Section 7.2 yields $|\mathcal{W}| = 2^{p/2+16}$, $|\mathcal{WP}| = 2^{p+32}$ (resp. $|\mathcal{W}| = 2^{p/2+32}$, $|\mathcal{WP}| = 2^{p+64}$). Therefore the advantage of any distinguisher in telling apart SIMD-256 from a VIL-RO with q queries is upper-bounded by:

$$\epsilon = 16 \cdot \frac{q^2}{2^p} + 4 \cdot \frac{2^{p/2+16} \cdot q}{2^p} + 4 \cdot 2^{p+32} \cdot \frac{q^2}{(2^p - q)^2}$$

SIMD-256 is then secure up to roughly 2^{256-16} queries (SIMD-512 is secure up to 2^{512-32} queries).

Application to *Lesamnta*. *Lesamnta* follows the prefix-free Merkle-Damgård mode of operation due to its special finalization function. An efficient distinguisher based on symmetries was shown in [BDFL10, 8.3], with $|\mathcal{W}| = 2^{p/2}$ and $|\mathcal{WP}| = 2^{p-1}$. According to Theorem 7.2, the advantage of any distinguisher in telling apart *Lesamnta*-256 from a random oracle with q queries is upper-bounded by:

$$\epsilon = 16 \cdot \frac{q^2}{2^p} + 4 \cdot \frac{2^{p/2} \cdot q}{2^p} + 4 \cdot 2^{p-1} \cdot \frac{q^2}{(2^p - q)^2} \approx 22 \cdot \frac{q}{2^{p/2}}$$

Note that since *Lesamnta* is a narrow-pipe design, we have $p = n$. Our result shows that *Lesamnta* remains secure against generic attacks up to the birthday bound. This is the best achievable proof for *Lesamnta*, since it does not behave as a good narrow-pipe hash function beyond that bound: a dedicated herding attack based on the symmetry property is shown in [BDFL10, 8.3], with complexity $2^{n/2}$.

7.3.3 Proof Sketch of Theorem 7.2

We give a sketch of the proof, while a more formal proof can be found in Appendix 7.C. The heart of the proof is a *simulator* $\mathcal{S}$ which has oracle access to the VIL-RO, and whose task is to simulate a biased FIL-RO. The pseudo-code of the simulator is shown in Algorithm 7.1, but a few preliminary remarks are in order. The simulator maintains a log of the queries it has answered to. This knowledge is maintained under the form of a graph $G = (V, E)$, where the set of vertices V is a subset of $\mathcal{H}$, and where the edges are labelled by message blocks from $\mathcal{M}$. The semantic of this graph is that there is an edge labelled by m between h and h' if the simulator lets the distinguisher know that $f(h, m) = h'$. We will use the notation $h \xrightarrow{m} h'$ to say that there is an

Algorithm 7.1 Pseudo-code of the Simulator $\mathcal{S}_0$, with abort conditions

```
function SIMULATOR($h, m$)
    if there exist a vertex $h' \in V$ and an edge $h \xrightarrow{m} h'$ in $E$ then
        return this $h'$
    else
        return FRESHVALUE($h, m$)
    end if
end function

function FRESHVALUE($h, m$)
    if there exist $(u, v) \leftrightarrow (h, m) \in G'_{\mathcal{R}_2}$ then $(\overline{h}, \overline{m}) \leftarrow (u, v)$
    if $IV \overset{M}{\underset{*}{\rightarrow}} \overline{h} \in \mathsf{Reach}$ then Swap $(h, m)$ and $(\overline{h}, \overline{m})$        ▷ (only if $\overline{h}$ is defined)
    if $IV \overset{M}{\underset{*}{\rightarrow}} h \in \mathsf{Reach}$ then
        if there exist $M'$ such that $M \| m = g(M')$ then
            $h' \leftarrow RO(M')$
        else
            $h' \overset{\$}{\leftarrow} \mathcal{H}$
        end if
        $\overline{h'} \leftarrow$ Sampler$_2$ $(h, m, \overline{h}, \overline{m}, h')$        ▷ (only if $\overline{h}$ is defined)
        if $h' \in \mathcal{W}$ or $h' \in V$ or $\mathsf{Reach} \cup \{h'\}$ covers an edge of $G_{\mathcal{R}_2}$ then Abort
        $\mathsf{Reach} \leftarrow \mathsf{Reach} \cup \left\{h \xrightarrow{m} h'\right\}$
    else
        $h' \leftarrow$ Sampler$_1(h, m)$
        $\overline{h}' \leftarrow$ Sampler$_2$ $(h, m, \overline{h}, \overline{m}, h')$        ▷ (only if $\overline{h}$ is defined)
    end if
    $V \leftarrow V \cup \{h, h', \overline{h}, \overline{h'}\}$        ▷ (only add $\overline{h}$ and $\overline{h'}$ if defined)
    $E \leftarrow E \cup \left\{h \xrightarrow{m} h', \overline{h} \xrightarrow{\overline{m}} \overline{h'}\right\}$        ▷ idem.
    return $h'$ (or $\overline{h'}$ if they were swapped in line 10)
end function
```

edge between h and h' labelled by m in G. Initially, the graph contains only a single vertex IV. The simulator also maintains a subset of V denoted by Reach, consisting of the vertices that are reachable from IV. It also associates to each vertex v in Reach an *ancestor* in Reach. This allows to efficiently reconstruct the sequence of message blocks that maps IV to v, given $v \in \mathsf{Reach}$. We will note $IV \overset{M}{\underset{*}{\rightarrow}} v$ when there is such a path between IV and v. At the beginning, Reach only contains the IV.

Now, a *distinguisher* $\mathcal{D}$ interacts with either H^F and F (we say that it is in the "construction world"), or with RO (which is a VIL-RO) and $\mathcal{S}$ (and we say that it is in the "random oracle world"), and it has to tell in which world it is. More formally, $\mathcal{D}$ is a Turing machine with two interfaces. It should output "1" when H^F and F are answering its oracle queries, and "0" when RO and $\mathcal{S}$ are. Our objective is to show that for all distinguisher $\mathcal{D}$ the following holds for a small ϵ:

$$\left|\mathbb{P}\left[\mathcal{D}^{H^F, F} = 1\right] - \mathbb{P}\left[\mathcal{D}^{RO, \mathcal{S}} = 1\right]\right| \leq \epsilon$$

The main idea of the proof is that our simulator aborts as soon as a state (or pair of states) on which the relation could be falsified becomes reachable (that is to say, a state in $\mathcal{W}$ or a pair of states in $\mathcal{WP}$). Therefore we do not have to study exactly how much information is revealed by the relation. We use the sampling algorithms to simulate the weakness of the compression function, but the adversary can never compare the outputs of the samplers with the output of the VIL-RO, because that would cause the simulator to abort. Moreover, when queried on (h, m), the simulator looks for a symmetric query $(\overline{h}, \overline{m})$ so that the relation $\mathcal{R}_2$ could be falsified (*i.e.* $(h, m) \leftrightarrow (\overline{h}, \overline{m}) \in G'_{\mathcal{R}_2}$). If such a symmetric query exists, the simulator computes both queries at the same time to ensure that they respect the relation $\mathcal{R}_2$.

The proof uses a hybrid argument through a sequence of games, which we summarize.

Game 1. The distinguisher is in the random oracle world. It has access to RO and $\mathcal{S}$.

Game 2. We introduce a dummy *relay algorithm* $\mathcal{T}$, which sits between the distinguisher and the RO. Given a random oracle query from the distinguisher, $\mathcal{T}$ just sends the query to RO, and transmits the answer of RO back to $\mathcal{D}$. This leaves the view of $\mathcal{D}$ unchanged.

Game 3. We modify the simulator $\mathcal{S}$, by making it *abort* in some cases, and report failure. The failure of $\mathcal{S}$ ensures that specific invariants of its internal data structures hold. Specifically, when queried on a reachable chaining value, $\mathcal{S}$ fails if its answer was already "known" by the distinguisher from a previous and different query. Thus, a (reachable) collision on the internal state, or the "connection" to some internal state already known would make $\mathcal{S}$ fail. Moreover, $\mathcal{S}$ will also fail if a weak state becomes reachable, or if the two members of a pair of weak states become reachable (these two events could be observed on the iteration if $\mathcal{S}$ did not fail).

An important point is that when queried on non-reachable chaining values, $\mathcal{S}$ uses the samplers to answer in conformance to the relation $\mathcal{R}$. However, when queried on reachable chaining values, it answers either randomly, or using the VIL-RO for consistency (but the result is still random). Thanks to this, Reach is a random subset of $\mathcal{H}$, and this allows to establish an upper-bound on the probability of the failure, which directly depends on the number of weak states, and on the density of the graph representing the weak pairs of states. The view of $\mathcal{D}$ only changes if $\mathcal{S}$ aborts, and it can be shown that:

$$\mathbb{P}\big[\mathcal{S} \text{ aborts}\big] \leq 4 \cdot \frac{(q_S+1)^2}{2^p} + |\mathcal{W}| \cdot \frac{q_S+1}{2^p} + |\mathcal{WP}| \cdot \frac{(q_S+1)^2}{(2^p - q_S - 1)^2}$$

Game 4. In this game, we modify the relay algorithm and leave the simulator unchanged. Instead of querying the VIL-RO, the new relay algorithm $\mathcal{T}_1$ now applies the Merkle-Damgård construction to the prefix-free encoding of its query. It uses the simulator to evaluate the compression function. Thus the relay algorithm $\mathcal{T}_1$ is essentially the same as $H^{\cdot}$, except that it is based on the simulator $\mathcal{S}$ instead of random function F.

The key argument is that the answers of $\mathcal{S}$ are consistent with those of RO: when $\mathcal{S}$ receives a sequence of queries corresponding to the prefix-free encoding of a message, it decodes it, queries the VIL-RO on the decoded message, and returns the answer of the VIL-RO. Another important detail is that before $\mathcal{S}$ fails, Reach exactly describes the reachable chaining values, and forms a tree rooted in IV. This latter property means that when a sequence of queries completes the prefix-free encoding of a message, then the message can be decoded in a unique way, which is critical in order to keep the simulator consistent with the VIL-RO.

So, all-in-all, the VIL-RO gives the same answers in Games 3 and 4, the simulator is consistent with the VIL-RO in both games, and conforms to the relation in both games. Therefore, when proceeding from Game 3 to Game 4, the view of the distinguisher only changes when $\mathcal{S}$ fails in either one of the games, but it fails more often in game 4 (because it also receives the queries of the relay algorithm).

Game 5. In this game, the VIL-RO is removed completely and the new simulator $\mathcal{S}_1$ always chooses a random p-bit response, even in situations where $\mathcal{S}$ would have consulted the VIL-RO. We also remove all the failure conditions from the new simulator $\mathcal{S}_1$.

The view of the distinguisher may only change if $\mathcal{S}$ would have failed (because now $\mathcal{S}_1$ does not).

Game 6. This is the final game of our argument. Here we finally replace the simulator $\mathcal{S}_1$ with the biased FIL-RO. Since the relay algorithm $\mathcal{T}_1$ simply implemented the prefix-free Merkle-Damgård construction, the view of the distinguisher is in fact in the construction world.

All-in-all, we find that the advantage of the distinguisher is upper-bounded by:

$$\epsilon = 2 \cdot \mathbb{P}\big[\mathcal{S} \text{ fails in } G_3\big] + 2 \cdot \mathbb{P}\big[\mathcal{S} \text{ fails in } G_4\big]$$

And for the sake of obtaining a simpler expression, since $\mathcal{S}$ fails more often in G_4 than in G_3, we find:

$$\epsilon \leq 4 \cdot \mathbb{P}\big[\mathcal{S} \text{ fails in } G_4\big]$$

This yields the result announced in the theorem.

7.4 On Differential Attacks against SIMD

In this section we will present our results concerning differential paths in SIMD. Using Integer Linear Programming, we show that if there is a difference in the message, then the probability of the path will be at most of the order of $2^{-n/2}$. We stress that this result is not tight, but the computational power needed to improve the bound using this technique grows exponentially.

Related Work. The first attempt to avoid differential attack in a SHA/MD-like hash function was proposed in [JP06], where Jutla and Patthak described a linear code similar to the message expansion of SHA-1, and proved that it has a much better minimal distance than the original SHA-1 message expansion. They proposed to use SHA-1 with this new message expansion and called the new design SHA-1-IME.

Our Results. The design of SIMD follows the same idea, using a strong message expansion with a high minimal distance. In this section we show that we can prove the security of SIMD more rigorously than the security of SHA-1-IME. While the security of SHA-1-IME is based on the heuristic assumption that the path is built out of local collisions, our proof gives an upper bound on the probability of *any* differential characteristic with a non-zero difference in the message.

Our results prove the following: for any message pair with a non-zero difference, the probability of going from an input difference Δ_i to an output difference Δ_o is bounded by 2^{-132} for SIMD-256, and 2^{-253} for SIMD-512.

7.4.1 Modeling Differential Paths

To study differential attacks against SIMD, we assume that the adversary builds a differential path. The differential path specifies the message difference and the state difference at each step. For each step i, we study the probability $p(i)$ that the new step difference conforms to the differential path, assuming that the previous state difference and the message difference conforms to the path, but that the values themselves are random. Since SIMD heavily uses modular additions, our analysis is based on a signed differential, as used by Wang *et al.*. A signed difference gives better differential paths than an XOR difference when two active bits cancel each other out: with an XOR difference this gives a probability 1/2, but with a signed difference we have a probability 1 if the signs are opposed.

To study differential paths, we will consider the inner state of SIMD, and the Boolean functions $\phi^{(i)}$. A state bit $A_j^{(i)}$ is called *active* if it takes two different values for a message pair following the differential path. Similarly, a Boolean function is called active if at least one of its inputs is *active*. A differential path consists of a set of active message bits, active state bits, active Boolean functions, and the sign of each active element. We assume that the adversary first builds such a differential path, and then looks for a conforming pair of message and chaining value. If we disregard the first and last rounds, each Boolean function has three inputs, and each state bit enters three Boolean functions. We use this simplification in Section 7.4.4.

7.4.2 The Message Expansion

Table 7.3 shows the minimal distance of the message expansion of SIMD compared to the message expansion of SHA-1 and SHA-1-IME. We know that the message expansion of SIMD has a minimal distance of 520, but this is the Hamming distance, *i.e.* an XOR difference. Since we assume that the adversary will use a signed difference to build the differential path, we must study the distance of the code when the difference is given by signed binary representation. The problem is that consecutive active bits might be used as a single modular difference. For instance `0b0111` and `0b1001` differ in three bit positions, but the modular difference is only 2^1 and it can introduce a single difference in the output of a modular addition.

To compute the minimal number of modular differences introduced by the message, we use the non-adjacent form (NAF). The NAF is a signed binary representation, *i.e.* a sum of signed powers of two. It is unique and can be efficiently computed. The good property of the NAF is that it is a signed binary representation of minimal weight. For each pair of inputs to the inner code, we can compute the NAF of the difference, and we see that the minimal distance is 4. This means that each active word in the output of the Reed-Solomon code will introduce at least 4 differences in the state, even when we consider a differential attack using modular difference.

However, two outputs of the inner code are packed together into a 32-bit word. If we have a difference in the MSB of the low order word and in the LSB of the high order word, they can collapse to a single modular difference. In Section 7.4.4, we disregard this property and we just consider that the message introduces 520 differences through the message expansion. However, in Section 7.4.5, our model will account for that.

7.4.3 Structure of a Differential Path

The basic idea of our analysis is to use the lower bound on the number of active message bits to derive a lower bound on the number of active state bits. Each message difference must either introduce a new difference in the state, or cancel the propagation of a previous state difference. A single difference propagates to between 2 and 5 differences, depending on whether the Boolean functions absorb it or let it go through. This means that a collision corresponds to between 3 and 6 message differences.

For instance, if a difference is introduced in the state $A_1^{(5)}$ by $W_1^{(5)}$, it will appear in $A_1^{(5)}$, $B_1^{(6)}$, $C_1^{(7)}$, $D_1^{(8)}$. Each of the Boolean function $\phi_1^{(6)}, \phi_1^{(7)}, \phi_1^{(8)}$ can either absorb it or pass it. This difference will propagate to $A_0^{(6)}$, and to $A_1^{(9)}$. Moreover, it can propagate to $A_1^{(6)}$, $A_1^{(7)}$ and $A_1^{(8)}$

Table 7.3 Minimal distance of the message expansion.

	Message block	Expanded message	Minimal distance
SHA-1 [1]	512 bits	1920 bits	25 bits
SHA-1-IME [1]	512 bits	1920 bits	75 bits
SIMD-256/16 [2]	512 bits	2048 bits	260 bits
SIMD-512/16 [2]	1024 bits	4096 bits	516 bits
SIMD-256	512 bits	4096 bits	520 bits
SIMD-512	1024 bits	8192 bits	1032 bits

[1] SHA-1 and SHA-1-IME codes are projected to the last 60 words.
[2] SIMD-$n/16$ is a reduced version using a single copy of the encoded message.

if the Boolean functions do not absorb it. Up to five active message bits can be used to cancel this propagation: $W_1^{(4)}$, $W_1^{(8)}$, $W_0^{(5)}$, and possibly $W_1^{(5)}$, $W_1^{(6)}$, $W_1^{(7)}$ if the corresponding Boolean functions are not absorbing.

We consider two parts of the compression function: the computation of ϕ, and the modular sum. In order to study the probabilities associated with these computations, we will count the conditions needed for a message pair to follow the characteristic.

ϕ-conditions. The Boolean functions MAJ and IF used in SIMD can either absorb or pass differences. When there is a single active input, the probability to absorb and to pass is 1/2. Each time a state difference enters a Boolean function, the differential characteristic specifies whether the difference should be passed or absorbed, and this gives one condition if the Boolean functions have a single active input. Thus, each isolated difference in the state will account for 3 ϕ-conditions: one for each Boolean function they enter. For instance, a difference in $A_1^{(4)}$ generates conditions for $\phi_1^{(6)}$, $\phi_1^{(7)}$, $\phi_1^{(8)}$.

$\boxplus$-conditions. When a difference is introduced in the state, it has to come from one of the inputs of the round function:

$$A_j^{(i)} = \left(D_j^{(i-1)} \boxplus W_j^{(i)} \boxplus \phi^{(i)}(A_j^{(i-1)}, B_j^{(i-1)}, C_j^{(i-1)})\right)^{\lll s^{(i)}} \boxplus \left(A_{p^{(i)}(j)}^{(i-1)}\right)^{\lll r^{(i)}}$$

The round function is essentially a sum of 4 terms, and the differential characteristic will specify which input bits and which output bits are active. Thus, the differential characteristic specifies how the carry should propagate, and this gives at least one condition per state difference.

In the end, a state difference accounts for 4 conditions.

7.4.4 Heuristics

We first give some results based on heuristics. We assume that the adversary can find message pairs that give a minimal distance in the expanded message, and we allow him to add some more constraints to the expanded message. Note that finding a message pair with a low difference in the expanded message is already quite difficult with the message expansion of SIMD.

Heuristic I assumes that the adversary can find message pairs with minimal distance, but no other useful property. The adversary gets a message pair with minimal distance, and connects the dots to build a differential characteristic.

Heuristic II assumes that the adversary can find message pairs with minimal distance and controls the relative positions of the message difference. He will use that ability to create local collisions.

Heuristic III assumes that the adversary can find a message pair with any message difference, limited only by the minimal weight of the code. He will cluster local collisions to avoid many conditions.

Heuristic I. In this section, we assume that the adversary can find a message pair such that the expanded messages reach the minimal distance of the code, but we assume that the message pair has no further useful properties.

In this case, this adversary gets a message pair with a small difference and he has to connect the dots to build a differential path. This is somewhat similar to the attacks on MD4 [WLF$^+$05]: the messages are chosen so as to make a local collision in the last round, and the adversary has to connect all the remaining differences into a path with a good probability.

It seems safe to assume that such a differential path will at least have as many active state bits as active message bits. Since an isolated difference in the state costs 4 conditions, we expect at least 2080 conditions (resp. 4128 for SIMD-512), which is very high.

This shows that the adversary needs some control over the expanded message. If he wants to succeed, he needs to find message pairs with some extra properties.

Heuristic II. We now assume that the adversary can force some structure in the expanded message difference. Namely, he can choose the relative location of the differences in the expanded message. Since the probability of the path is essentially given by the number of active bits in the state, the path should minimize this. This is achieved with local collisions, and each local collision will use as many message differences as possible. Due to the structure of the round function of SIMD, a local collision can use between 3 and 6 message differences, depending on whether the Boolean functions absorb or pass the differences. In order to minimize the number of state differences, the path will make all the Boolean functions pass the differences, yielding six message differences per state difference. This is somewhat counter-intuitive because most attacks try to minimize the propagation of differences by absorbing them. However, in our case it is more efficient to let the differences go through the Boolean functions, and to use more message differences to cancel them, because we have a lower bound on the number of message differences.

Since the adversary only controls the relative position of the message differences, we assume that most local collisions will be isolated, so that each local collision gives 4 conditions. Thus, a differential is expected to have at least $520 \times 4/6 \approx 347$ conditions (688 for SIMD-512). This leaves a significant security margin, and even if the adversary can use message modifications in the first 16 rounds, it can only avoid half of those conditions.

This can be compared to the attacks on SHA-1 [CJ98, WYY05b]. These attacks are based on local collisions, but we do not know how to find a message pair which would have both minimal distance and yield a series of local collisions in SHA-1. Instead, attacks on SHA-1 use the fact that the message expansion is *linear* and *circulant*: given a codeword, if we shift it by a few rounds we get another valid codeword and similarly if we rotate each word we get another valid codeword. Then we can combine a few rotated and/or shifted codewords so as to build local collisions. The attacks on SHA-1 start with a codeword of minimal distance, and combines 6 rotated versions. Thus the weight of the actual expanded message difference used in the attack is six times the minimal weight of the code.

Note that message expansion of SIMD is more complex than the one from SHA-1, and it seems very hard to find this kind of message pairs in SIMD. Moreover, the trick used in SHA-1 cannot be used here because the message expansion is neither linear nor circulant.

Algorithm 7.2 Linear Program

Minimize $S + \alpha - \beta$ with the constraints:

$$3S = \alpha + \beta + \gamma \tag{7.1}$$

$$520 \leq 3S + \alpha \tag{7.2}$$

$$\gamma \leq \beta \leq \alpha \tag{7.3}$$

$\alpha \geq 0$	is the number of Boolean functions with at least one active input
$\beta \geq 0$	is the number of Boolean functions with at least two active inputs
$\gamma \geq 0$	is the number of Boolean functions with at least three active inputs
$S \geq 0$	is the number of active state bits

Heuristic III. We now remove all heuristic assumptions and we try to give a bound on *any* differential trail. However, to keep this analysis simple, we still disregard the specificities of the first round, and the fact that one can combine some of the message differences.

The adversary will still use local collisions to minimize the number of differences in the state, but he will also try to reduce the number of conditions for each local collision by clustering them. We have seen that an isolated state difference costs 4 conditions, but if two state differences are next to each other, the cost can be reduced when using a signed difference. For instance, if two inputs of the MAJ function are active, the adversary does not have to pay any probability: if both active inputs have the same sign, then the output is active with the same sign, but if the inputs have opposite signs then the output will be inactive. In this section we consider that a Boolean function with more than one active input does not cost any probability.

Thus, the best strategy for the adversary is to place the state differences so that each active Boolean function has two active inputs, in order to avoid any ϕ-conditions. Each state difference costs only one $\boxplus$-condition, and gets 4.5 message differences (these message differences corresponding to the Boolean functions are shared between two Boolean functions). This gives a lower bound of 116 conditions.

More rigorously, this can be described by a linear program, as shown in Algorithm 7.2. Equation (7.1) comes from counting the number of active inputs to the Boolean functions in two different ways, while Equation (7.2) counts the number of message differences that can be used. The objective value $S + \alpha - \beta$ counts the conditions: one for each state difference, plus one for each Boolean function with exactly one active input. The optimal solution to this program is $520/4.5 \approx 115.55$.

In the next section we will see how to improve this bound and get a bound on the probability of any differential path.

Comparison with SHA-1-IME. The security of SHA-1-IME is based on a heuristic that is quite similar to our Heuristic I. Jutla and Patthak assume that the adversary will use the same technique as the attacks on SHA-1, *i.e.* create local collisions using the fact that the code is linear and circulant. They deduce that the probability of a differential characteristic will be about $2^{75\times 2.5}$. They implicitly assume that the adversary cannot find minimal codewords that would already give local collisions. Our Heuristic II assumes that the adversary can find such codewords, and if we apply it to SHA-1-IME, it would only guarantee that we have at least 13 local collisions (each local collision accounts for 6 message differences). Since a local collision in SHA-1 has an average probability of $2^{-2.5}$, this would only prove that an attack has at least a complexity $2^{12\times 2.5} = 2^{32.5}$.

This shows that our Heuristics II and III are much weaker than the heuristic used in SHA-1-IME.

7.4.5 Upper Bounding the Probability of a Differential Path

The bound given by Heuristic III is slightly lower than $n/2$ so we would like to improve it. To find a better bound, we will follow the approach of Linear Algorithm 7.2. Note that in the optimal solution, all the Boolean functions have either zero or two active inputs, but it is unlikely that such a path actually exists because of the way the Boolean functions share inputs. In order to remove some impossible solutions, we use a more detailed modeling of differential paths where each individual state bit is treated separately. This also allows us to express some extra constraints that will help to improve the lower bound.

Algorithm 7.3 Integer Linear Program (simplified)

Minimize $\sum S_i^{(j)[k]} + \sum \alpha_i^{(j)[k]} - \sum \beta_i^{(j)[k]}$ **with the constraints:**

$$S_i^{(j-1)[k]} + S_i^{(j-2)[k]} + S_i^{(j-3)[k]} = \alpha_i^{(j)[k]} + \beta_i^{(j)[k]} + \gamma_i^{(j)[k]} \tag{7.1'}$$

$$W_i^{(j)[k]} \leq S_i^{(j)[k+s_j]} + S_i^{(j-4)[k-r_j]} + S_{p_j(i)}^{(j-1)[k-r_j+s_j]} + \alpha_i^{(j)[k]} \tag{7.2'}$$

$$\gamma_i^{(j)[k]} \leq \beta_i^{(j)[k]} \leq \alpha_i^{(j)[k]} \tag{7.3'}$$

$$\sum_{k=0}^{15} W_i^{(j)[k]} \geq 4Y_{P_1(i,j)} \qquad \sum_{k=16}^{31} W_i^{(j)[k]} \geq 4Y_{P_0(i,j)} \tag{7.4}$$

$$\sum Y_i \geq 65 \tag{7.5}$$

$\alpha_i^{(j)[k]} \in \{0,1\}$	is true iff $\phi_i^{(j)[k]}$ has at least one active input
$\beta_i^{(j)[k]} \in \{0,1\}$	is true iff $\phi_i^{(j)[k]}$ has at least two active input
$\gamma_i^{(j)[k]} \in \{0,1\}$	is true iff $\phi_i^{(j)[k]}$ has at least three active input
$S_i^{(j)[k]} \in \{0,1\}$	is true iff the state bit $A_i^{(j)[k]}$ is active
$W_i^{(j)[k]} \in \{0,1\}$	is true iff the expended message bit $W_i^{(j)[k]}$ is active
$Y_i \in \{0,1\}$	is true iff the word i is active in the output of the NTT
P_0 and P_1	encode the permutations inside the message expansion

Constraints related to the message expansion. We know that the message expansion gives at least 520 differences in the expanded message, but there are some constraints on the positions of these differences. Namely, we have at least 65 active words in each copy of the message, and each active word has at least 4 active bits. For instance, a difference pattern with 3 active bits in each word would have 768 bit differences, but it is not a valid pattern. Moreover, the active words in both copies have to be the same up to the permutations in the message expansion. To include these constraints in our model, we add a set of binary variables Y_i which encode whether word i is active in the output of the NTT. This is modeled by Equations (7.4) and (7.5). Note that this still allows many difference patterns that cannot be the output of a real message pair.

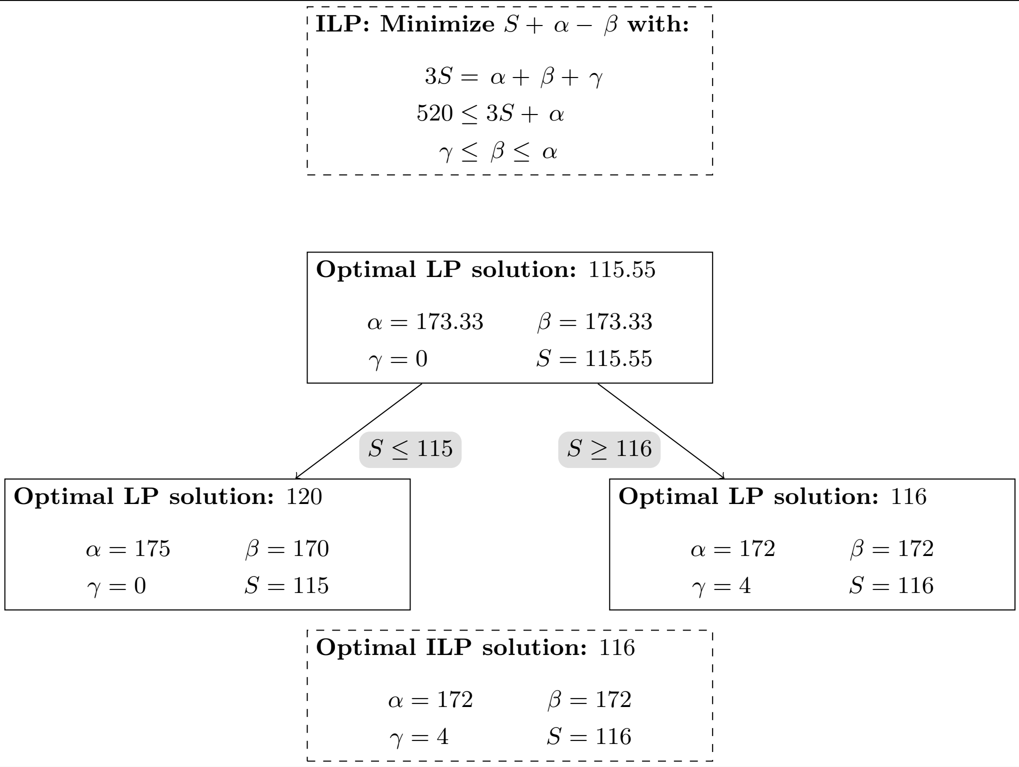
Better cost estimation. In Algorithm 7.2, we only count a condition for the Boolean functions with a single active input. In fact, if we look at the truth table of the Boolean functions we see that the IF function still needs a condition when inputs 1 and 2, or 1 and 3 are active. Since we are using distinct variables for each of these inputs, we can include this in our description.

We can write all these constraints as a huge optimisation problem with approximately 30,000 variables and 80,000 equations, but we need some tool to find the optimal solution of the system, or at least find a lower bound. We decided to write our problem as an Integer Linear Program.

Integer Linear Programming. Integer Linear Programming (ILP) is a generalisation of Linear Programming (LP) where some variables are restricted to integer values. While LP is solvable in polynomial time, ILP is NP-complete. ILP solvers usually use some variants of the branch-and-bound algorithm. In the case of minimization problem, the branch-and-bound algorithm computes a lower bound to the optimal solution and incrementally raises this lower bound. Meanwhile, non-optimal solutions give an upper bound, and when the two bounds meet, the search is over.

To compute a lower bound, the problem is relaxed by considering all variables as real numbers instead of integers. This gives a Linear Program which can be solved efficiently, and the optimal solution of the Linear Program is a lower bound of the Integer Linear Program. To improve this lower bound, the search space is divided into two or more subspaces, and a lower bound is computed recursively for each subproblem. For instance, Figure 7.1 shows how to solve Algorithm 7.2 as an ILP.

Figure 7.1 Solving Program 7.2 as a ILP. The problem is divided into two subproblems: $S \leq 115$ ans $S \geq 116$. For each of these subproblems, the optimal solution of the relaxed problem is integral so this gives us the optimal solution of the ILP.


ILP: Minimize $S + \alpha - \beta$ **with:**
$3S = \alpha + \beta + \gamma$
$520 \leq 3S + \alpha$
$\gamma \leq \beta \leq \alpha$
Optimal LP solution: 115.55
$\alpha = 173.33$ $\beta = 173.33$
$\gamma = 0$ $S = 115.55$
$S \leq 115$
$S \geq 116$
Optimal LP solution: 120
$\alpha = 175$ $\beta = 170$
$\gamma = 0$ $S = 115$
Optimal LP solution: 116
$\alpha = 172$ $\beta = 172$
$\gamma = 4$ $S = 116$
Optimal ILP solution: 116
$\alpha = 172$ $\beta = 172$
$\gamma = 4$ $S = 116$

Results. A simplified version of the ILP is given by Algorithm 7.3. The first equations and the objective value mirrors Algorithm 7.2, but use many variables to allow for more precise extra constraints. The full program has 28,576 variables and 80,162 equations for SIMD-256. We used the solver SYMPHONY, an open-source solver for mixed-integer linear programs, available at `http://www.coin-or.org/SYMPHONY/`. The solver could not find an optimal solution to the program, but it gave a reaches an interesting lower bound after some time: a differential path for SIMD-256 has at least 132 conditions, while a differential path for SIMD-512 has at least 253. The computation for SIMD-512 took one month on a bi-quadcore machine.

Summary. The optimal strategy of the adversary is to use local collisions (avoiding any difference propagation) and to cluster the local collisions so as to avoid most conditions. Our modeling allows the adversary to do this because he can choose the message difference and the expanded message difference independently, and he can position the differences arbitrarily in the inner code. However, this is not possible in practice, and most solutions of the Integer Linear Program will require an expanded message difference that is not actually feasible. It should also be noted that we do not model the sign of the differences, and we always assume that the sign is correct when two differences cancel out.

Therefore, we expect that the best differential path in SIMD is much worse that the optimal solution of our Integer Linear Program. Moreover, the program is too large to be solved to optimality, and we only have a lower bound on the number of conditions (this lower bound keep improving if we let the solver run).

Limitations.

About Message Modifications. When we consider the Heuristic III, our proof does not leave enough margin to account for message modifications. However the mode of operation of the compression function is designed to make message modification difficult, by XORing the message and the chaining value in the very beginning. This prevents usual message modification techniques, because the adversary has to commit to some message before he can begin the real computation of the compression function.

We note that given a message M, one can compute a new message M^* so that the expanded message W^* is identical to W in the first steps, up to almost 8 steps. However in order to keep the same state in the Feistel rounds, one has to counter the modification of the message by a modification of the input chaining value. Therefore it is only applicable to free-start attacks. Since SIMD is a wide-pipe design, free-start attacks on the compression function cannot be turned into attacks on the hash function.

Redundant Conditions. There might be some redundant ϕ-conditions in a differential path. As opposed to MD4 or MD5, we can never have the same condition for two different Boolean functions (because of the rotations), but there might still be some redundant conditions. However, since most of the conditions are $\boxplus$-conditions, we believe this is negligible.

7.5 Security Status of SIMD

7.5.1 On the Symmetry-based Distinguisher

The distinguisher of Section 7.2 shows that the compression function of SIMD is not ideal. It does not affect the security of the hash function, but it is nonetheless an unwanted property. Since this distinguisher is based on symmetry properties, it is easy to avoid this property by slightly changing the design. Therefore, we plan to tweak the SIMD design by adding non-symmetric constants, if given such an opportunity. We also note that other SHA-3 candidates are in a similar situation:

– CubeHash has strong symmetry properties in its round transformation [ABM+09, FLM10]. It is thought that since the initial state in not symmetric, it is not possible to reach a symmetric state.
– Shabal has strong distinguishers on its compression function: there are differential paths with probability 1 [AMM09], and the inverse permutation does not have full diffusion (some input bits do not depend on all output bits). The Shabal team has shown that these distinguishers do not affect the security [BCCM+09].

Countermeasures. An interesting way to avoid the symmetry properties would be to add a counter to the expanded message after the multiplication by a constant (second layer of the message expansion). This would ensure that each expanded message word has a different value modulo 185 (respectively modulo 223), and it prevents equality constraints between the expanded message words.

7.5.2 On Differential Attacks

Concerning differential attacks, our results are two-fold:

1. A differential path with a non-zero difference in the input chaining value does not affect the security of the hash function because it is wide-pipe
2. A differential path with a non-zero difference in the message cannot have a high success probability, because of the strong message expansion.

This shows that successful attacks on the hash function based on differential properties are very unlikely.

7.A Study of the Symmetry Classes

In this section we study the sets of messages that can be used for the symmetry property of SIMD by solving the equation on the NTT. We give an explicit description of the sets, and we show that for a given symmetry relation, a pair of output cannot be used as input for the symmetry property.

7.A.1 Symmetries for SIMD-256

There are three symmetry classes in SIMD-256.

Class 1: $\overset{\leftrightarrow}{\bullet}$ **($y_i = y'_{i\oplus 2}$, $W_i = W'_{i\oplus 1}$).** For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned} M_{\alpha,\beta} &= 162 \times e_{31} + 241 \times e_{63} + \alpha \times e_0 + \beta \times e_{32} \\ M'_{\alpha,\beta} &= 162 \times e_{31} + 241 \times e_{63} + \alpha \times e_0 - \beta \times e_{32} \end{aligned}$$

Note that if $\beta = 1$ we have $-\beta = -1$ and the corresponding M' is not a valid message. So we have only $256 \cdot 255$ valid message pairs.

When $\beta = 0$, we have $M_{\alpha,\beta} = M_{\alpha,\beta}$ and this gives a symmetric message instead of a pair of symmetric message.

Class 2: $\underset{\leftrightarrow}{\bullet}$ **($y_i = y'_{i\oplus 4}$, $W_i = W'_{i\oplus 2}$).** For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 4:

$$\begin{aligned} M_{\alpha,\beta,\gamma,\delta} &= 55 \times e_{15} + 232 \times e_{31} + 37 \times e_{47} + 16 \times e_{63} + \alpha \times e_0 + \beta \times e_{32} + \gamma \times e_{16} + \delta \times e_{48} \\ M'_{\alpha,\beta,\gamma,\delta} &= 55 \times e_{15} + 232 \times e_{31} + 37 \times e_{47} + 16 \times e_{63} + \alpha \times e_0 + \beta \times e_{32} - \gamma \times e_{16} - \delta \times e_{48} \end{aligned}$$

Class 3: $\underset{\leftrightarrow}{\overset{\leftrightarrow}{\bullet}}$ **($y_i = y'_{i\oplus 6}$, $W_i = W'_{i\oplus 3}$).** For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned} M_{\alpha,\beta} &= 212 \times e_{15} + 181 \times e_{31} + 139 \times e_{47} + 20 \times e_{63} + \alpha \times e_0 + \beta \times e_{32} \\ M'_{\alpha,\beta} &= 212 \times e_{15} + 181 \times e_{31} + 139 \times e_{47} + 20 \times e_{63} + \alpha \times e_0 - \beta \times e_{32} \end{aligned}$$

Input and Output Pairs. Let M, M' be a pair of messages in one of these classes. Without loss of generality, we denote the symmetry class by $\overset{\leftrightarrow}{\bullet}$.

Let h, h' be a pair of chaining values that can used with this message, *i.e.* $h' = \overleftrightarrow{h \oplus M} \oplus M'$. We denote the inputs to the Feistel compression part by $\mathcal{S}^{(0)} = A^{(0)}_{[0,1,2,3]}, B^{(0)}_{[0,1,2,3]}, C^{(0)}_{[0,1,2,3]}, D^{(0)}_{[0,1,2,3]}$ (respectively $\mathcal{S}'^{(0)}$). Similarly, $\mathcal{S}^{(31)}$ is the state after the 32 Feistel rounds using the message, and $\mathcal{S}^{(35)}$ if the final state after the feed-forward.

We can express $D^{(35)}_3$ in terms of $\mathcal{S}^{(31)}$:

$$\begin{aligned} D'^{(35)}_3 &= \left(\left(D'^{(31)}_3 \boxplus IV'_3 \boxplus \mathsf{IF}(A'^{(31)}_3, B'^{(31)}_3, C'^{(31)}_3) \right)^{\lll 13} \boxplus A'^{(31)\lll 4}_0 \right)^{\lll 13} \\ \overleftrightarrow{D}^{(35)}_3 &= \left(\left(\overleftrightarrow{D}^{(31)}_3 \boxplus \overleftrightarrow{IV}_3 \boxplus \mathsf{IF}(\overleftrightarrow{A}^{(31)}_3, \overleftrightarrow{B}^{(31)}_3, \overleftrightarrow{C}^{(31)}_3) \right)^{\lll 13} \boxplus \overleftrightarrow{A}^{(31)\lll 4}_0 \right)^{\lll 13} \end{aligned}$$

Since $\mathcal{S}^{(31)}$ and $\mathcal{S}'^{(31)}$ are symmetric, we have:

$$A_3'^{(31)} = \overleftrightarrow{A^{(31)}}_3 \quad B_3'^{(31)} = \overleftrightarrow{B^{(31)}}_3 \quad C_3'^{(31)} = \overleftrightarrow{C^{(31)}}_3 \quad D_3'^{(31)} = \overleftrightarrow{D^{(31)}}_3 \quad A_0'^{(31)} = \overleftrightarrow{A^{(31)}}_0$$

Therefore, the difference between $D_3'^{(35)}$ and $\overleftrightarrow{D}_3^{(35)}$ comes from the difference between IV_3' and $\overleftrightarrow{IV}_3$. By looking at those differences, we can see that they are not compatible, because of the rotation by 26 bits:

Class 1 $IV_3' \oplus \overleftrightarrow{IV}_3 = 0$, $D_3'^{(35)} \oplus \overleftrightarrow{D}_3^{(35)} = 241 \times 2^{24}$
Class 2 $IV_3' \oplus \overleftrightarrow{IV}_3 = 55 \times 2^{24}$, $D_3'^{(35)} \oplus \overleftrightarrow{D}_3^{(35)} = 16 \times 2^{24}$
Class 3 $IV_3' \oplus \overleftrightarrow{IV}_3 = 212 \times 2^{24}$, $D_3'^{(35)} \oplus \overleftrightarrow{D}_3^{(35)} = 20 \times 2^{24}$

7.A.2 Symmetries for SIMD-512

There are seven symmetry classes for SIMD-512.

Class 1 ($y_i = y'_{i\oplus 2}$, $W_i = W'_{i\oplus 1}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned} M_{\alpha,\beta} &= 180 \times e_{63} + 241 \times e_{127} + \alpha \times e_0 + \beta \times e_{64} \\ M'_{\alpha,\beta} &= 180 \times e_{63} + 241 \times e_{127} + \alpha \times e_0 - \beta \times e_{64} \end{aligned}$$

Class 2 ($y_i = y'_{i\oplus 4}$, $W_i = W'_{i\oplus 2}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 4:

$$\begin{aligned} M_{\alpha,\beta,\gamma,\delta} &= 74 \times e_{31} + 232 \times e_{63} + 218 \times e_{95} + 16 \times e_{127} + \alpha \times e_0 + \beta \times e_{64} + \gamma \times e_{32} + \delta \times e_{96} \\ M'_{\alpha,\beta,\gamma,\delta} &= 74 \times e_{31} + 232 \times e_{63} + 218 \times e_{95} + 16 \times e_{127} + \alpha \times e_0 + \beta \times e_{64} - \gamma \times e_{32} - \delta \times e_{96} \end{aligned}$$

Class 3 ($y_i = y'_{i\oplus 6}$, $W_i = W'_{i\oplus 3}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned} M_{\alpha,\beta} &= 58 \times e_{31} + 16 \times e_{63} + 150 \times e_{95} + 122 \times e_{127} + \alpha \times e_0 + \beta \times e_{64} \\ M'_{\alpha,\beta} &= 58 \times e_{31} + 16 \times e_{63} + 150 \times e_{95} + 122 \times e_{127} + \alpha \times e_0 - \beta \times e_{64} \end{aligned}$$

Class 4 ($y_i = y'_{i\oplus 8}$, $W_i = W'_{i\oplus 4}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 8:

$$\begin{aligned} M_{\alpha,\beta,\gamma,\delta,\epsilon,\zeta,\eta,\theta} &= 211 \times e_{15} + 8 \times e_{31} + 199 \times e_{47} + 234 \times e_{63} + 116 \times e_{79} + 32 \times e_{95} + 111 \times e_{111} + 16 \times e_{127} \\ &\quad + \alpha \times e_0 + \beta \times e_{64} + \gamma \times e_{32} + \delta \times e_{96} + \epsilon \times e_{16} + \zeta \times e_{48} + \eta \times e_{80} + \theta \times e_{112} \\ M'_{\alpha,\beta,\gamma,\delta,\epsilon,\zeta,\eta,\theta} &= 211 \times e_{15} + 8 \times e_{31} + 199 \times e_{47} + 234 \times e_{63} + 116 \times e_{79} + 32 \times e_{95} + 111 \times e_{111} + 16 \times e_{127} \\ &\quad + \alpha \times e_0 + \beta \times e_{64} + \gamma \times e_{32} + \delta \times e_{96} - \epsilon \times e_{16} - \zeta \times e_{48} - \eta \times e_{80} - \theta \times e_{112} \end{aligned}$$

Class 5 ($y_i = y'_{i\oplus 10}$, $W_i = W'_{i\oplus 5}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned} M_{\alpha,\beta} &= 195 \times e_{15} + 237 \times e_{31} + 154 \times e_{47} + 254 \times e_{63} + 70 \times e_{79} + 40 \times e_{95} + 121 \times e_{111} + 195 \times e_{127} \\ &\quad + \alpha \times e_0 + \beta \times e_{64} \\ M'_{\alpha,\beta} &= 195 \times e_{15} + 237 \times e_{31} + 154 \times e_{47} + 254 \times e_{63} + 70 \times e_{79} + 40 \times e_{95} + 121 \times e_{111} + 195 \times e_{127} \\ &\quad + \alpha \times e_0 - \beta \times e_{64} \end{aligned}$$

Class 6 ($y_i = y'_{i\oplus 12}$, $W_i = W'_{i\oplus 6}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 4:

$$\begin{aligned}
M_{\alpha,\beta,\gamma,\delta} &= 251 \times e_{15} + 35 \times e_{31} + 36 \times e_{47} + 223 \times e_{63} + 57 \times e_{79} + 159 \times e_{95} + 0 \times e_{111} + 114 \times e_{127} \\
&\qquad + \alpha \times e_0 + \beta \times e_{64} + \gamma \times e_{32} + \delta \times e_{96} \\
M'_{\alpha,\beta,\gamma,\delta} &= 251 \times e_{15} + 35 \times e_{31} + 36 \times e_{47} + 223 \times e_{63} + 57 \times e_{79} + 159 \times e_{95} + 0 \times e_{111} + 114 \times e_{127} \\
&\qquad + \alpha \times e_0 + \beta \times e_{64} - \gamma \times e_{32} - \delta \times e_{96}
\end{aligned}$$

Class 7 ($y_i = y'_{i\oplus 14}$, $W_i = W'_{i\oplus 7}$). For this symmetry relation, the pairs of suitable messages are in an affine space of dimension 2:

$$\begin{aligned}
M_{\alpha,\beta} &= 32 \times e_{15} + 212 \times e_{31} + 157 \times e_{47} + 218 \times e_{63} + 129 \times e_{79} + 162 \times e_{95} + 174 \times e_{111} + 199 \times e_{127} \\
&\qquad + \alpha \times e_0 + \beta \times e_{64} \\
M'_{\alpha,\beta} &= 32 \times e_{15} + 212 \times e_{31} + 157 \times e_{47} + 218 \times e_{63} + 129 \times e_{79} + 162 \times e_{95} + 174 \times e_{111} + 199 \times e_{127} \\
&\qquad + \alpha \times e_0 - \beta \times e_{64}
\end{aligned}$$

Input and Output Pairs. Like in SIMD-256, we can show than a pair of output cannot be a valid input pair for the symmetry relation. We have:

$$\begin{aligned}
D_7^{\prime(35)} &= \left(\left(D_7^{\prime(31)} \boxplus IV'_7 \boxplus \mathsf{IF}(A_7^{\prime(31)}, B_7^{\prime(31)}, C_7^{\prime(31)}) \right)^{\lll 13} \boxplus A_2^{\prime(31)\lll 4} \right)^{\lll 13} \\
\overleftrightarrow{D}_7^{(35)} &= \left(\left(\overleftrightarrow{D}_7^{(31)} \boxplus \overleftrightarrow{IV}_7 \boxplus \mathsf{IF}(\overleftrightarrow{A}_7^{(31)}, \overleftrightarrow{B}_7^{(31)}, \overleftrightarrow{C}_7^{(31)}) \right)^{\lll 13} \boxplus \overleftrightarrow{A}_2^{(31)\lll 4} \right)^{\lll 13}
\end{aligned}$$

Since $\mathcal{S}^{(31)}$ and $\mathcal{S}'^{(31)}$ are symmetric, we have:

$$A_7^{\prime(31)} = \overleftrightarrow{A}_7^{31} \qquad B_7^{\prime(31)} = \overleftrightarrow{B}_7^{31} \qquad C_7^{\prime(31)} = \overleftrightarrow{C}_7^{31} \qquad D_7^{\prime(31)} = \overleftrightarrow{D}_7^{31} \qquad A_2^{\prime(31)} = \overleftrightarrow{A}_2^{31}$$

Again, the differences in $D_7^{(35)}$ and in IV_7 are not compatible:

Class 1 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 0$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 241 \times 2^{24}$

Class 2 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 74 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 16 \times 2^{24}$

Class 3 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 58 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 122 \times 2^{24}$

Class 4 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 219 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 127 \times 2^{24}$

Class 5 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 237 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 195 \times 2^{24}$

Class 6 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 35 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 114 \times 2^{24}$

Class 7 $IV'_7 \oplus \overleftrightarrow{IV}_7 = 212 \times 2^{24}$, $D_7^{\prime(35)} \oplus \overleftrightarrow{D}_7^{(35)} = 199 \times 2^{24}$

7.A.3 Symmetry Classes for the Final Transformation

The final transformation is based on a slightly modified compression function and similar symmetry classes can be found. However, we note that all the messages and message pairs than can give a symmetric expanded message have a non zero value in the last message byte ($M[63]$ for SIMD-256, or $M[127]$ for SIMD-512). Since the message input of the final compression function is in fact the length of the message being hashed, this means that the message length must be at least 2^{504} for SIMD-256, and at least 2^{1016} for SIMD-512. Hashing such a long message is completely meaningless[3], so it is safe to say that the distinguisher cannot be used against the final transformation.

7.B Proof of Theorem 7.1

In this section we show that the prefix-free iteration of an ideal compression function is indifferentiable from a random oracle, thus proving Theorem 7.1. The content of this section borrows very much to the proof in the extended version of [CDMP05].

We consider a *simulator* $\mathcal{S}$, which has oracle access to a random oracle $RO : \{0,1\}^* \to \{0,1\}^p$, and whose task is to simulate a random compression function. The pseudo-code of the simulator is shown in Algorithm 7.4 page 166, but here are a few comments. The simulator maintains a log of the queries it has answered to. This knowledge is maintained under the form of a graph $G = (V, E)$, where the set of vertices V is a subset of $\mathcal{H}$, and where the edges are labelled by message blocks from $\mathcal{M}$. The semantic of this graph is that there is an edge labelled by m between h and h' if the simulator lets the distinguisher know that $f(h,m) = h'$. We denote this by $h \xrightarrow{m} h'$. Initially, the graph contains only a single vertex IV. The simulator also maintains a subset of V denoted by Reach, consisting of the vertices that are reachable from IV. It also associates to each vertex v in Reach an *ancestor* in Reach. This allows to efficiently reconstruct the sequence of message blocks that maps IV to v, given $v \in$ Reach. We will note $IV \xrightarrow[*]{M} v$ when there is such a path between IV and v. At the beginning, Reach only contains the IV.

Let F be a random function. Now, a *distinguisher* $\mathcal{D}$ interacts with either H^F and F (we say that it is in the "construction world"), or with RO and $\mathcal{S}$ (and we say that it is in the "random oracle world"), and it has to tell in which world it is. More formally, $\mathcal{D}$ is a Turing machine with two interfaces. It should output "1" when H^F and F are answering its oracle queries, and "0" when RO and $\mathcal{S}$ are. Our objective is to show that for all distinguisher $\mathcal{D}$ the following holds for a small ϵ:

$$\left|\mathbb{P}\left[\mathcal{D}^{H^F,F} = 1\right] - \mathbb{P}\left[\mathcal{D}^{RO,\mathcal{S}} = 1\right]\right| \leq \epsilon$$

The proof uses a hybrid argument through a sequence of games. We will denote by q_S and q_O the number of queries sent to the Simulator and the Oracle respectively, by the distinguisher.

Game 1. The distinguisher is in the random oracle world. It has access to RO and $\mathcal{S}$. Let G_1 be the event that $\mathcal{D}$ outputs "1" in this setting:

$$\mathbb{P}\left[G_1\right] = \mathbb{P}\left[\mathcal{D}^{RO,\mathcal{S}} = 1\right]$$

Game 2. We introduce a dummy *relay algorithm* $\mathcal{T}$, which has oracle access to RO. Given a random oracle query from the distinguisher, $\mathcal{T}$ just sends the query to RO, and transmits the answer of RO back to $\mathcal{D}$. Let G_2 be the event that $\mathcal{D}$ outputs "1" in this case. Since the view of $\mathcal{D}$ is left unchanged, we have:

$$\mathbb{P}\left[G_2\right] = \mathbb{P}\left[\mathcal{D}^{\mathcal{T}^{RO},\mathcal{S}} = 1\right] = \mathbb{P}\left[G_1\right]$$

3. If it is feasible to hash these messages, then the hash function can be broken by brute force and does not offer any kind of security.

Algorithm 7.4 Pseudo-code of the Simulator $\mathcal{S}$

```
1: function FreshValue($h, m$)
2:     if $IV \xrightarrow[*]{M} h \in \mathsf{Reach}$ then
3:         if there exists $M'$ such that $M \| m = g(M')$ then
4:             $h' \leftarrow RO(M')$
5:         else
6:             $h' \overset{\$}{\leftarrow} \mathcal{H}$
7:         end if
8:         $\mathsf{Reach} \leftarrow \mathsf{Reach} \cup \left\{h \xrightarrow{m} h'\right\}$
9:     else
10:        $h' \overset{\$}{\leftarrow} \mathcal{H}$
11:    end if
12:    $V \leftarrow V \cup \{h, h'\}$
13:    $E \leftarrow E \cup \left\{h \xrightarrow{m} h'\right\}$
14:    return $h'$
15: end function

16: function Simulator($h, m$)
17:     if there exists a vertex $h' \in V$ and an edge $h \xrightarrow{m} h'$ in $E$ then
18:         return this $h'$
19:     else
20:         $h' \leftarrow$ FreshValue($h, m$)
21:         return $h'$
22:     end if
23: end function
```

Game 3. In this game, we modify the simulator $\mathcal{S}$. In particular, we restrict the responses of the simulator such that they never satisfy certain specific failure conditions. If the simulator comes up with a response that would result in an inconsistent state, then it fails explicitly instead of sending that response. The failure conditions describe certain situations that could be exploited by the distinguisher, such as collisions on the internal state. We just slightly change the FreshValue function:

```
1: function FreshValue($h, m$)
2:     if $IV \xrightarrow[*]{M} h \in \mathsf{Reach}$ then
3:         if there exists $M'$ such that $M \| m = g(M')$ then
4:             $h' \leftarrow RO(M')$
5:         else
6:             $h' \overset{\$}{\leftarrow} \mathcal{H}$
7:         end if
8:         if $h' \in V$ then
9:             Abort
10:        end if
11:        $\mathsf{Reach} \leftarrow \mathsf{Reach} \cup \left\{h \xrightarrow{m} h'\right\}$
12:    else
13:        $h' \overset{\$}{\leftarrow} \mathcal{H}$
14:    end if
```

```
15:     $V \leftarrow V \cup \{h, h'\}$
16:     $E \leftarrow E \cup \left\{h \xrightarrow{m} h'\right\}$
17:     **return** $h'$
18: **end function**
```

It should be clear that until no abort occur, the subgraph Reach is in fact a tree rooted in IV. This follows from the fact that the simulator aborts as soon as a collision in the internal state is detected. The new value h' is always drawn uniformly at random. It should be clear that as long as the simulator does not abort, the number of nodes in V is upper-bounded by $2q_S + 1$.

Therefore, for a given query, the probability of failure is upper-bounded by $(2q_S + 1)/2^p$. For all the q_S queries sent by the distinguisher, the probability of failure is therefore less than $q_S \cdot (2 \cdot q_s + 1)/2^p$. Let G_3 be the event that $\mathcal{D}$ outputs "1" in this case. Since the view of $\mathcal{D}$ only changes when the simulator aborts, we have:

$$\left|\mathbb{P}[G_3] - \mathbb{P}[G_2]\right| \leq 2 \cdot \frac{(q_S+1)^2}{2^p}$$

Game 4. In this game, we modify the relay algorithm and leave the simulator unchanged. The underlying idea is to make the responses of the relay algorithm directly dependent on the simulator. Thus, instead of giving the new relay algorithm $\mathcal{T}_1$ an oracle access to the random oracle RO, it is now given oracle access to the simulator $\mathcal{S}_0$. On a random oracle query X, the relay algorithm $\mathcal{T}_1$ computes the prefix-free encoding of X, $g(X)$. It then applies the Merkle-Damgård construction to $g(X)$ and queries the simulator $\mathcal{S}_0$ to evaluate the compression function. Thus the relay algorithm $\mathcal{T}_1$ is essentially the same as the random oracle construction pf-MD, except that it is based on the simulator $\mathcal{S}_0$ instead of random function F. Let G_4 denote the event that the distinguisher $\mathcal{D}$ outputs "1" when given oracle access to $\mathcal{T}_1$ and $\mathcal{S}_0$ in this game. Thus, we know that

$$\mathbb{P}[G_4] = \mathbb{P}[\mathcal{D}^{\mathcal{T}_1,\mathcal{S}_0} = 1]$$

Before going further, we establish two key properties of $\mathcal{S}_0$. Let us consider the sequence $\mathcal{Q}$ of queries (h_i, m_i, h'_i) sent to $\mathcal{S}_0$, where h'_i is the answer and (h_i, m_i) is the question. We say that the IV is *reachable*, and at a given point in the simulation h'_i is reachable if there has been a previous query (h_i, m_i, h'_i) where h_i was reachable. Then:

i) Until $\mathcal{S}_0$ fails, Reach precisely describes the set of reachable chaining values.

ii) Until $\mathcal{S}_0$ fails, Reach describes a tree.

These two properties are easy to establish by induction on the number of queries. When the simulator detects that h_i is reachable, it puts its answer h'_i in Reach. What guarantees that our two properties hold is that $\mathcal{S}_0$ aborts if h'_i was already "known". Thus, the set of reachable values can only be extended by one element, namely h'_i, and Reach is updated accordingly.

Next, we claim that the following three statements hold:

i) In Game 3, *i.e.* when $\mathcal{D}$ interacts with $\left(\mathcal{T}^{RO}, \mathcal{S}_0\right)$, the answers of $\mathcal{S}_0$ are consistent with those of RO as long as $\mathcal{S}_0$ does not abort.

ii) In Game 4, *i.e.* when $\mathcal{D}$ interacts with $\left(\mathcal{T}_1^{pf-MD(\mathcal{S}_0)}, \mathcal{S}_0\right)$, the answers of $\mathcal{S}_0$ are consistent with those of RO as long as $\mathcal{S}_0$ does not abort.

iii) $\mathcal{T}^{RO}$ and $\mathcal{T}_1^{pf-MD(\mathcal{S}_0)}$ give the same answers until the simulator aborts.

From these three points, we can deduce that the view of the distinguisher $\mathcal{D}$ remains unchanged from game 3 to game 4 if the simulator $\mathcal{S}_0$ does not fail in either of the two games.

Proof. *i)* To detect an inconsistency between $\mathcal{S}_0$ and RO, one has to build a chain of queries corresponding to a valid message, and compare with the output of RO with the last query of the chain. Note that if the chain is built out-of-order, then the simulator will abort. Therefore the last query to be sent to $\mathcal{S}_0$ is the final block of the prefix free encoding of M. When $\mathcal{S}_0$ detects the final block of a message, it queries RO on the decoded message, which is unique because Reach is a tree. The answers of RO and $\mathcal{S}_0$ are thus consistent.

ii) The justification is the same as in the previous point. The fact that $\mathcal{T}_1$ sends extra queries does not change the fact that $\mathcal{S}_0$ answers are consistent with the Random Oracle.

iii) Since $\mathcal{S}_0$ is consistent with the VIL-RO, the relay algorithm $\mathcal{T}_1$ does in fact return $RO(M)$ by applying the pf-MD construction with $\mathcal{S}_0$. □

We can finally complete the argument by observing that if the maximum length of the prefix-free encoding of a random oracle query made by $\mathcal{D}$ is κ blocks, then,

$$\begin{aligned} \left|\mathbb{P}\big[G_4\big] - \mathbb{P}\big[G_3\big]\right| &\leq \mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 3}\big] + \mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 4}\big] \\ &\leq 2 \cdot \frac{(q_S+1)^2 + (q_S + \kappa \cdot q_O + 1)^2}{2^p} \end{aligned}$$

Game 5. In this game, we modify the simulator $\mathcal{S}_0$ so as to make its responses independent of the random oracle RO. For this purpose, we remove the random oracle RO from this game entirely and the new simulator $\mathcal{S}_1$ always chooses a random p-bit response itself, even in situations where $\mathcal{S}_0$ would have consulted RO. We also remove all the failure conditions from the new simulator $\mathcal{S}_1$. More precisely, we change the simulator in the following way:

```
1: function FreshValue($h, m$)
2:     $h' \xleftarrow{\$} \mathcal{H}$
3:     $V \leftarrow V \cup \{h, h'\}$
4:     $E \leftarrow E \cup \left\{h \xrightarrow{m} h'\right\}$
5:     return $h'$
6: end function
```

The responses of these two simulators are identical apart from the failure conditions which are used by $\mathcal{S}_0$ and not by $\mathcal{S}_1$: even when $\mathcal{S}_0$ consults the random oracle, its response is still uniformly distributed. Thus, the distinguisher does not notice a difference between these games if in game 4, the simulator $\mathcal{S}_0$ does not fail.

Let G_5 denote the event that the distinguisher $\mathcal{D}$ outputs "1" in game 5, so that

$$\mathbb{P}\big[G_5\big] = \mathbb{P}\big[\mathcal{D}^{\mathcal{T}_1,\mathcal{S}_1} = 1\big]$$

Then we can deduce that:

$$\begin{aligned} \left|\mathbb{P}\big[G_5\big] - \mathbb{P}\big[G_4\big]\right| &\leq \mathbb{P}\big[\mathcal{S}_0 \text{ fails in game 4}\big] \\ &\leq 2 \cdot \frac{(q_S + \kappa \cdot q_O + 1)^2}{2^p} \end{aligned}$$

Game 6. This is the final game of our argument. Here we finally replace the simulator $\mathcal{S}_1$ with the random function F. Since the relay algorithm $\mathcal{T}_1$ simply implemented the prefix-free Merkle-Damgård construction, then the view of the distinguisher is in fact the construction world.

Now, by combining games 1 to 6, we can show that

$$\left|\mathbb{P}\big[\mathcal{D}^{H^F,F} = 1\big] - \mathbb{P}\big[\mathcal{D}^{RO,\mathcal{S}} = 1\big]\right| \leq 4 \cdot \frac{(q_S+1)^2 + (q_S + \kappa \cdot q_O + 1)^2}{2^p}$$

7.C Proof of Theorem 7.2

The proof proceed in the same way as the proof of Theorem 7.1, given in Appendix 7.B. The simulator is shown in Algorithm 7.1 page 151. The pseudo-code shows $\mathcal{S}_0$ with the failure conditions. Before going further, a few comments on $\mathcal{S}_0$ are in order. When it receives a query (h, m), it checks whether there exist a *symmetric query* $\left(\overline{h}, \overline{m}\right)$, that would trigger the symmetry relation (*i.e.* it checks whether the node (h, m) is connected to something in $G'_{\mathcal{R}_2}$). If such a query exists, then both are treated simultaneously in a "symmetric" way. In particular, if either one of these concerns a reachable state, then it is treated specially, even if it is not the original query, but the "symmetric" one.

Game 3. Let us discuss the probability that $\mathcal{S}_0$ fails. It can only happen if h is reachable, which in turn means that h' is randomly distributed in $\mathcal{H}$. $\mathcal{S}_0$ aborts when $h' \in \mathcal{W}$, $h' \in V$ or $\mathsf{Reach} \cup \{h'\}$ covers an edge of $G_{\mathcal{R}_2}$. The probability that $h' \in \mathcal{W}$ is $|\mathcal{W}|/2^p$, and the probability that $h' \in V$ is upper-bounded by $(4 \cdot q_S + 1)/2^p$, since $|V| \leq 4 \cdot q_s + 1$. Let us now discuss the probability that an edge of $G_{\mathcal{R}_2}$ is covered by Reach.

A simple induction on the number of queries shows that the chaining values in Reach are all randomly and independently distributed in $\mathcal{H}$ (this is because Reach is always extended by h' on line 19, and h' is itself always generated randomly). If we ignore the abort conditions, Reach is a random subset of $\mathcal{H}$ of size $k \leq q_s + 1$ after q_S queries. There are $\binom{2^p}{k}$ such subsets, and amongst these $\binom{2^p}{k-2}$ cover a given edge. The probability that at least one edge out of $|\mathcal{WP}|$ is covered is thus upper-bounded by $|\mathcal{WP}| \cdot \binom{2^p}{k-2} / \binom{2^p}{k}$, which it itself upper-bounded by $|\mathcal{WP}| \cdot k^2/(2^p - k)^2$. After q_s queries, the probability of failure is therefore bounded by:

$$\left|\mathbb{P}\left[G_3\right] - \mathbb{P}\left[G_2\right]\right| \leq 4 \cdot \frac{\left(q_S+1\right)^2}{2^p} + |\mathcal{W}| \cdot \frac{q_S+1}{2^p} + |\mathcal{WP}| \cdot \frac{q_S+1^2}{\left(2^p - q_S - 1\right)^2}$$

Game 4. We claim that the following four statements hold:

i) In Game 3, *i.e.* when $\mathcal{D}$ interacts with $\left(\mathcal{T}^{RO}, \mathcal{S}_0\right)$, the answers of $\mathcal{S}_0$ are consistent with those of RO as long as $\mathcal{S}_0$ does not abort.

ii) In Game 4, *i.e.* when $\mathcal{D}$ interacts with $\left(\mathcal{T}_1^{pf-MD(\mathcal{S}_0)}, \mathcal{S}_0\right)$, the answers of $\mathcal{S}_0$ are consistent with those of RO as long as $\mathcal{S}_0$ does not abort.

iii) $\mathcal{T}^{RO}$ and $\mathcal{T}_1^{pf-MD(\mathcal{S}_0)}$ give the same answers until the simulator aborts.

iv) As long as it does not abort, the answer of $\mathcal{S}_0$ always comply with the relation $\mathcal{R}$.

Consistency with the VIL-RO. Establishing the first three points can be done in the same as it was done in the proof of Theorem 7.1. The simulator relies on the fact that Reach is a tree, and that it exactly describes the reachable chaining values in V. This can be established by arguing that if $\mathcal{S}$ does not abort, then h' is the only new reachable chaining value created by the current invocation of FreshValue. Note that $\overline{h}$, if it exists, is not reachable.

Conformance to the Relation. The main point is that the relation can never be falsified on reachable states, and that the samplers are used on non-reachable states to ensure that the answers are consistent with the relation. More precisely, the simulator aborts as soon as a state in $\mathcal{W}$ becomes reachable, or a pair of states in $\mathcal{WP}$ becomes reachable.

Let us assume that the distinguisher can find a query (h, m, h') with $h \xrightarrow{m} h'$ such that $\mathcal{R}_1(h, m, h')$ does not hold. Then we have $h \in \mathcal{W}$ by definition of $\mathcal{W}$, therefore h cannot be reachable and h' has necessarily been build by $\mathbf{Sampler}_1$. By definition of $\mathbf{Sampler}_1$, $\mathcal{R}_1(h, m, h')$ must hold.

Similarly, let us assume that the distinguisher finds $h_1 \xrightarrow{m_1} h'_1$ and $h_2 \xrightarrow{m_2} h'_2$ such that $\mathcal{R}_2(h_1, m_1, h_2, m_2, h'_1, h'_2)$ does not hold. By definition of $\mathcal{WP}$ we have $(h_1, h_2) \in \mathcal{WP}$ therefore h_1 and h_2 cannot both be reachable. Moreover, we have $(h_1, m_1) \leftrightarrow (h_2, m_2) \in G'_{\mathcal{R}_2}$. Without loos of generality, we assume that h_2 is not reachable. When the first query involving (h_1, m_1) or (h_2, m_2) was sent to $\mathcal{S}_0$, the simulator built the second query. If h_1 was reachable, h'_1 has been built by calling the VIL-RO and h'_2 has been built by $\mathbf{Sampler}_2$, with assures that $\mathcal{R}_2(h_1, m_1, h_2, m_2, h'_1, h'_2)$ holds. Similarly, if h_1 is not reachable, h'_1 has been built by $\mathbf{Sampler}_1$, and h'_2 by $\mathbf{Sampler}_2$. We note that if h_1 was not reachable at the time when it was queried, it cannot become reachable later without causing the simulator to abort.

Finally, we obtain that the view of the distinguisher does not change as long as the simulator does not abort:

$$\begin{aligned}
\big|\mathbb{P}\big[G_4\big] - \mathbb{P}\big[G_3\big]\big| &\leq \mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 3}\big] + \mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 4}\big] \\
&\leq 2 \cdot \mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 4}\big] \\
&\leq 8 \cdot \frac{\left(q_S + \kappa \cdot q_O\right)^2}{2^p} + 2|\mathcal{W}| \cdot \frac{q_S + \kappa \cdot q_O}{2^p} + 2|\mathcal{WP}| \cdot \frac{\left(q_S + \kappa \cdot q_O\right)^2}{\left(2^p - q_S - \kappa \cdot q_O\right)^2}
\end{aligned}$$

And we conclude:

$$\begin{aligned}
\Big|\mathbb{P}\big[\mathcal{D}^{H^F, F} = 1\big] - \mathbb{P}\big[\mathcal{D}^{RO, \mathcal{S}} = 1\big]\Big| &\leq \big|\mathbb{P}\big[G_1\big] - \mathbb{P}\big[G_6\big]\big| \\
&\leq 4\mathbb{P}\big[\mathcal{S}_0 \text{ fails in Game 4}\big] \\
&\leq 16 \cdot \frac{\left(q_S + \kappa \cdot q_O\right)^2}{2^p} + 4 \cdot |\mathcal{W}| \cdot \frac{q_S + \kappa \cdot q_O}{2^p} \\
&\quad + 4 \cdot |\mathcal{WP}| \cdot \frac{\left(q_S + \kappa \cdot q_O\right)^2}{\left(2^p - q_S - \kappa \cdot q_O\right)^2}
\end{aligned}$$

Part Three

New Cryptanalytic Techniques

The half-life of new stream ciphers is one year. The half-life of new hash functions is 6 months.
Matt Robshaw

Abstract

With the launch of the SHA-3 competition, many new hash function designs have been proposed, and most of them differ significantly from the design principles of the MD/SHA family. New dedicated techniques are needed to study these functions. For instance, the rebound attack [MRST09] has been quite successful against AES-based designs, such as Echo, Grøstl or Lane. Cryptanalysis of block ciphers is mostly based on statistical properties, such as differential and linear cryptanalysis, but in the case of hash functions, other approaches are possible because of the high number of degrees of freedom. In this part I present new cryptanalytic techniques that I have developed to attack several SHA-3 candidates.

The first technique is based on generalisations of the complementation property of DES. We present a collection of attacks based on symmetry relations in the key schedule and in the actual rounds. We use these symmetries to build distinguishers for any number of rounds when the relation is deterministic. We further explore these properties, and show that if the relations have easily found fixed points, a new kind of attacks can be applied.

Our main result is a self-similarity property on the SHA-3 candidate *Lesamnta*. We show a very efficient distinguisher on the full compression function. We also show how to use this self-similarity property to find collisions on the full compression function of *Lesamnta* much faster than generic attacks, and to mount a herding attack on the full hash function. We also describe a new related-key differential attack on round-reduced versions of the XTEA block-cipher, using a similar technique.

The second technique applies to hash functions which are based on generalized Feistel schemes. We describe a new kind of attack based on a cancellation property in the round function. This new technique allows to use efficiently the degrees of freedom available to attack a hash function. Using the cancellation property, we can avoid the non-linear parts of the round function, at the expense of some degrees of freedom.

Our attacks are mostly independent of the round function in use. We describe a 22-round generic attack on the structure of *Lesamnta*, and adapt it to the actual round function to attack 24-round *Lesamnta* (the full function has 32 rounds). We also apply the technique to *SHAvite-3*$_{512}$, and we show a generic attack against 10-round *SHAvite-3*$_{512}$. Moreover, we expand this attack to a pseudo-attack on the full 14-round *SHAvite-3*$_{512}$ using extra degrees of freedom from the salt.

Finally, we use similar techniques to study block ciphers based on generalized Feistel schemes. We show an integral attack on the inner block cipher of *Lesamnta*, and a new truncated differential path for *SMS4*.

Contents

Chapter **8**

Self-similarity Attacks

8.1 Introduction

In this chapter we study the existence of simple relations that can go through the rounds of a cipher with a very high probability. For example, in DES if all the key bits are flipped, then all the subkeys are flipped as well. Moreover, if we also negate the plaintext, then all the F functions receive the original input, and the ciphertext is also negated: $DES_{\overline{K}}(\overline{P}) = \overline{(DES_K(P))}$. This is known as the complementation property of DES, described by Figure 8.1.

Several other constructions have similar simple relations that can be exploited:

– LOKI has a complementation property [KP91] for several values of α:

$$E_{K\oplus\alpha}(P \oplus (\alpha,\alpha)) = E_K(P) \oplus (\alpha,\alpha)$$

– TEA has equivalent keys [KSW96], *i.e.* for several values of Δ we have:

$$E_{K\oplus\Delta}(P) = E_K(P)$$

– There is a simple pseudo-collision attack [ABMM09] on the SHA-3 candidate CHI:

$$F(\overline{X}, \overline{M}) = F(X, M)$$

– Den Boer and Bosselaers [dBB93] demonstrated a simple pseudo-collision attack on MD5:

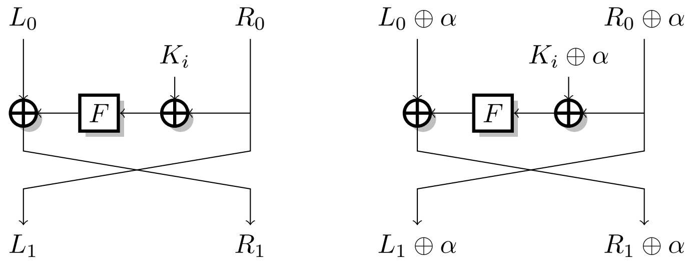
$$F(X, M \oplus \Delta_{msb}) = F(X, M) \quad \text{with probability } 2^{-48}$$

– The round function of the AES has strong symmetry properties [LSWD04], which are only avoided by the key schedule.

In this work, we try to give a generalization of these properties as self-similarities and we show new kinds of self-similarity properties in block ciphers and hash functions. The new ideas generalize the previous attacks, by treating a wider set of relations. Moreover, some of the similarity relations we use have fixed points, which is not the case for the complementation property. The keys which are mapped to themselves by the similarity relation can be treated as weak keys, and they actually allow to mount various attacks when the cipher is used to build a hash function.

Self similarity properties usually need a related-key setting, or can be used with a class of weak keys. However in the case of hash functions, the key is often under the control of the adversary, and self-similarity properties can be used more easily.

Figure 8.1 Complementation property of DES and LOKI.
If one can find K, K' such that $\forall i, K'_i = K_i \oplus \alpha$, then $E_{K'}(P \oplus (\alpha, \alpha)) = E_K(P) \oplus (\alpha, \alpha)$



Deterministic self-similarity properties can usually be detected with a very small number of queries (one or two), over *any* number of rounds of a cipher, making them very interesting properties to study. Our distinguishers are very simple and efficient, so that they can be practical if the primitive is used in an unusual setting. For instance, the equivalent keys of TEA do not seem to be a practical threat for the block cipher, up to a loss of two bits of security in exhaustive search. However, Microsoft used TEA as a hash function in Davies-Meyer mode to enforce limitations on the Xbox, and this weakness of TEA has been used *in practice* to bypass the security limitations [Ste05].

8.1.1 Our Results

We describe new self-similarity relations in several hash functions and block ciphers, and new attacks based on these relations. Our main result is a self-similarity relation in the hash function *Lesamnta* which gives a distinguisher with a single query to the compression function. Using this relation, we show a herding attack on the full hash function which is more efficient than the generic herding attack. We also show a probabilistic self-similarity in the hash function *ESSENCE*, which is similar to the rotational cryptanalysis of Khovratovich and Nikolić [KN10]. Concerning block ciphers, we describe a new iterative related-key differential for XTEA, which gives the best known related key attack on XTEA. We also identify a class of weak keys in the block cipher $\mathcal{PURE}$, showing that self-similarity relations can be found using algebraic relations.

One of the interesting outcomes of this research, is a new way to tackle round constants. While differing round constants seem to thwart slide attacks and symmetry properties, they are not necessarily sufficient to protect against our more generalized approach, as we present in the attack on the compression function of *Lesamnta*. In particular, this cast a doubt on the effectiveness of simple round constants, *e.g.* derived from a round counter. This can be related to recent attacks on Grøstl exploiting the sparse constants [Pey10].

8.1.2 Road-map

First we formally define the notion of self-similarity in Section 8.2, and we show several ways to exploit such properties in a generic way. In particular we discuss new attacks based on keys which are fixed under the similarity relation. Then we show concrete examples of self-similarity relation: we study *Lesamnta* in Section 8.3, *ESSENCE* in Section 8.4, XTEA in Section 8.5 and $\mathcal{PURE}$ in Section 8.6. We conclude the chapter in Section 8.7.

8.2 Self-Similarity

Following [BB02], we define self-similarity as follows:

Definition 1 (Self-similarity relation in a block cipher). A block cipher E encrypts the plaintext P under the key K to a ciphertext $C = E_K(P)$. A self-similarity relation is given by invertible and easy to compute transformations [1] ϕ, ψ and θ such that:

$$\forall K, P: \quad \theta(E_K(P)) = E_{\psi(K)}(\phi(P))$$

If such relations exist then the cipher is self-dual according to the terminology of [BB02]. Similarly, we can define self-similarity for compression functions and for stream ciphers:

Definition 2 (Self-similarity relation in a compression function). A compression function F maps a chaining value X and a message M to a new chaining value $H = F(X, M)$. A self-similarity relation is given by invertible and easy to compute transformations ϕ, ψ and θ such that:

$$\forall X, M: \quad \theta(F(X, M)) = F(\phi(X), \psi(M))$$

Definition 3 (Self-similarity relation in a stream cipher). A stream cipher G generates the key-stream $G_K(I)$ with the key K and the IV I. A self-similarity relation is given by invertible and easy to compute transformations ϕ, ψ and θ such that:

$$\forall K, I: \quad \theta(G_K(I)) = G_{\psi(K)}(\phi(I))$$

These definitions are wide enough to include several types of known attacks such as:

– The complementation property of DES:

$$\psi(K) = \overline{K}, \qquad \phi(P) = \overline{P}, \qquad \theta(C) = \overline{C}$$

– The complementation property of LOKI [KP91]:

$$\psi(K) = K \oplus \alpha, \qquad \phi(P) = P \oplus \alpha, \qquad \theta(C) = C \oplus \alpha$$

for several values of α.

– The equivalent keys of TEA [KSW96]

$$\psi(K) = K \oplus \Delta, \qquad \phi(P) = P, \qquad \theta(C) = C$$

for several values of Δ.

– The pseudo-collision attack in the compression function of the SHA-3 candidate CHI [ABMM09]:

$$\psi(X) = \overline{X}, \qquad \phi(M) = \overline{M}, \qquad \theta(H) = H$$

Moreover, we say that a set of transformation is a *weak self-similarity* if the relation only holds with a probability which is less than 1 (but higher than for a random permutation). Notable examples of weak self-similarity include:

– The pseudo-collision attack of den Boer and Bosselaers on MD5 [dBB93]:

$$\psi(X) = X \oplus \Delta_{msb} \qquad \phi(M) = M \qquad \theta(H) = H$$

1. In this context, and for the reminder of the chapter, we require that at least one of the three transformations ϕ, ψ, and θ is not the identity.

– The rotational cryptanalysis of Khovratovich and Nikolić [KN10]:

$$\psi(K) = K^{\lll 1} \qquad \phi(P) = P^{\lll 1} \qquad \theta(C) = C^{\lll 1}$$

Note that our self-similarity attack on *ESSENCE* in Section 8.4 is an independent use of rotational cryptanalysis.

It is also possible to consider high probability differentials as (weak) self-similarity properties. For example, in Section 8.5 we present some iterative related-key differential for XTEA that was found using the self-similarity approach.

We note that the definition of weak self-similarity is in fact quite wide and include many different kind of attacks such as related-key attacks [Bih93, Bih94, Knu92] and slide attacks [BW99]. In this work, we restrict ourselves to iterated self-similarity properties, with simple key-independent relations. As opposed to related-key slide attacks, we look for relations θ such that the subkey RK_i of round i is related to the subkey RK'_i, not to RK'_{i-1}.

8.2.1 Generic Attacks Based on Self-Similarity

Self-similarity properties obviously offer an efficient distinguisher in the related-key setting. If one is given access to an oracle E_{K^*} and an oracle $E_{\psi(K^*)}$ for an unknown K^*, he can distinguish the block cipher E from an ideal cipher by querying $E^*_K(P)$ and $E_{\psi(K^*)}(\phi(P))$ for a random P.

Moreover, a self-similarity property can be used to speed up the exhaustive search of the key by a small factor, as explained in [Bih93]. Let k be the size of the longest cycle of the permutation ψ. In most cases, k will be quite small, *e.g.* $k = 2$ for a complementation property. Then, query $C_i = E_{K^*}(\phi^{(i)}(P))$ for $i \in 0, 1, \ldots, k-1$, and compute $\widehat{C_i} = \theta^{(-i)}(C_i)$. Now, compute $E_K(P)$ for a set of keys $\mathcal{K}$ and look for a match with one of the $\widehat{C_i}$. If there is a match, then $\psi^{(i)}(K)$ is likely to be the key:

$$\begin{aligned} E_K(P) = \widehat{C_i} &\iff E_K(P) = \theta^{(-i)}(C_i) \\ &\iff E_{\psi^{(i)}(K)}(\phi^{(i)}(P)) = C_i = E_{K^*}(\phi^{(i)}(P)). \end{aligned}$$

The idea of the attack is that the set $\mathcal{K}$ has to include only one key per each cycle of ψ. Each time a new $E_K(P)$ is computed, this allows to test all the key candidates in the cycle of K, by applying θ iteratively to the obtained ciphertext, and comparing the resulting ciphertext with the respective pre-computed ciphertext. Hence, if the evaluation of θ is faster than $E_{\psi(K)}$, one can reduce the time of exhaustive search.

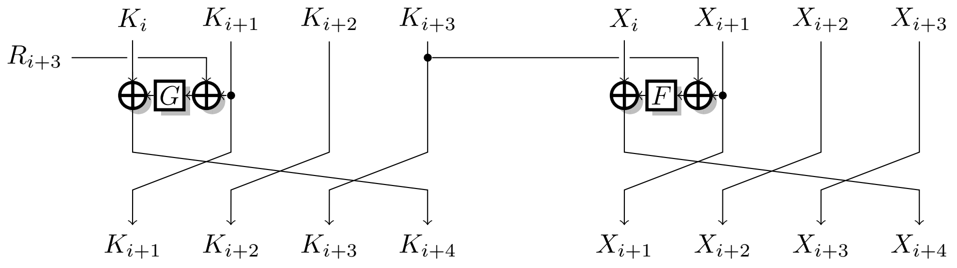
8.2.2 Generic Attacks Based on Fixed Points of Self-Similarity Relations

For some of our relations, ϕ and ψ have a large number of fixed points. These points can be used in several different attack scenarios. The fixed points of ψ will be weak keys, so we consider the fixed points of ϕ as weak plaintexts.

First, let us show that the set of the fixed points of ψ is a weak-key class. If a weak plaintext is encrypted under a weak key, the ciphertext will also be a fixed point of the similarly relation, since $\theta(C) = \theta(E_K(P)) = E_{\psi(K)}(\phi(P)) = E_K(P) = C$. This allows to distinguish the weak keys with a single query using a weak plaintext.

In the context of hash functions, fixed points in the similarity relations allow to find collisions in the compression function more efficiently than exhaustive search. One just has to evaluate the compression function on weak inputs, and the output has to lie in the set of fixed point of θ. Note that the attack becomes more efficient when the number of fixed-points in θ becomes smaller. A similar property has been used to find semi-free-start collisions in CubeHash [ABM+09, FLM10].

Figure 8.2 The Round Function of *Lesamnta*. This Round Function is Iterated 32 Times.



In Section 8.3.3 we show how to use the self-similarity property of *Lesamnta* to reduce its security in several usage scenarios. These attacks are based on the idea that one can randomly reach a fixed point of ϕ, and from there it is easy to reach any fixed point of θ. In this case, the attacks become more efficient when the number of fixed points of ϕ grows.

8.3 Application to *Lesamnta*

Our most interesting result is a self-similarity property of *Lesamnta*. This self-similarity property is based on swapping the two halves of the state and XOR-ing them with a constant. The most surprising part about this property is that it can actually deal with the round constants, which are supposed to break symmetry relations.

8.3.1 A Short Description of *Lesamnta*

Lesamnta is a hash function proposal by Hirose, Kuwakado, and Yoshida as a candidate in the SHA-3 competition [HKY08]. It is based on a 32-round unbalanced Feistel scheme with four registers used in MMO mode. The key schedule is also based on a similar Feistel scheme. The round function is described by Figure 8.2 and can be written as:

$$\begin{aligned} X_{i+4} &= X_i \oplus F\left(X_{i+1} \oplus K_{i+3}\right) \\ K_{i+4} &= K_i \oplus G\left(K_{i+1} \oplus R_{i+3}\right) \end{aligned}$$

where $R_0, \ldots, R_{31}$ are round constants, the state register X is initialized with the message in X_{-3}, X_{-2}, X_{-1}, X_0, and the key register is initialized with the chaining value in K_{-3}, K_{-2}, K_{-1}, K_0. The output of the compression function is $X_{-3} \oplus X_{29}, X_{-2} \oplus X_{30}, X_{-1} \oplus X_{31}, X_0 \oplus X_{32}$.

Lesamnta has two main variants: *Lesamnta*-256 with 64-bit registers (hence, a message block and chaining value of 256 bits), and *Lesamnta*-512 with 128-bit registers (hence, a message block and chaining value of 512 bits).

The round functions F and G are inspired by the AES round function. F uses four rounds of transformations similar to SubBytes, ShiftRows and MixColumns, while G uses only one round of similar transforms. The transformations used in F and G are different, even though they are both heavily inspired by the AES. In *Lesamnta*-256 a 64-bit register is represented by a 2-by-4 byte matrix in F, and by a 4-by-2 matrix in G. In *Lesamnta*-512, a 128-bit register is seen as a 4-by-4 matrix. The round constants are defined by a simple counter: $R_i = (2i, 2i+1)$. For more details, we refer the reader to the full specification [HKY08].

8.3.2 The Self-Similarity Relation of *Lesamnta*

The round functions F and G are very similar to the AES round function, and they have the same symmetry property: if the two halves of the input are swapped, then the output is also swapped. Indeed, it is easy to see that SubBytes, ShiftRows and MixColumns do have this property.

Let us denote the swap of the halves of a state by $\overleftrightarrow{\bullet}$, *i.e.*, $\overleftrightarrow{(a,b)} = (b,a)$. More precisely, for the state of *Lesamnta*-256, we use:

$$\overleftrightarrow{\begin{array}{|c|c|c|c|}\hline s & t & u & v \\ \hline w & x & y & z \\ \hline\end{array}} = \begin{array}{|c|c|c|c|}\hline u & v & s & t \\ \hline y & z & w & x \\ \hline\end{array}$$

Then we have:

$$F(\overleftrightarrow{x}) = \overleftrightarrow{F(x)} \tag{8.1}$$

$$G(\overleftrightarrow{x}) = \overleftrightarrow{G(x)} \tag{8.2}$$

An obvious way to use this property would be to set $\psi(K) = \overleftrightarrow{K}$, but this fails because of the round constants. In fact the round constants are exactly supposed to prevent such a property.

However, we note that the round constants are quite structured. Since they are defined as $R_i = (2i, 2i+1)$, we have $R_i \oplus \overleftrightarrow{R_i} = (1,1)$. In order to study the effect of the round constants, we introduce a new symmetry relation $\widetilde{\bullet}$, defined as $\widetilde{x} = \overleftrightarrow{x} \oplus (1,1)$. More precisely, for *Lesamnta*-256, we use:

$$\widetilde{\begin{array}{|c|c|c|c|}\hline s & t & u & v \\ \hline w & x & y & z \\ \hline\end{array}} = \begin{array}{|c|c|c|c|}\hline u & v & s & t \\ \hline y & z & w & x \\ \hline\end{array} \oplus \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline\end{array}$$

Then we have $\widetilde{R_i} = R_i$.

Our idea is to combine the swapping of halves of the state with flipping the least significant bit of each half to compensate the difference in the constants. The swapping commutes with the round functions F and G, and the masking is canceled by the Feistel structure. This can be written as:

$$\widetilde{x} \oplus \widetilde{y} = \overleftrightarrow{x \oplus y} \tag{8.3}$$

$$\widetilde{x} \oplus \overleftrightarrow{y} = \widetilde{x \oplus y} \tag{8.4}$$

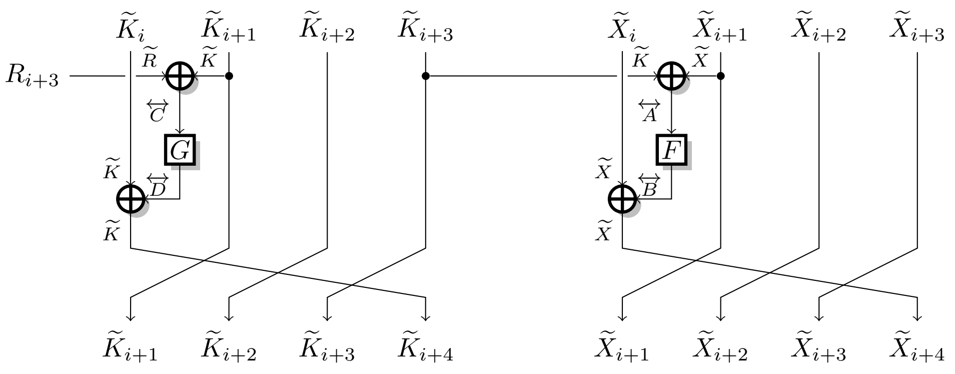
Now let us consider the relation $\psi(K) = \widetilde{K}$. If we have a pair of key K, K' with $K' = \widetilde{K}$, we can show that all the subkeys will be such that $K'_i = \widetilde{K_i}$. The input to the first G function is $K_{-2} \oplus R_0$. According to Property (8.3), we have $K'_{-2} \oplus R_0 = \widetilde{K}_{-2} \oplus \widetilde{R}_0 = \overleftrightarrow{K_{-2} \oplus R_0}$. Since the inputs of G are symmetric, the outputs will be symmetric according to Property (8.2). Finally, using Property (8.4), we can see that the new subkey $K_1 = K_{-3} \oplus G(K_{-2} \oplus R_0)$ will also be symmetric: $K'_1 = K'_{-3} \oplus G(K'_{-2} \oplus R_0) = \widetilde{K}_{-3} \oplus \overleftrightarrow{G(K_{-2} \oplus R_0)} = \widetilde{K}'_1$.

Iteratively, we can show that all the subkeys satisfy $K'_i = \widetilde{K}_i$. Similarly, if we use a pair of plaintexts X, X' with $X' = \widetilde{X}$, then all the registers will satisfy $X'_i = \widetilde{X}_i$. This can be seen in Figure 8.3.

Lastly, there is a feed-forward operation: the output of the compression function is $Y_0 = X_{-3} \oplus X_{29}, \ldots, Y_3 = X_0 \oplus X_{32}$. Again, thanks to the Property (8.3), we find that $Y' = \overleftrightarrow{Y}$. This yields:

$$F(\widetilde{K}, \widetilde{M}) = \overleftrightarrow{F(K,M)}$$

Figure 8.3 The Self-similarity Relation of *Lesamnta*



A Distinguisher on the Compression Function of *Lesamnta*. Finally, we note that if we pick a chaining value x which is weak (*i.e.* $x = \widetilde{x}$), and a message block m which is also weak (*i.e.* $m = \widetilde{m}$), then $F(x, m)$ is weak as well, but in a different manner (we have $F(x, m) = \overleftrightarrow{F(x, m)}$), and this can be easily identified (the top and bottom halves of each output word are the same). Hence, it is possible to distinguish the compression function of *Lesamnta* using one single query. For example, in *Lesamnta*-256

$$x = m = \begin{pmatrix} (\texttt{00000000}, \texttt{00000001}), (\texttt{00000000}, \texttt{00000001}), \\ (\texttt{00000000}, \texttt{00000001}), (\texttt{00000000}, \texttt{00000001}) \end{pmatrix}$$

leads to

$$F(x, m) = \begin{pmatrix} (\texttt{52afa888}, \texttt{52afa888}), (\texttt{61c0aebc}, \texttt{61c0aebc}), \\ (\texttt{1c9d4d3a}, \texttt{1c9d4d3a}), (\texttt{95f45a98}, \texttt{95f45a98}) \end{pmatrix}$$

and in *Lesamnta*-512

$$x = m = \begin{pmatrix} (\texttt{0000000000000000}, \texttt{0000000000000001}), \\ (\texttt{0000000000000000}, \texttt{0000000000000001}), \\ (\texttt{0000000000000000}, \texttt{0000000000000001}), \\ (\texttt{0000000000000000}, \texttt{0000000000000001}) \end{pmatrix}$$

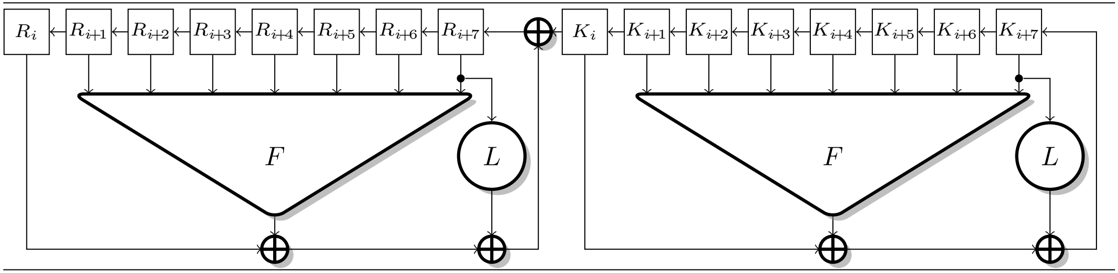
leads to

$$F(x, m) = \begin{pmatrix} (\texttt{b0421baf4899c67e}, \texttt{b0421baf4899c67e}), \\ (\texttt{e6b528589fadd0ce}, \texttt{e6b528589fadd0ce}), \\ (\texttt{3547c4021eb4c7ee}, \texttt{3547c4021eb4c7ee}), \\ (\texttt{a8188b26052d044d}, \texttt{a8188b26052d044d}) \end{pmatrix}$$

Faster Collisions in the Compression Function. It is possible to use the above property to find collisions in the compression function faster than exhaustive search. Using weak inputs $x = \widetilde{x}$ and $m = \widetilde{m}$, then the output satisfies $h = \overleftrightarrow{h}$, *i.e.* each of the four output words has the same top and bottom halves. This means that the output of the compression function is restricted to a subspace of size $2^{n/2}$ for *Lesamnta-n*.

Hence, by taking $2^{n/4}$ pairs of (chaining values, message blocks) which are weak, we expect to find a collision in the output of the compression function.

Figure 8.4 *ESSENCE* Round Function (iterated 32 times).



8.3.3 Attacks on the Full *Lesamnta* Hash Function

Herding Attack on *Lesamnta*. The self-similarity property of the compression function can be used to build a herding attack more efficient than the generic attack of [KK06]. In the herding attack, the adversary commits to a hash value H^*, and is given a challenge message M afterwards. His goal is to find a suffix S such that $H(M\|S) = H^*$. The generic herding attack on hash functions of n bits presented in [KK06] requires an offline time complexity of $2^{(n+\ell)/2+2}$ online time complexity of $2^{n-\ell}$ and memory of $2^{\ell+1}$. The optimal choice is $\ell = n/3$ which leads to a complexity of $2n/3$. We now present a customized herding attack using self-similarity for *Lesamnta* with complexity $2^{n/2}$ and negligible precomputation and memory.

1. Choose a weak chaining value h^*, and choose an upper-bound L on the number of message blocks of the prefix M. Then, compute the finalization function on h^*, assuming a message of $L + 2$ full blocks. This yields the committed value H^* (of a message of length $L + 2$ blocks).
2. Receive the challenge M, and append random suffixes S of $L + 1 - |M|$ blocks, until hitting a weak chaining value h. It is expected that $2^{n/2}$ random trials are sufficient.
3. Try random weak message blocks S'. We know that $H(M\|S\|S')$ is weak, with $H(M\|S\|S') = \overleftrightarrow{H(M\|S\|S')}$. Therefore, we expect to reach h^* after only $2^{n/2}$ trials.
4. Output $M\|S\|S'$ as the answer to the challenge.

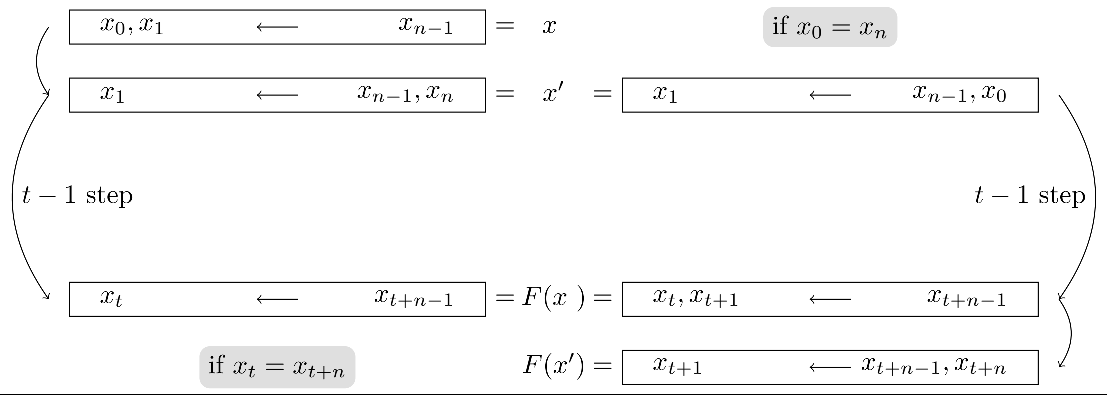
Distinguishing-H Attack on HMAC. The self-similarity property can also be used to distinguish HMAC-*Lesamnta* from HMAC instantiated with another PRF. The distinguisher follows the ideas of Wang *et. al* in [WYW⁺09], and the complexity would be $2^{3n/4}$. We note that this line of research may not be harmful, as distinguishing HMAC from PRF (for any hash function) has a complexity of $2^{n/2}$. We also note that the claimed security level of HMAC-*Lesamnta* is only $2^{n/2}$.

8.4 Application to *ESSENCE*

ESSENCE is a hash function proposal by Martin as a candidate in the SHA-3 competition [Mar08]. The design is based on two shift registers with 8 words each. One shift register is used to expand the message by generating subkeys, while the other processes the chaining value with the subkeys.

$$R_{i+8} = R_i \oplus F\left(R_{i+1}, R_{i+2}, \ldots, R_{i+7}\right) \oplus L(R_{i+7}) \oplus K_i$$
$$K_{i+8} = K_i \oplus F\left(K_{i+1}, K_{i+2}, \ldots, K_{i+7}\right) \oplus L(K_{i+7})$$

Figure 8.5 Symmetry relation on an LFSR-based function. L is a linear function defined by clocking an LFSR a fixed number of times. If the initial state of the LFSR is rotated by 1 place, there is a probability 2^{-2} that the output state is rotated as well.


$x_0, x_1 \longleftarrow x_{n-1}$ $= x$
if $x_0 = x_n$
$x_1 \longleftarrow x_{n-1}, x_n$ $= x' =$ $x_1 \longleftarrow x_{n-1}, x_0$
$t-1$ step
$t-1$ step
$x_t \longleftarrow x_{t+n-1}$ $= F(x) =$ $x_t, x_{t+1} \longleftarrow x_{t+n-1}$
if $x_t = x_{t+n}$
$F(x') =$ $x_{t+1} \longleftarrow x_{t+n-1}, x_{t+n}$

The feedback function is designed with two functions: a non-linear bit-wise function F, and a linear function L that mixes the bits inside the words. The linear function is based on clocking an LFSR a fixed number of times. The chaining value X is loaded into R_{-8}, R_{-7}, $\dots$, R_{-1}, while the message block M is loaded into K_{-8}, K_{-7}, $\dots$, K_{-1}. After 32 rounds, the output is computed as $R_{-8} \oplus R_{24}$, $R_{-7} \oplus R_{25}$, $\dots$, $R_{-1} \oplus R_{31}$.

ESSENCE has two main variants: *ESSENCE*-256 with 32-bit words (hence a message block and a chaining value of 256 bits each), and *ESSENCE*-512 with 64-bit words (hence a message block and chaining value of 512 bits). The best known attack on *ESSENCE* is a collision attack with complexity 2^{68} [NPRA+10]. The design of *ESSENCE* uses no constants. Therefore, it is possible to build a distinguisher against *ESSENCE* using a slide attack [MTT09].

8.4.1 The Self-Similarity Relation

Most of the components used in *ESSENCE* are bitwise, and the L function is the only part responsible of mixing the bits of the words. Moreover, L is based on an LFSR, and LFSRs have a nice behavior with regards to rotations. More precisely, for a binary LFSR, we have $LFSR(x) = x^{\lll 1}$ with probability 1/2, and this gives $\Pr(L(x^{\lll 1}) = L(x)^{\lll 1}) = 1/4$ as shown by Figure 8.5. Therefore, we can build a new self-similarity attack based on rotating the message and the chaining value: this gives subkeys such that $K'_i = K_i^{\lll 1}$ as opposed to $K'_i = K_{i+1}$ in a slide attack. Therefore, we consider the following relations:

$$M'_i = M_i^{\lll 1} \qquad X'_i = X_i^{\lll 1}$$

8.4.2 Constructing a Good Pair

We consider a pair of messages M, M' such that $M'_i = M_i^{\lll 1}$, ie $K'_{-8} = K_{-8}^{\lll 1}$, $K'_{-7} = K_{-7}^{\lll 1}$, $\dots$, $K'_{-1} = K_{-1}^{\lll 1}$. For each round, there is a probability $1/4$ that the new subkey K'_i is equal to $K_i^{\lll 1}$, because L is the only non-bitwise component. The full compression function uses the subkeys K_{-8} to K_{23}, so a random message M and its related message M' will give suitable subkeys with probability 2^{-48}.

To find a suitable chaining value we use a chaining value modification technique similar to the simple message modification technique of Wang *et al.* We start from round 6, by picking R_{-1}, R_0,

…, R_6 such that each value satisfies $L(R_i^{\lll 1}) = L(R_i)^{\lll 1}$, and define $R'_{-1} = R_{-1}^{\lll 1}$, $R'_0 = R_0^{\lll 1}$, …, $R'_6 = R_6^{\lll 1}$. We compute the LFSR backwards to get R_{-7}, R_{-6}, …, R_{-1} and R'_{-7}, R'_{-6}, …, R'_{-1}. By construction, we have $R'_i = R_i^{\lll 1}$ for $-1 \leq i < 7$, but we also have $R'_7 = R_7^{\lll 1}$ because the non-bitwise part used to compute R_7 is $L(R_6)$. Similarly, the non-bitwise parts used to compute $R_{-2}, R_{-3}, \dots, R_{-8}$ backwards are respectively $L(R_5)$, $L(R_4)$, $\dots L(R_{-1})$, so we actually have $R'_i = R_i^{\lll 1}$ for $-7 \leq i < 8$. Finally, with probability 2^{-48}, we have $R'_i = R_i^{\lll 1}$ for all i's.

Hence, with complexity 2^{48}, we can construct a pair of messages and chaining values such that:

$$M' = M^{\lll 1} \qquad X' = X^{\lll 1} \qquad F(M', X') = F(M, X)^{\lll 1}$$

Note that it is probably possible to use advanced message modifications to further improve this complexity.

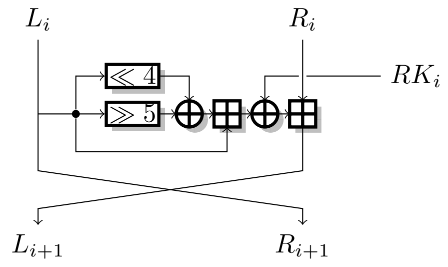
8.5 Application to XTEA

XTEA is a block cipher designed by Needham and Wheeler. XTEA is used in several places, such as in the Linux kernel, due to its simplicity. It is a Feistel network that encrypts 64-bit plaintexts with a 128-bit key. The round function is pretty simple, and the security relies more on the high number of times it is iterated: 64 rounds are the recommended setting.

8.5.1 Description of XTEA

The Feistel structure of XTEA can be described as:

$$\begin{aligned} R_{i+1} &= L_i \\ L_{i+1} &= R_i \boxplus (F(L_i) \oplus RK_i) \\ F(x) &= ((x \ll 4) \oplus (x \gg 5)) \boxplus x \end{aligned}$$



The registers are 32-bit wide, so the block size is 64 bits. The plaintext is (L_{-1}, R_{-1}), while the ciphertext is (L_{63}, R_{63}). The key schedule is rather simple: the master key K is made of four 32-bit words, and each 32-bit subkey RK_i is computed from one of the four master keys, and a round counter:

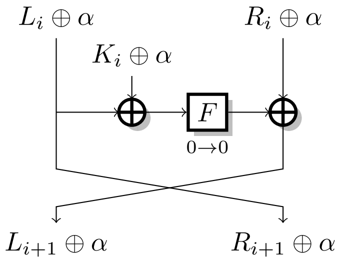
$$\begin{aligned} RK_{2i} &= (i \cdot \delta) \boxplus K_{((i \cdot \delta) \gg 11) \bmod 4} \\ RK_{2i+1} &= ((i+1) \cdot \delta) \boxplus K_{((i+1) \cdot \delta) \bmod 4} \end{aligned}$$

where $\delta =$ `0x9E3779B9` is a constant derived from the golden ratio.

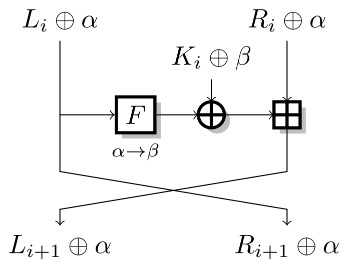
Currently, the best known attack on XTEA is due to Lu [Lu09]. It can break up to 36 rounds of XTEA in the related-key model (with four keys) using 2^{65} chosen plaintexts and $2^{126.44}$ time (or 2^{64} chosen plaintexts and 2^{104} time for a weak-key class of 2^{111} weak keys).

8.5.2 The Self-Similarity Relation

The idea of our attack on XTEA is inspired by the complementation property of the DES. In a complementation property, the difference in the plaintext and the difference in the key cancel each other *before* entering the F function. Such attacks are possible if the key schedule allows a fixed difference in all the subkeys. The key schedule of XTEA is weak enough to do so, but the key is added *after* the F function, which prevents complementation attacks. However, if there is a good differential characteristic $\alpha \to \beta$ in the F function, we can put a difference α in the plaintext, and a difference β in the key to cancel it *after* the F function. This gives an iterative related-key differential characteristic.



Complementation property



RK iterative differential on XTEA

More precisely, in the case of XTEA, we use the following weak self-similarity:

$$\begin{aligned} \psi(K_i) &= K_i \boxplus 2^{31} \boxplus 2^{26} \\ \phi(L_i, R_i) = \theta(L_i, R_i) &= \left(L_i \boxplus 2^{31}, R_i \boxplus 2^{31}\right) \end{aligned}$$

Because of the key-schedule, if $K' = \psi(K)$, then we have $RK'_i = \psi(RK_i)$ for all rounds. We also use the fact that:

$$F(x \boxplus 2^{31}) = \begin{cases} F(x) \boxplus 2^{31} \boxplus 2^{26} & \text{or} \\ F(x) \boxplus 2^{31} \boxminus 2^{26} \end{cases}$$

Therefore, the differences will cancel out as long as the carries in $F(x) \boxplus \left(2^{31} \pm 2^{26}\right)$ are the same as the carries in $RK_i \boxplus 2^{31} \boxplus 2^{26}$. For a given key K, we can compute the carry chains in each subkey, and deduce the probability of the differential characteristic. For example if there is no carry in the key, then there is a probability 1/2 that there will also be no carry in the F function. If there is a one-bit carry in the key, there is a probability 1/4 that there will be exactly a one-bit carry in the F function.

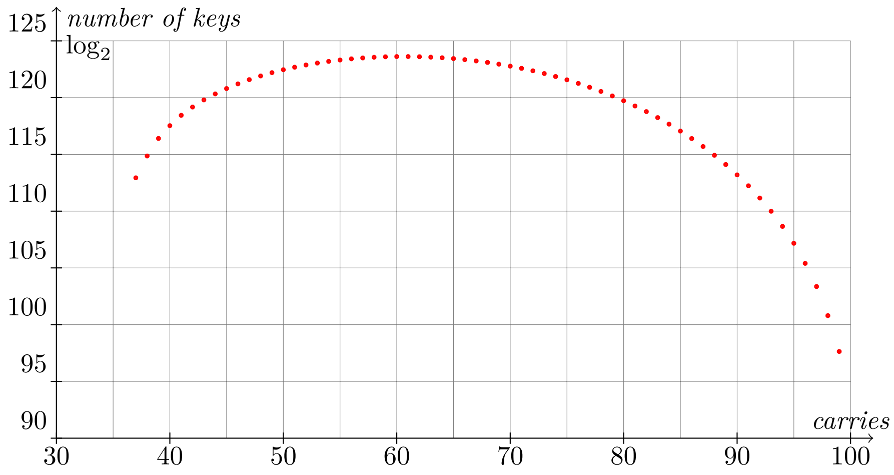
The probability of the differential path is therefore quite dependent on the key. However, we can compute exactly the subsets of the key space that correspond to a given set of carry chains. Therefore, we can count the number of keys giving a specific probability for the self-similarity relation (Figure 8.6 shows the number of keys yielding a specific number of carries). We discuss two possible attack scenarios: attacking all the keys, and attacking a class of weak keys. Note that due to the irregular key schedule, we obtain better attacks by selecting a subset of the rounds that do not start at round 0.

8.5.3 Attacking All the Keys

If we consider rounds 20 to 50, the self-similarity relation holds with probability greater than 2^{-60} for about half of the keys as shown in Figure 8.6. Moreover, rounds 51 to 55 only involve K_2 and K_3 which can be guessed. We can mount a key-recovery attack that is successful to all keys.

First, we assume that the key we are attacking is such that the probability of the path is greater than 2^{-60}. If we consider 2^{62} message pairs, then we expect at least 4 right pairs at round

Figure 8.6 Distribution of the number of carries in rounds 20–50 of XTEA.
$\log_2$ of the number of keys with a given number of carries.
48% of the keys have less than 60 carries, and 52% have less than 61 carries.



50, using partial decryption of rounds 51 to 55. Note that we do not have to do partial decryption of all the ciphertext pairs: a right pair will have no difference in the low 31 bits of L_{50} and R_{50}. Due to the limited diffusion of the round function this implies that there is no difference in the low 6 bits of L_{55} nor in the low 11 bits of R_{55}. This can be used to limit the candidates pairs to 2^{45}, thereby making the partial decryption negligible in the full attack. A wrong subkey guess is expected to lead to about $2^{-2} = 1/4$ pairs. Hence, if we consider only key proposals for K_2 and K_3 which suggest two or more pairs, we expect to deal with $2^{64} \cdot 2^{-5.23} < 2^{59}$ remaining wrong subkeys (while the probability of discarding the wrong value is only 9.2%). For each key candidate, we run an exhaustive search over K_1 and K_2 in order to recover the full key. The key recovery costs less than 2^{123}. Moreover, it is possible to discard wrong guesses using the assumption that the key yields a probability greater that 2^{-60}.

Second, we note that if the first step fails to yield the secret key, then the key is probably in the subset with more than 60 carries, which reduces the search space by 1/2. In this case, we run an exhaustive search over this subset of the keys, which needs 2^{126} time on average (in addition to the 2^{123} spend in the first step).

This gives an attack on 36 rounds (20 to 55), with data complexity 2^{63}, time complexity $2^{125.3}$ on average, and success probability more than 95%.

8.5.4 Attacking a Class of Weak Keys

If we consider rounds 10 to 55, we have a weak key class with only 60 carries, which contains $2^{107.5}$ keys out of 2^{128}. This class of weak key contains keys such that the subkeys derived from K_0, K_1, K_2 and K_3 generate respectively 12, 17, 16 and 15 carries. We can attack a key in this weak key class using 2^{62} message pairs following the same attack scenario as in the previous case. We use partial decryption of rounds 56 to 59 by guessing keys K_0 and K_1, and we can filter the ciphertext pairs using the fact a good pair will have no difference in the 11 low bits of L_{59} and in the 16 low bits of R_{59}. Note that when guessing K_0 and K_1, and when doing the exhaustive search over K_2 and K_3, we only need to consider keys in the weak-key class (the weak-key class is the product a weak key classes for K_0, K_1, K_2 and K_3). The complexity of the attack will be

$2^{-5.23}$ times the size of the class of weak keys.

This gives an attack on 50 rounds (10 to 59), with data complexity of 2^{63} chosen plaintexts, and time complexity 2^{102} on average, for a weak key class of size $2^{107.5}$.

A similar attack can be applied to XXTEA using $\psi(K_i) = K_i \boxplus 2^{26} \boxplus 2^{28}$. However, as the difference may cause more complex carry chains whose probabilities are lower, this attack can be applied only to a significantly reduced version.

8.6 Application to $\mathcal{PURE}$

$\mathcal{PURE}$ is a block cipher introduced by Jakobsen and Knudsen to demonstrate the interpolation attack [JK97]. It is designed as a block cipher with a very strong algebraic structure, and good resistance to differential and linear cryptanalysis. However, it is weak against algebraic attacks, which was the point of the article. $\mathcal{PURE}$ is not to be used as a real cipher, but serves as a demonstration that algebraic attacks can be useful against ciphers with a very strong algebraic structure.

By applying a self-similarity attack to $\mathcal{PURE}$, we intend to show that the algebraic structure of a block cipher can be used to mount a simple self-similarity attack that does not need complex polynomial computations (as opposed to Gröbner basis techniques). Moreover, self-similarities can be applied to any number of rounds, if the key schedule is weak.

8.6.1 Description of $\mathcal{PURE}$

$\mathcal{PURE}$ is a simple Feistel cipher with r rounds using a monomial S-Box:

$$L_{i+1} = R_i \qquad\qquad R_{i+1} = L_i \oplus (R_i \oplus K_i)^3.$$

All the operations are carried in the finite field $\mathbb{F}_{2^m}$: the plaintext is composed of two field elements (L_{-1}, R_{-1}) while the ciphertext is (L_{r-1}, R_{r-1}). There is no key schedule in $\mathcal{PURE}$, the key is given as r field elements $K_0, K_1, \ldots, K_{r-1}$ which are used as round subkeys.

8.6.2 The Self-Similarity Relation

Because of the strong algebraic structure of $\mathcal{PURE}$, it is natural to look for algebraic relations. In particular we can use the Frobenius mapping $(x \mapsto x^2)$ in the field $\mathbb{F}_{2^m}$: it commutes with any monomial S-Box, and it is linear. It is straightforward to check that the following is a self-similarity relation for $\mathcal{PURE}$:

$$\theta(z) = \psi(z) = \phi(z) = z^2$$

More precisely, if $K'_i = K_i^2$ for all rounds and $P'_L = P_L^2, P'_R = P_R^2$, then we will have $L'_i = L_i^2$ and $R'_i = R_i^2$ for all rounds. This gives a very efficient distinguisher for $\mathcal{PURE}$ in the related-key setting. However, we note that since $\mathcal{PURE}$ has no key-schedule, there are trivial related-key attacks. Still, a slight generalization of this initial observation leads to the discovery of a class of weak keys for $\mathcal{PURE}$.

8.6.3 A Class of Weak Keys

Using the previous self-similarity relation $\psi(k) = k^2$, we have only two fixed points for ψ. This suggests a class of weak keys of size 2^r out of 2^{mr}. In order to build a larger class of weak keys, we look for family of relations with more fixed points. For a given $\beta \in \mathbb{F}_{2^m}$, and $0 < k < m$, we now consider the following self-similarity relation:

$$\theta(z) = \psi(z) = \phi(z) = z^{2^k} \oplus \beta$$

Again, it is easy to check that this is actually a self-similarity. If $K'_i = \psi(K_i)$, $L'_i = \phi(L_i)$ and $R'_i = \phi(R_i)$, then we have:

$$\begin{aligned} R'_{i+1} &= L'_i \oplus (R'_i \oplus K'_i)^3 \\ &= L_i^{2^k} \oplus \beta \oplus \left((R_i \oplus K_i)^{2^k}\right)^3 \\ &= \left(L_i \oplus (R_i \oplus K_i)^3\right)^{2^k} \oplus \beta \\ &= \phi(R_{i+1}) \end{aligned}$$

The fixed points of ψ will be weak keys. Since ψ is distinct from the identity this can actually be used to distinguish $\mathcal{PURE}$ used with a weak key from a random function. More precisely, if we have $\beta = \alpha \oplus \alpha^{2^k}$, then a fixed point z of ψ satisfies $(z \oplus \alpha)^{2^k} = z \oplus \alpha$. Note that if k divides m, then the mapping $x \mapsto x^{2^k}$ over $\mathbb{F}_{2^m}$ admits all the subfield $\mathbb{F}_{2^k}$ as fixed points. This means that a key is weak if there exists an α such that $\forall i, K_i \oplus \alpha \in \mathbb{F}_{2^k}$. It is possible to check this with only one query: just encrypt (α, α), and test the ciphertext for self-similarity. Testing all the possible α's requires $2^{m/2}$ queries. With $k = m/2$, this yields $2^{(m+rm)/2}$ weak keys out of 2^{mr}.

8.7 Conclusion

This work shows one more time that symmetry in the building blocks of a cryptographic primitive can be dangerous. We have shown new ways to build symmetry relations based on rotations or algebraic expressions. We have also described new attacks based on these relations when there exist keys and/or plaintexts which are fixed under the similarity relations.

The common way to avoid such symmetries is to include round constants (either in the key schedule or in the actual rounds). However, our results on the hash function *Lesamnta* show that if the constants have a strong structure, it may interact with other components to suggest a self-similarity property of the entire primitive.

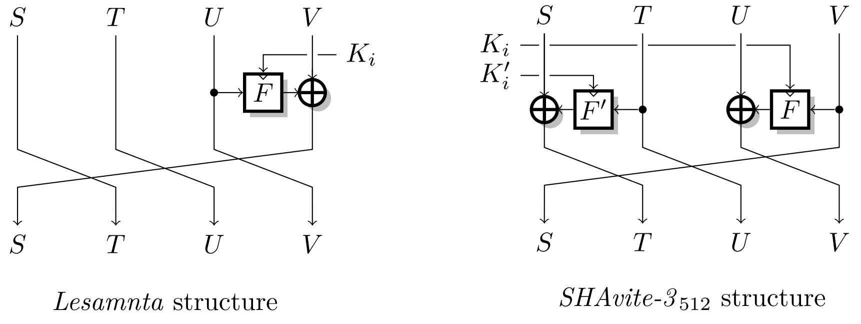
Chapter 9

Cancellation Attacks on Generalized Feistel schemes

9.1 Introduction

In this chapter we describe new attacks on generalized Feistel constructions. Our attacks are based on a cancellation property, meaning that we cancel the affects of the nonlinear operations using specially selected message words. Our technique gives an efficient way to use the degrees of freedom available in a hash function. We study two different ways to build a $4n$-bit Feistel scheme out of an n-bit round function as shown in Figure 9.1. Note that the direction of the rotation in the Feistel structure is not really important: changing the rotation is equivalent to considering decryption instead of encryption.

Figure 9.1 The Generalized Feistel Constructions Studied in this Chapter



In the ideal Luby-Rackoff case, the round functions are independent random functions. In practice, however, the keyed function $F(k, x)$ is usually defined as $P(k \oplus x)$, where P is a fixed permutation (or a fixed function). Our attacks exploit this property in order to build a cancellation property: if the fixed function P receives twice the same input, it will produce twice the same output. In a hash function setting, there is no secret key, so one can actually make sure that the inputs are the same. We consider concrete constructions based on the structures of Figure 9.1, and we show generic attacks (*i.e.* independent of the actual P function) as well as specific attacks using the properties of the fixed function.

9.1.1 Overview of the Attacks

Our attacks are based on a partial preimage attack, *i.e.* we can construct specific inputs where part of the output H is equal to a target value $\overline{H}$. To achieve such a partial preimage attack, we use truncated differentials built with the cancellation property, and we express the constraints needed on the state of the Feistel network in order to have the cancellation with probability one. We use degrees of freedom in the inputs of the compression function to satisfy those constraints. Then, we can compute some part of the output as a function of some of the remaining degrees of freedom, and try to invert the equation. The main idea is to obtain a simple equation that can be easily inverted using cancellations to limit the diffusion.

This approach is similar to the approach used to attack MD4 in Chapter 5: we consider the compression function as a system of equations, and we fix some variables to particular values in order to simplify the system.

A partial preimage attack on the compression function allows to choose k bits of the output for a cost of 2^t (with $t < k$), while the remaining $n - k$ bits are random. We can use such an attack on the compression function to target the hash function itself, in several scenarios.

Preimage Attacks. By repeating such an attack 2^{n-k} times, we can obtain a full preimage attack on the compression function, with complexity 2^{n+t-k}. This preimage attack on the compression function can be used for a second preimage attack on the hash function with complexity $2^{n+(t-k)/2}$ using a standard unbalanced meet-in-the middle attack [LM92]. Note that $2^{n+(t-k)/2} < 2^n$ if $t < k$.

Moreover, we point out that *Lesamnta* is built following the Matyas-Meyer-Oseas construction, *i.e.* the chaining value is used as a key, and the message enters the Feistel rounds. Since our partial preimage attack does not use degrees of freedom in the key (we only need the key to be known, not chosen), we can use a chaining value reached from the IV as the key. We have to repeat the partial preimage attack with many different keys in order to build a full preimage, but we can use a first message block to randomize the key. This gives a second preimage attack on the hash function with complexity 2^{t+n-k}.

Collision Attacks. The partial preimage attack can also be used to find collisions in the compression function. By generating $2^{(n-k)/2}$ inputs where k bits of the output are fixed to a common value, we expect a collision by the birthday paradox. This collision attack on the compression function costs $2^{t+(n-k)/2}$. If $t < k/2$, this is more efficient than a generic birthday attack on the compression function.

If the compression function is built with the Matyas-Meyer-Oseas mode, like *Lesamnta*, this attack translates to a collision attack on the hash function with the same complexity. However, if the compression function follows the Davies-Meyer mode, like *SHAvite-3*, this does not translate to an attack on the hash function.

9.1.2 Our results

The first candidate for the technique is the *Lesamnta* hash function. The best known generic attack against this structure is a 16-round attack by Mendel described in the submission document of *Lesamnta* [HKY08]. Using a cancellation property, we extend this attack to a generic attacks on 22-round *Lesamnta*. The attack allows to fix one of the output words for an amortized cost of 1, which gives collisions in time $2^{3n/8}$ and second preimages in time $2^{3n/4}$ for *Lesamnta-n*. Moreover, the preimage attack can be extended to 24 rounds using $2^{n/4}$ memory. We follow

Table 9.1 Cancellation property on *Lesamnta*.
On the left side, we have full diffusion after 9 rounds.
On the right side, we use the cancellation property to control the diffusion of the differences.

i	S_i	T_i	U_i	V_i		S_i	T_i	U_i	V_i	
0	x	-	-	-		x	-	-	-	
1	-	x	-	-		-	x	-	-	
2	-	-	x	-		-	-	x	-	
3	y_1	-	-	x	$x \to y_1$	y	-	-	x	$\underline{x \to y}$
4	x	y_1	-	-		x	y	-	-	
5	-	x	y_1	-		-	x	y	-	
6	z	-	x	y_1	$y_1 \to z$	z	-	x	y	$y \to z$
7	y'	z	-	x	$x \to y_2,\ y' = y_1 \oplus y_2$	$\underline{-}$	z	-	x	$\underline{x \to y}$
8	x	y'	z	-		x	$\underline{-}$	z	-	
9	w	x	y'	z	$z \to w$	w	x	$\underline{-}$	z	$z \to w$

with adaptations of the 24-round attacks without memory using specific properties of *Lesamnta*'s round function.

The second target for our technique is the hash function *SHAvite-3*$_{512}$. We first show a 10-round attack using a cancellation property on the generalized Feistel structure of *SHAvite-3*$_{512}$. The attack also works for the tweaked version of *SHAvite-3*$_{512}$, and allows fixing one out of the four output words. This allows a second preimage attack on 10-round *SHAvite-3*$_{512}$ that takes about 2^{480} time and 2^{32} memory, or 2^{448} time and 2^{128} (intermediate trade-offs are possible). Second, we describe a pseudo-attack against the full 14-round compression function of *SHAvite-3*$_{512}$, using additional degrees of freedom in the salt input.

Finally, we show some applications to block ciphers. We describe an integral attack on 21 rounds of the inner block cipher of *Lesamnta* using a cancellation property, and a new truncated differential for *SMS4*.

The chapter is organized as follows. Section 9.2 explains the basic idea of our cancellation attacks. Our results on *Lesamnta* are presented in Section 9.3, while application to *SHAvite-3*$_{512}$ is discussed in Section 9.4. Finally, application to the inner block cipher of *Lesamnta* is shown in Appendix 9.A, while an attack on *SMS4* is described in Appendix 9.B. These results are summarized in Section 9.6.

9.2 The Cancellation Property

In this chapter we apply cancellation cryptanalysis to generalized Feistel schemes. The main idea of this technique is to impose constraints on the values of the state in order to limit the diffusion in the Feistel structure. When attacking a hash function, we have many degrees of freedom from the message and the chaining value, and it is important to find efficient ways to use those degrees of freedom.

Table 9.1 shows the diffusion of a single difference in *Lesamnta*. After 9 rounds, all the state words are active. However, we note that if the transitions $x \to y_1$ at rounds 3 and $x \to y_2$ at round 7 actually go to the *same* y, *i.e.* $y_1 = y_2$, this limits the diffusion. In the ideal case, the round functions are all independent, and the probability of getting the same output difference is very small. However, in practice, the round functions are usually all derived from a single fixed permutation (or function). Therefore, if we add some constraints so that the input *values* of the

Table 9.2 Cancellation property on *SHAvite-3*$_{512}$.
On the left side, we have full diffusion after 4 rounds.
On the right side, we use the cancellation property to control the diffusion.

i	S_i	T_i	U_i	V_i		S_i	T_i	U_i	V_i	
0	x	-	-	-		x	-	-	-	
1	-	x	-	-		-	x	-	-	
2	-	y_1	x	-	$x \to y_1$	-	y	x	-	$\underline{x \to y}$
3	-	z	y_1	x	$y_1 \to z$	-	z	y	x	$y \to z$
4	x	w	z	y'	$x \to y_2,\ y' = y_1 \oplus y_2$	x	w	z	$\underline{-}$	$\underline{x \to y}$

fixed permutation at round 3 and 7 are the same, then we have the same output values, and therefore the same output difference with probability one. This is the *cancellation property*.

Similarly, Table 9.2 shows the diffusion of a difference in *SHAvite-3*$_{512}$. If the transitions $x \to y_1$ at round 2 and $x \to y_2$ at round 4 actually go to the *same* y, we can limit the diffusion.

Our attacks use an important property of the Feistel schemes of *Lesamnta* and *SHAvite-3*$_{512}$: the diffusion is relatively slow. When a difference is introduced in the state, it takes several rounds to affect the full state and two different round functions can receive the same input difference x. Note that the slow diffusion of *Lesamnta* is the basis of a 16-round attack in [HKY08] (recalled in Section 9.3.2), and the slow diffusion of *SHAvite-3*$_{512}$ gives a similar 8-round attack [M$^+$09]. Our new attacks can be seen as extensions of those.

We now describe how to enforce conditions of the state so as to have this cancellation with probability 1. Our attacks are independent of the round function, as long as all the round functions are derived from a single function as $F_i(X_i) \triangleq F(K_i \oplus X_i)$.

9.2.1 Generic Properties of $F_i(X_i) = F(K_i \oplus X_i)$

We assume that the round functions F_i are built by applying a fixed permutation (or function) F to $K_i \oplus X_i$, where K_i is a round key and X_i is the state input. This practice is common in many primitives like DES, *SMS4*, GOST, or *Lesamnta*.

This implies the followings, for all i, j, k:

(i) $\exists c_{i,j} : \forall x,\ F_i(x \oplus c_{i,j}) = F_j(x)$.

(ii) $\forall \alpha,\ \#\{x : F_i(x) \oplus F_j(x) = \alpha\}$ is even.

(iii) $\bigoplus_x F_k\big(F_i(x) \oplus F_j(x)\big) = 0$.

Property (i) is the basis of our cancellation attack. We refer to it as the *cancellation property*. It states that if the inputs of two round functions are related by a specific fixed difference, then the outputs of both rounds are equal. The reminder of the chapter is exploring this property.

Properties (ii) and (iii) can be used in an integral attack, as shown in Appendix 9.A. Note that Property (ii) is a well known fact from differential cryptanalysis.

Proof.

(i) Set $c_{ij} = K_i \oplus K_j$.

(ii) If $K_i = K_j$, then $\forall x,\ F_i(x) \oplus F_j(x) = 0$. Otherwise, let x be such that $F_i(x) \oplus F_j(x) = \alpha$. Then $F_i(x \oplus K_i \oplus K_j) \oplus F_j(x \oplus K_i \oplus K_j) = F_j(x) \oplus F_i(x) = \alpha$. Therefore x is in the set if and only if $x \oplus K_i \oplus K_j$ is in the set, and all the elements can be grouped in pairs.

(iii) Each term $F_k(\alpha)$ in the sum appears an even number of times following (ii). □

Table 9.3 Values of the Registers for Five Rounds of *Lesamnta*.

i	S_i	T_i	U_i	V_i
2	a	b	c	d
3	$F_2(c) \oplus d$	a	b	c
4	$F_3(b) \oplus c$	$F_2(c) \oplus d$	a	b
5	$F_4(a) \oplus b$	$F_3(b) \oplus c$	$F_2(c) \oplus d$	a
6	$F_5(F_2(c) \oplus d) \oplus a$	$F_4(a) \oplus b$	$F_3(b) \oplus c$	$F_2(c) \oplus d$
7	$\cancel{F_6(F_3(b) \oplus \underline{c})} \oplus \cancel{F_2(\underline{c})} \oplus d$	$F_5(F_2(c) \oplus d) \oplus a$	$F_4(a) \oplus b$	$F_3(b) \oplus c$

9.2.2 Using the Cancellation Property

To better explain the cancellation property, we describe how to use it with the truncated differential of Table 9.1. In Table 9.3, we show the *values* of the registers during the computation of the truncated differential, starting at round 2 with $(S_2, T_2, U_2, V_2) = (a, b, c, d)$. To use the cancellation property, we want to make S_7 independent of c. Since we have $S_7 = F_6(F_3(b) \oplus \underline{c}) \oplus F_2(\underline{c}) \oplus d$, we can cancel the highlighted terms using property (i). The dependency of S_7 on c disappears if $F_3(b) = K_2 \oplus K_6$, *i.e.* if $b = F_3^{-1}(K_2 \oplus K_6)$:

$$\begin{aligned} S_7 &= F_6(F_3(b) \oplus c) \oplus F_2(c) \oplus d \\ &= F(K_6 \oplus F_3(b) \oplus c) \oplus F(K_2 \oplus c) \oplus d \\ &= F(K_2 \oplus c) \oplus F(K_2 \oplus c) \oplus d = d. \end{aligned}$$

Therefore, we can put any value c in U_2, and it does not affect S_7 as long as we fix the value of T_2 to be $F^{-1}(K_2 \oplus K_6) \oplus K_3$. Note that in a hash function, we can compute $F^{-1}(K_2 \oplus K_6) \oplus K_3$ since the keys are known to the adversary (or controlled by him), and we can choose to have this value in T_2.

This shows the three main requirements of our cancellation attacks:
- The generalized Feistel structures we study have a relatively slow diffusion. Therefore, the same difference can be used as the input difference of two different round functions.
- The round functions are built from a fixed permutation (or a fixed function), using a small round key. This differs from the ideal Luby-Rackoff case where all round functions are chosen independently at random.
- In a hash function setting the key is known to the adversary, and he can control some of the inner values.

Note that some of these requirements are not strictly necessary. For example, we show a 21-round integral attack on *Lesamnta*, without knowing the keys in Section 9.A. Moreover, in Section 9.4 we show attacks on *SHAvite-3*$_{512}$, where the round functions use more keying material.

9.3 Application to *Lesamnta*

9.3.1 A Short Description of *Lesamnta*

Lesamnta is a hash function proposal by Hirose, Kuwakado, and Yoshida as a candidate in the SHA-3 competition [HKY08]. It is based on a 32-round unbalanced Feistel scheme with four registers used in MMO mode. The key schedule is also based on a similar Feistel scheme. The round function can be written as:

$$S_{i+1} = V_i \oplus F\left(U_i \oplus K_i\right) \qquad T_{i+1} = S_i \qquad U_{i+1} = T_i \qquad V_{i+1} = U_i$$

Alternatively, we can write it with a single register X, equivalent to the original S

$$X_{i+4} = X_i \oplus F\left(X_{i+1} \oplus K_{i+3}\right)$$

where $K_0, \ldots, K_{31}$ are round keys derived from the chaining value, and the state register X is initialized with the message in X_{-3}, X_{-2}, X_{-1}, X_0. The output of the compression function is $X_{-3} \oplus X_{29}, X_{-2} \oplus X_{30}, X_{-1} \oplus X_{31}, X_0 \oplus X_{32}$.

9.3.2 Previous Results on *Lesamnta*

The best known attack on *Lesamnta* is the self-similarity attack of Section 8.3, published in [BDFL10]. Following this attack, the designers have tweaked *Lesamnta* by changing the round constants [HKY09]. In this chapter we consider attacks that work with any round constants, and thus are applicable to the tweaked version as well.

Several attacks on reduced-round *Lesamnta* are presented in the submission document [HKY08]. A series of 16-round attacks for collisions and (second) preimage attacks are presented, all of which are based on the following 16-round truncated differential with probability 1:

i	S_i	T_i	U_i	V_i	
0	x_1	x_2	x_3	$x \oplus x_4$	
1	x	x_1	x_2	x_3	$x_3 \to x_4$
2	-	x	x_1	x_2	$x_2 \to x_3$
3	-	-	x	x_1	$x_1 \to x_2$
4	-	-	-	x	$x \to x_1$
5	x	-	-	-	
6	-	x	-	-	
7	-	-	x	-	
8	?	-	-	x	
9	x	?	-	-	
11	-	x	?	-	
11	?	-	x	?	
12	?	?	-	x	
13	x	?	?	-	
14	?	x	?	?	
15	?	?	x	?	
16	?	?	?	x	
FF	?	?	?	x_4	

where [1]

$$\begin{aligned}
x_3 &= M_2 \oplus F^{-1}(F(M_2 \oplus K_0) \oplus x_4) \oplus K_0, && \textit{i.e. } F_0(U_0) \oplus F_0(U_0 \oplus x_3) = x_4 \\
x_2 &= M_1 \oplus F^{-1}(F(M_1 \oplus K_1) \oplus x_3) \oplus K_1, && \textit{i.e. } F_1(U_1) \oplus F_1(U_1 \oplus x_2) = x_3 \\
x_1 &= M_0 \oplus F^{-1}(F(M_0 \oplus K_2) \oplus x_2) \oplus K_2, && \textit{i.e. } F_2(U_2) \oplus F_2(U_2 \oplus x_1) = x_2 \\
x &= (M_3 \oplus F(M_2 \oplus K_0)) \oplus F^{-1}(F(M_3 \oplus K_3 \oplus F(M_2 \oplus K_0)) \oplus x_1) \oplus K_3, \\
& && \textit{i.e. } F_3(U_3) \oplus F_3(U_3 \oplus x\ ) = x_1
\end{aligned}$$

1. Note that the expression given for x in [HKY08] is incorrect.

and M_i are the corresponding message words of the message block.

This truncated differential allows fixing the fourth output word to a constant value determined by the adversary using two queries to the compression function. One first picks a random message m, and computes the difference x_4 between the desired value $\overline{H}_4$ and the actual value $H_4 = V_o \oplus V_{16}$ of the fourth output word. Since the key is known, it is easy to compute x_3 from x_4, and similarly x_2, x_1 and x, as shown above. Then, by picking $m' = m \oplus (x_1, x_2, x_3, x \oplus x_4)$, it is assured that the fourth output word is equal to $\overline{H}_4$.

This allows a collision attack (of expected time complexity 2^{97}) and second preimage attack (of expected time complexity 2^{193}). We note that this property is independent of F (as long as F is bijective), and can be applied even when the round functions are ideal independent permutations.

In the next sections we show new attacks using the cancellation property. We first show some attacks that are generic in F, as long as the round functions are defined as $F_i(X_i) = F(K_i \oplus X_i)$, and then improved attacks using specific properties of the round functions of *Lesamnta*.

9.3.3 Generic Attacks

Our attacks are based on the differential of Table 9.4, which is an extension of the differential of Table 9.1. In this differential we use the cancellation property three times to control the diffusion. Note that we do not have to specify the values of y, z, w, r and t. This specifies a truncated differential for *Lesamnta*: starting from a difference $(x, -, -, -)$, we reach a difference $(?, ?, ?, x_1)$ after 22 rounds. In order to use this truncated differential in our cancellation attack, we use two important properties: first, by adding constraints on the state, the truncated differential is followed with probability 1; second, the transition $x \to x_1$ is known because the key and values are known. Therefore, we can actually adjust the value of the last output word.

In order to express the constraints that we need for the cancellation properties, we look at the *values* of the registers for this truncated differential. In Table 9.5, we begin at round 2 with $(S_2, T_2, U_2, V_2) = (a, b, c, d)$, and we compute the state values up to round 19. This is an extension of the values computed in Table 9.3.

We can see that we have

$$X_{19} = F(c \oplus \alpha) \oplus \beta,$$

where

$$\alpha = K_{10} \oplus F_7(F_4(a) \oplus b) \oplus F_3(b) \qquad \text{and} \qquad \beta = d$$

provided that (a, b, d) is the unique triplet satisfying the following cancellation conditions:

Round 7: we have $F_6(F_3(b) \oplus \underline{c}) \oplus F_2(\underline{c})$. They cancel if:
$F_3(b) = c_{2,6} = K_2 \oplus K_6$ $\qquad$ *i.e.* $b = F_3^{-1}(K_2 \oplus K_6)$

Round 13: we have $F_{12}(F_9(d) \oplus \underline{F_5(F_2(c) \oplus d) \oplus a}) \oplus F_8(\underline{F_5(F_2(c) \oplus d) \oplus a})$. They cancel if:
$F_9(d) = c_{8,12} = K_8 \oplus K_{12}$ $\qquad$ *i.e.* $d = F_9^{-1}(K_8 \oplus K_{12})$

Round 19: we have $F_{18}(F_{15}(F_4(a) \oplus b) \oplus \underline{X_{12}}) \oplus F_{14}(\underline{X_{12}})$. They cancel if:
$F_{15}(F_4(a) \oplus b) = c_{14,18} = K_{14} \oplus K_{18}$ $\qquad$ *i.e.* $a = F_4^{-1}(F_{15}^{-1}(K_{14} \oplus K_{18}) \oplus b)$

Note that a, b, d and α, β are uniquely determined from the subkeys. Hence, one can set X_{19} to any desired value X_{19}^* by setting $c = F^{-1}(X_{19}^* \oplus \beta) \oplus \alpha$.

22-round Attacks. The truncated differential of Table 9.4 can be used to attack 22-round *Lesamnta*. We start with the state at round 2 $(S_2, T_2, U_2, V_2) = (a, b, c, d)$ satisfying the cancellation properties, and we can compute how the various states depend on c, as shown in Table 9.6. A dash (-) is used to denote a value that is independent of c. We know exactly how c affects the last output word, and we can select c in order to get a specific value at the output. Suppose we

Table 9.4 Cancellation Property on 22 Rounds of *Lesamnta*

i	S_i	T_i	U_i	V_i	
0	x	-	-	-	
1	-	x	-	-	
2	-	-	x	-	
3	y	-	-	x	$\underline{x \to y}$
4	x	y	-	-	
5	-	x	y	-	
6	z	-	x	y	$y \to z$
7	$\underline{-}$	z	-	x	$\underline{x \to y}$
8	x	$\underline{-}$	z	-	
9	w	x	$\underline{-}$	z	$\underline{z \to w}$
10	z	w	x	$\underline{-}$	
11	x_1	z	w	x	$x \to x_1$
12	r	x_1	z	w	$w \to x \oplus r$
13	$\underline{-}$	r	x_1	z	$\underline{z \to w}$
14	?	$\underline{-}$	r	x_1	
15	$x_1 + t$	?	$\underline{-}$	r	$\underline{r \to t}$
16	r	$x_1 + t$	?	$\underline{-}$	
17	?	r	$x_1 + t$	?	
18	?	?	r	$x_1 + t$	
19	$\underline{x_1}$	?	?	r	$\underline{r \to t}$
20	?	$\underline{x_1}$	?	?	
21	?	?	$\underline{x_1}$	?	
22	?	?	?	$\underline{x_1}$	
FF	?	?	?	x_1	

are given a set of subkeys, and a target value $\overline{H}$ for the fourth output word. Then the attack proceeds as follows:

1. Set a, b, and d to the values that allow the cancellation property.
   Then we have $V_0 \oplus V_{22} = \eta \oplus F(c \oplus \alpha) \oplus \beta$, as shown in Table 9.6.
2. Compute c as $F^{-1}(\overline{H} \oplus \eta \oplus \beta) \oplus \alpha$.
3. This sets the state at round 2: $(S_2, T_2, U_2, V_2) \triangleq (a, b, c, d)$.
   With this state, we have $V_0 \oplus V_{22} = \overline{H}$.
4. Compute the round function backwards up to round 0, to get the input $(S_0, T_0, U_0.V_0)$.

This costs less than one compression function call, and does not require any memory.

For a given chaining value (*i.e.* a set of subkeys), this algorithm can only output one message. To build a full preimage attack or a collision attack on the compression function, this has to be repeated with random chaining values. Since the attack works for any chaining value, we can build attacks on the hash function using a prefix block to randomize the chaining value. This gives a collision attack with complexity 2^{96} (2^{192} for *Lesamnta*-512), and a second-preimage attack with complexity 2^{192} (2^{384} for *Lesamnta*-512).

24-round Attacks. We can add two rounds at the beginning of the truncated differential at the cost of some memory. The resulting 24-round differential is given in Table 9.8. The output

Table 9.5 Values of the Register for the 22-round Cancellation Property of *Lesamnta*

i	$X_i (= S_i)$
-1	d
0	c
1	b
2	a
3	$F_2(c) \oplus d$
4	$F_3(b) \oplus c$
5	$F_4(a) \oplus b$
6	$F_5(F_2(c) \oplus d) \oplus a$
7	$\cancel{F_6(F_3(b) \oplus c)} \oplus \cancel{F_2(c)} \oplus d$
8	$F_7(F_4(a) \oplus b) \oplus F_3(b) \oplus c$
9	$F_8(F_5(F_2(c) \oplus d) \oplus a) \oplus F_4(a) \oplus b$
10	$F_9(d) \oplus F_5(F_2(c) \oplus d) \oplus a$
11	$F_{10}(F_7(F_4(a) \oplus b) \oplus F_3(b) \oplus c) \oplus d$
12	$F_{11}(F_8(F_5(F_2(c) \oplus d) \oplus a) \oplus F_4(a) \oplus b) \oplus F_7(F_4(a) \oplus b) \oplus F_3(b) \oplus c$
13	$\cancel{F_{12}(F_9(d) \oplus F_5(F_2(c) \oplus d) \oplus a)} \oplus \cancel{F_8(F_5(F_2(c) \oplus d) \oplus a)} \oplus F_4(a) \oplus b$
14	?
15	$F_{14}(X_{12}) \oplus F_{10}(F_7(F_4(a) \oplus b) \oplus F_3(b) \oplus c) \oplus d$
16	$F_{15}(F_4(a) \oplus b) \oplus X_{12}$
17	?
18	?
19	$\cancel{F_{18}(F_{15}(F_4(a) \oplus b) \oplus X_{12})} \oplus \cancel{F_{14}(X_{12})} \oplus F_{10}(F_7(F_4(a) \oplus b) \oplus F_3(b) \oplus c) \oplus d$

Table 9.6 Collision and Preimage Characteristic for the 22-Round Attack

i	S_i	T_i	U_i	V_i
0	c	-	-	η
1	-	c	-	-
2	-	-	c	-
2–19	Repeated Cancellation Property: Table 9.5			
19	$F(c \oplus \alpha) \oplus \beta$	?	?	?
20	?	$F(c \oplus \alpha) \oplus \beta$	?	?
21	?	?	$F(c \oplus \alpha) \oplus \beta$	?
22	?	?	?	$F(c \oplus \alpha) \oplus \beta$
FF	?	?	?	$\eta \oplus F(c \oplus \alpha) \oplus \beta$

η, α and β can be computed from a, b, d and the key:
$\eta = b \oplus F_0(a \oplus F_3(d))$, $\alpha = K_{11} \oplus F_8(F_5(a) \oplus b) \oplus F_4(b)$, $\beta = d$.

word we try to control is equal to $F(c \oplus \gamma) \oplus F(c \oplus \alpha)$, for some constants α, and γ that depend on the chaining value (note that $\beta = \lambda$ in Table 9.8). We define a family of functions $h_\mu(x) = F(x) \oplus F(x \oplus \mu)$, and for a given target value $\overline{H}$, we tabulate $\varphi_{\overline{H}}(\mu) = h_\mu^{-1}(\overline{H})$. For each μ, $\varphi_{\overline{H}}(\mu)$ is a possibly empty set, but the average size is one (the non-empty values form a partition of the input space). In the special case where $\overline{H} = 0$, $\varphi_0(\mu)$ is empty for all $\mu \neq 0$, and $\varphi_0(0)$ is the full space.

Table 9.7 Computing Values Backwards from the State $(S_4, T_4, U_4, V_4) = (a, b, c, d)$

i	X_i
-3	$d \oplus F_0(c \oplus F_1(b \oplus F_2(a \oplus F_3(d))))$
-2	$c \oplus F_1(b \oplus F_2(a \oplus F_3(d)))$
-1	$b \oplus F_2(a \oplus F_3(d))$
0	$a \oplus F_3(d)$
1	d
2	c
3	b
4	a

$V_0 = X_{-3} = \lambda \oplus F_0(c \oplus \gamma)$, with $\lambda = d$

Table 9.8 Collision and Preimage Path for the 24-round Attack

i	S_i	T_i	U_i	V_i
0	-	-	$c \oplus \gamma$	$F(c \oplus \gamma) \oplus \lambda$
1	-	-	-	$c \oplus \gamma$
2	c	-	-	-
3	-	c	-	-
4	-	-	c	-
4–21	Repeated Cancellation Property: Table 9.5			
21	$F(c \oplus \alpha) \oplus \beta$	?	?	?
22	?	$F(c \oplus \alpha) \oplus \beta$	?	?
23	?	?	$F(c \oplus \alpha) \oplus \beta$	?
24	?	?	?	$F(c \oplus \alpha) \oplus \beta$

α, β, γ and λ can be computed from a, b, d and the key by:
$\alpha = K_{13} \oplus F_{10}(F_7(a) \oplus b) \oplus F_6(b)$, $\beta = d$ and
$\gamma = F_1(b \oplus F_2(a \oplus F_3(d)))$, $\lambda = d$

We store $\varphi_{\overline{H}}$ in a table of size $2^{n/4}$, and we can compute it in time $2^{n/4}$ by looking for values such that $F(x) \oplus F(y) = \overline{H}$ (this gives $\varphi_{\overline{H}}(x \oplus y) = x$). Using this table, we are able to choose one output word just like in the 22-round attack.

We start with a state $(S_4, T_4, U_4, V_4) = (a, b, c, d)$ such that a, b, d satisfy the cancellation conditions, and we compute α, β, γ, λ. If we use $c = u \oplus \alpha$, where $u \in \varphi_{\overline{H}}(\alpha \oplus \gamma) = h^{-1}_{\alpha \oplus \gamma}(\overline{H})$, we have:

$$\begin{aligned} V_0 \oplus V_{24} &= F(c \oplus \gamma) \oplus F(c \oplus \alpha) \\ &= F(u \oplus \alpha \oplus \gamma) \oplus F(u) = h_{\alpha \oplus \gamma}(u) = \overline{H} \end{aligned}$$

On average this costs one compression function evaluation to find a $n/4$-bit partial preimage. If the target value is 0, this only succeeds if $\alpha \oplus \gamma = 0$, but in this case it gives $2^{n/4}$ solutions. This gives a preimage attack with complexity $2^{3n/4}$ using $2^{n/4}$ memory.

Note that it is possible to make a time-memory trade-off with complexity 2^{n-k} using 2^k memory for $k < n/4$.

9.3.4 Dedicated 24-round Attacks on *Lesamnta*

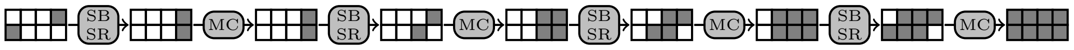
We now describe how to use specific properties of the round functions of *Lesamnta* to remove the memory requirement of our 24-round attacks.

Neutral Subspaces in F_{256}. The F function of *Lesamnta*-256 has a property that limits the difference propagation to linear subspaces. Namely, we identified two linear subspaces Γ and Λ for which

$$x \oplus x' \in \Gamma \Rightarrow F(x) \oplus F(x') \in \Lambda$$

The subspaces Γ and Λ have dimensions of 16 and 48, respectively.

The AES-like round function of *Lesamnta*-256 achieves full diffusion of the values after its four rounds, but some linear combinations of the output are not affected. Starting from a single active diagonal, we have:



All the output bytes are active, but there are some linear relations between them. More precisely, the inverse MixColumns operation leads to a difference with two inactive bytes. Therefore, we can equivalently say that there are 16 linear relations of the output bits that are not affected by 16 input bits.

Collision and Second Preimage Attacks on Lesamnta-256. Using this property, we can choose 16 linear relations of the output of the family of function h_μ, or equivalently, choose 16 linear relations of the output of the compression function.

Let $\bar{\Lambda}$ be a supplementary subspace of Λ. Any 64-bit value x can be written as $x = x_\Lambda + x_{\bar{\Lambda}}$, where $x_\Lambda \in \Lambda$ and $x_{\bar{\Lambda}} \in \bar{\Lambda}$. We show how to find values x such that $h_\mu(x)_{\bar{\Lambda}} = \overline{H}_{\bar{\Lambda}}$ for an amortized cost of one, without memory:

1. Compute $h_\mu(u)$ for random u's until $h_\mu(u)_{\bar{\Lambda}} = \overline{H}_{\bar{\Lambda}}$
2. For all v in Γ, we have $h_\mu(u+v)_{\bar{\Lambda}} = \overline{H}_{\bar{\Lambda}}$

This gives 2^{16} messages with 16 chosen relations for a cost of 2^{16}. It allows a second-preimage attack on 24-round *Lesamnta*-256 with complexity 2^{240}, and a collision attack with complexity 2^{120}, both memoryless.

Symmetries in F_{256} and F_{512}. As seen in Chapter 8, symmetry properties of the AES also give symmetry properties of the F functions of *Lesamnta*. More specifically if an AES state is such that the left half is equal to the right half, then this property still holds after any number of SubBytes, ShiftRows, and MixColumns operations [LSWD04]. Explicitly, after one AES-like round of *Lesamnta*-256, we have:

$$\begin{array}{|c|c|c|c|}\hline w & x & w & x \\ \hline y & z & y & z \\ \hline\end{array} \xrightarrow{AES-like} \begin{array}{|c|c|c|c|}\hline w' & x' & w' & x' \\ \hline y' & z' & y' & z' \\ \hline\end{array} \text{ where } \begin{cases} w' = 2 \bullet S[w] \oplus S[z] & x' = 2 \bullet S[x] \oplus S[y] \\ y' = S[w] \oplus 2 \bullet S[z] & z' = S[x] \oplus 2 \bullet S[y] \end{cases}$$

And similarly in *Lesamnta*-512:

$$\begin{array}{|c|c|c|c|}\hline w & x & w & x \\ \hline y & z & y & z \\ \hline s & t & s & t \\ \hline u & v & u & v \\ \hline\end{array} \xrightarrow{AES} \begin{array}{|c|c|c|c|}\hline w' & x' & w' & x' \\ \hline y' & z' & y' & z' \\ \hline s' & t' & s' & t' \\ \hline u' & v' & u' & v' \\ \hline\end{array}$$

When we consider the F functions of *Lesamnta*, we have that: *if $x \oplus K_i$ is symmetric, then $F_i(x) = F(x \oplus K_i)$ is also symmetric.* More precisely, if we denote the set of symmetric values by $\mathcal{S}$, we use the following property:

$$x, x' \in \mathcal{S} \Rightarrow F(x) \oplus F(x') \in \mathcal{S}$$

Collision Attacks on Lesamnta-256 and Lesamnta-512. This property can be used for an improved collision attack. As seen earlier we have $V_0 \oplus V_{24} = F(c \oplus \gamma) \oplus F(c \oplus \alpha)$. In order to use the symmetry property, we first select random chaining values, and we compute the value of α and γ until $\alpha \oplus \gamma$ is symmetric ($\alpha \oplus \gamma \in \mathcal{S}$). Then, if we select c such that $c \oplus \gamma \in \mathcal{S}$, we also have $c \oplus \alpha \in \mathcal{S}$, and this gives $V_0 \oplus V_{24} \in \mathcal{S}$.

We need to try 2^{32} (respectively, 2^{64} for *Lesamnta*-512) random chaining values in order to get $\alpha \oplus \gamma \in \mathcal{S}$, but once we have a good chaining value, we can use it with 2^{32} (respectively, 2^{64}) messages, and for each one $V_0 \oplus V_{24} \in \mathcal{S}$. So we can have 32 equalities between output bits (respectively, 64 equalities) for an average cost of one compression function call. This leads to a collision attack with complexity 2^{112} for *Lesamnta*-256, and 2^{224} for *Lesamnta*-512. In Appendix 9.C, we give an example of inputs where 64 bits of the output are set to 1 using this property.

9.4 Application to *SHAvite-3*$_{512}$

SHAvite-3 is a hash function designed by Biham and Dunkelman for the SHA-3 competition [BD08]. It is based on a generalized Feistel construction with an AES-based round function, used in Davies-Meyer mode. In this section we study *SHAvite-3*$_{512}$, the version of *SHAvite-3* designed for output size of 257 to 512 bits. The cancellation property can not be used on *SHAvite-3*$_{256}$ because the Feistel structure is different and has a faster diffusion. We describe attacks on the *SHAvite-3*$_{512}$ hash function reduced to 10 rounds out of 14.

In the next section we will show pseudo-attacks on the full 14-round compression function, using extra degrees of freedom in the salt.

9.4.1 A Short Description of *SHAvite-3*$_{512}$

The compression function of *SHAvite-3*$_{512}$ accepts a chaining value of 512 bits, a message block of 1024 bits, a salt of 512 bits, and a bit counter of 128 bits. As this is a Davies-Meyer construction, the message block, the salt, and the bit counter enter the key schedule algorithm of the underlying block cipher. The key schedule algorithm transforms them into 112 subkeys of 128 bits each. The chaining value is then divided into four 128-bit words, and at each round two words enter the nonlinear round functions and affect the other two:

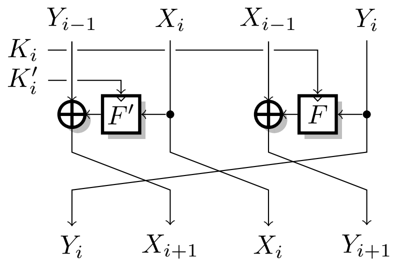
$$S_{i+1} = V_i \qquad T_{i+1} = S_i \oplus F'_i(T_i) \qquad U_{i+1} = T_i \qquad V_{i+1} = U_i \oplus F_i(V_i)$$

The nonlinear function F and F' are composed of four full rounds of AES, with 4 subkeys from the message expansion:

$$\begin{aligned} F_i(x) &= P(k^3_{0,i} \oplus P(k^2_{0,i} \oplus P(k^1_{0,i} \oplus P(k^0_{0,i} \oplus x)))) \\ F'_i(x) &= P(k^3_{1,i} \oplus P(k^2_{1,i} \oplus P(k^1_{1,i} \oplus P(k^0_{1,i} \oplus x)))) \end{aligned}$$

where P is one AES round (without the AddRoundKey operation).

Figure 9.2 The Underlying Block Cipher of C_{512}



$$X_{i+1} = Y_{i-1} \oplus F_i(X_i)$$
$$Y_{i+1} = X_{i-1} \oplus F'_i(Y_i)$$

$$X_{i-1} = Y_{i+1} \oplus F'_i(Y_i)$$
$$Y_{i-1} = X_{i+1} \oplus F_i(X_i)$$

Table 9.9 Cancellation Property on 9 Rounds of *SHAvite-3*$_{512}$

i	S_i	T_i	U_i	V_i	
0	?	x_2	?	x	
1	x	-	x_2	x_1	
2	x_1	x	-	-	$x_1 \to x_2$
3	-	-	x	-	$x \to x_1$
4	-	-	-	x	
5	x	-	-	y	$\underline{x \to y}$
6	y	x	-	z	$y \to z$
7	z	$\underline{-}$	x	w	$\underline{x \to y}$, $\underline{z \to w}$
8	w	z	$\underline{-}$	?	
9	?	$\underline{-}$	z	?	$\underline{z \to w}$
FF	?	x_2	?	?	

In this section we use an alternative description of *SHAvite-3*$_{512}$ with only two variables per round, as shown in Figure 9.2. We have

$$S_i = Y_{i-1} \qquad T_i = X_i \qquad U_i = X_{i-1} \qquad V_i = Y_i$$

The message expansion is detailed in Appendix 9.D.

Note that with the original key schedule of *SHAvite-3*, a specific set of message, salt, and counter leads to all the subkeys being zero [NP09, Pey09]. Thus, the key schedule was tweaked for the second round of the SHA-3 competition. Our attack applies both to the original message expansion and to the tweaked version.

9.4.2 Cancellation Attacks on *SHAvite-3*$_{512}$

The cancellation path is described in Table 9.9, which is an extension of Table 9.2 page 192. We use the cancellation property twice to control the diffusion. Note that we do not have to specify the values of y, z, and w. Like in the *Lesamnta* attack, this path is a truncated differential, and we use constraints on the state to enforce that it is followed. Moreover, the transitions $x \to x_1$ and $x_1 \to x_2$ are known because the key is known.

Note that the round functions of *SHAvite-3*$_{512}$ are not defined as $F(k, x) = P(k \oplus x)$ for a fixed permutation P. Instead, each function takes 4 keys and it is defined as:

$$F(k_i^0, k_i^1, k_i^2, k_i^3, x) = P(k_i^3 \oplus P(k_i^2 \oplus P(k_i^1 \oplus P(k_i^0 \oplus x))))$$

Table 9.10 Values of the Registers for the 9-round Cancellation Property of *SHAvite-3*$_{512}$

i	X_i/Y_i
X_0	$b \oplus F_3(c) \oplus F_1'(c \oplus F_2(d \oplus F_3'(a)))$
Y_0	$d \oplus F_3'(a) \oplus F_1(a \oplus F_2'(b \oplus F_3(c)))$
X_1	$a \oplus F_2'(b \oplus F_3(c))$
Y_1	$c \oplus F_2(d \oplus F_3'(a))$
X_2	$d \oplus F_3'(a)$
Y_2	$b \oplus F_3(c)$
X_3	c
Y_3	a
X_4	b
Y_4	d
X_5	$a \oplus F_4(b)$
Y_5	$c \oplus F_4'(d)$
X_6	$d \oplus F_5(a \oplus F_4(b))$
Y_6	$b \oplus F_5'(c \oplus F_4'(d))$
X_7	$c \oplus \cancel{F_4'(\underline{d})} \oplus \cancel{F_6(\underline{d} \oplus F_5(a \oplus F_4(b)))}$
Y_7	$a \oplus F_4(b) \oplus F_6'(b \oplus F_5'(c \oplus F_4'(d)))$
X_8	$b \oplus F_5'(c \oplus F_4'(d)) \oplus F_7(c)$
Y_8	$d \oplus F_5(a \oplus F_4(b)) \oplus F_7'(a \oplus F_4(b) \oplus F_6'(b \oplus F_5'(c \oplus F_4'(d))))$
X_9	$a \oplus F_4(b) \oplus \cancel{F_6'(\underline{b \oplus F_5'(c \oplus F_4'(d))})} \oplus \cancel{F_8(\underline{b \oplus F_5'(c \oplus F_4'(d))} \oplus F_7(c))}$

where P is one AES round. In order to apply the cancellation property to *SHAvite-3*$_{512}$, we need that the subkeys k^1, k^2, k^3 of two functions be equal, so that $F_i(x)$ collapses to $P'(k_i^0 \oplus x)$ and F_j to $P'(k_j^0 \oplus x)$, where $P'(x) \triangleq P(k_i^3 \oplus P(k_i^2 \oplus P(k_i^1 \oplus P(x)))) = P(k_j^3 \oplus P(k_j^2 \oplus P(k_j^1 \oplus P(x))))$.

In order to express the constraints needed for the cancellation properties, we look at the *values* of the registers for this truncated differential. In Table 9.10, we begin at round 4 with $(S_4, T_4, U_4, V_4) = (Y_3, X_4, X_3, Y_4) = (a, b, c, d)$, and we compute up to round 9.

We have a cancellation property on 9 rounds under the following conditions:

Round 7 We have $F_4'(\underline{d}) \oplus F_6(\underline{d} \oplus F_5(a \oplus F_4(b)))$. They cancel if:
$F_5(a \oplus F_4(b)) = k_{1,4}^0 \oplus k_{0,6}^0$ and $(k_{1,4}^1, k_{1,4}^2, k_{1,4}^3) = (k_{0,6}^1, k_{0,6}^2, k_{0,6}^3)$.

Round 9 We have $F_6'(\underline{b \oplus F_5'(c \oplus F_4'(d))}) \oplus F_8(\underline{b \oplus F_5'(c \oplus F_4'(d))} \oplus F_7(c))$. They cancel if:
$F_7(c) = k_{1,6}^0 \oplus k_{0,8}^0$ and $(k_{1,6}^1, k_{1,6}^2, k_{1,6}^3) = (k_{0,8}^1, k_{0,8}^2, k_{0,8}^3)$.

Therefore, the truncated differential is followed if:

$$F_5(a \oplus F_4(b)) = k_{1,4}^0 \oplus k_{0,6}^0 \qquad F_7(c) = k_{1,6}^0 \oplus k_{0,8}^0 \tag{C0}$$

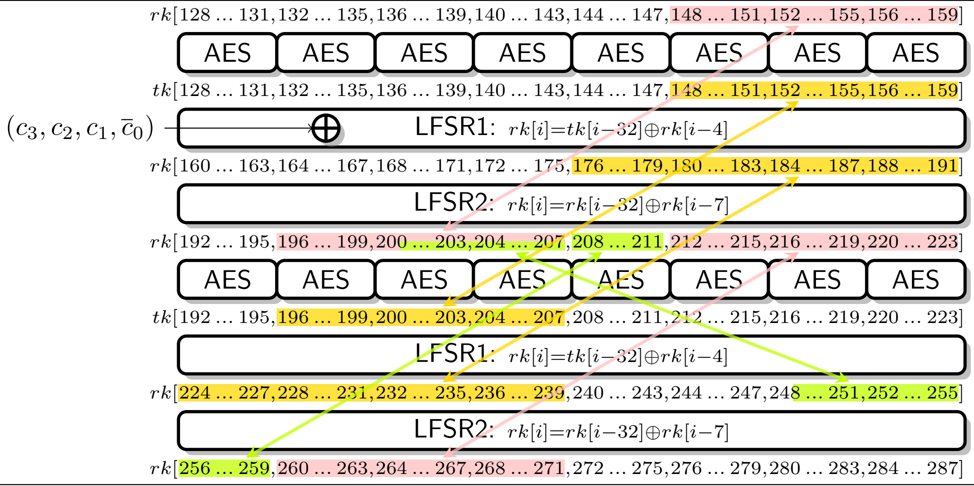
$$(k_{1,4}^1, k_{1,4}^2, k_{1,4}^3) = (k_{0,6}^1, k_{0,6}^2, k_{0,6}^3) \qquad (k_{1,6}^1, k_{1,6}^2, k_{1,6}^3) = (k_{0,8}^1, k_{0,8}^2, k_{0,8}^3) \tag{C1}$$

The constraints for the cancellation at round 7 are easy to satisfy and allow a 7-round attack on *SHAvite-3*$_{512}$. However, for a 9-round attack we have more constraints on the subkeys, and this requires special attention.

9.4.3 Dealing with the Key Expansion

Let us outline an algorithm to find a suitable message (recall that *SHAvite-3*$_{512}$ is used in a Davies-Meyer mode) for a given salt and counter value. We have to solve a system involving linear and non-linear equations, and we use the fact that the system is almost triangular. We note

Figure 9.3 Constraints in the Key Expansion of *SHAvite-3*$_{512}$
Initial constraints in pink, constraints from steps 1 to 3 in yellow, constraints from step 4 in green


rk[128 … 131,132 … 135,136 … 139,140 … 143,144 … 147,148 … 151,152 … 155,156 … 159]
AES
tk[128 … 131,132 … 135,136 … 139,140 … 143,144 … 147,148 … 151,152 … 155,156 … 159]
(c3, c2, c1, c̄0)
LFSR1: rk[i]=tk[i−32]⊕rk[i−4]
rk[160 … 163,164 … 167,168 … 171,172 … 175,176 … 179,180 … 183,184 … 187,188 … 191]
LFSR2: rk[i]=rk[i−32]⊕rk[i−7]
rk[192 … 195,196 … 199,200 … 203,204 … 207,208 … 211,212 … 215,216 … 219,220 … 223]
tk[192 … 195,196 … 199,200 … 203,204 … 207,208 … 211,212 … 215,216 … 219,220 … 223]
rk[224 … 227,228 … 231,232 … 235,236 … 239,240 … 243,244 … 247,248 … 251,252 … 255]
rk[256 … 259,260 … 263,264 … 267,268 … 271,272 … 275,276 … 279,280 … 283,284 … 287]

that it might be possible to improve our results using the technique of Khovratovich, Biryukov and Nikolić [KBN09] to find a good message efficiently.

For the cancellation attack on 9-round *SHAvite-3*$_{512}$, we need to satisfy a 768-bit condition on the subkeys, *i.e.*:

$$(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6}) \qquad (k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8}) \tag{C1}$$

Or in $rk[\cdot]$ terms:

$$rk[148,\dots,159] = rk[196,\dots,207] \qquad rk[212,\dots,223] = rk[260,\dots,271]$$

We are actually trying to solve a system of equations with:
- 224 variables: $tk[128..159]$, $tk[192..223]$ and $rk[128..287]$
- 192 equations from the key schedule (64 non-linear and 128 linear)
- 24 constraints

Therefore we have 8 degrees of freedom. The relations between the variables are shown in Figure 9.3, while the full key expansion of *SHAvite-3*$_{512}$ is described in Appendix 9.D.

Propagation of the Constraints. First, we propagate the constraints and deduce new equalities between the variables. Figure 9.3 shows the initial constraints and the propagated constraints.

1. The non-linear equations of the key-schedule give:

$$tk[156..159] = AESR\Big((rk[157,158,159,156]) \oplus (salt[12..15])\Big)$$
$$tk[204..207] = AESR\Big((rk[205,206,207,204]) \oplus (salt[12..15])\Big)$$

since $rk[156..159] = rk[204..207]$, we know that

$$tk[156..159] = tk[204..207]$$

Similarly, we get

$$tk[148..159] = tk[196..207]$$

2. From the key expansion, we have $rk[191] = rk[223] \oplus rk[216]$, and $rk[239] = rk[271] \oplus rk[264]$. Since we have the constraints $rk[223] = rk[271]$ and $rk[216] = rk[264]$, we can deduce that
$$rk[191] = rk[239]$$
Similarly, we get
$$rk[187..191] = rk[235..239]$$
3. From the linear part of the expansion, we have $rk[186] = rk[190] \oplus tk[158]$ and $rk[234] = rk[238] \oplus tk[206]$. We have seen that $rk[190] = rk[238]$ at step 2 and $tk[158] = tk[206]$ at step 1, therefore
$$rk[186] = rk[234]$$
Similarly, we get
$$rk[176..186] = rk[224..234]$$
4. Again, from the linear part of the key expansion, we have $rk[211] = rk[218] \oplus rk[186]$ and $rk[259] = rk[266] \oplus rk[234]$. We have seen that $rk[186] = rk[234]$ at step 3 and we have $rk[218] = rk[266]$ as a constraint, thus
$$rk[211] = rk[259]$$
Similarly, we obtain
$$rk[201..211] = rk[249..259]$$
Note that we have $rk[201..207] = rk[153..159]$ as a constraint, so we must have
$$rk[249..255] = rk[153..159]$$

Finding a Solution. To find a solution to the system, we use a guess and determine technique. We guess 11 state variables, and we show how to compute the rest of the state and check for consistency. Since we have only 8 degrees of freedom, we expect the random initial choice to be valid once out of $2^{32\times 3} = 2^{96}$ times. This gives a complexity of 2^{96} to find a good message.

- Choose random values for $rk[200], rk[204..207], rk[215..216], rk[220..223]$
- Compute $tk[220..223]$ from $rk[220..223]$
- Compute $rk[248..251]$ from $tk[220..223]$ and $rk[252..255]$ $(= rk[204..207])$
- Deduce $rk[201..203] = rk[249..251]$, so $rk[200..207]$ is known
- Compute $tk[152..159]$ from $rk[152..159]$ $(= rk[200..207])$
- Compute $rk[190..191]$ from $rk[215..216]$ and $rk[222..223]$
- Compute $rk[186..187]$ from $rk[190..191]$ and $rk[158..159]$
- Compute $rk[182..183]$ from $rk[186..187]$ and $rk[154..155]$
- Compute $rk[214]$ from $rk[207]$ and $rk[182]$
- Compute $rk[189]$ from $rk[214]$ and $rk[219]$; then $rk[185]$ and $rk[181]$
- Compute $rk[213]$ from $rk[206]$ and $rk[181]$
- Compute $rk[188]$ from $rk[213]$ and $rk[220]$, then $rk[184]$ and $rk[180]$
- Compute $rk[212]$ from $rk[205]$ and $rk[180]$
- Compute $rk[219]$ from $rk[212]$ and $rk[187]$
- Compute $rk[208, 209]$ from $rk[215, 216]$ and $rk[183, 184]$
- We have $tk[216..219] = AESR\Big((rk[216..219])\Big)$ with a known key. Since $rk[216]$ and $rk[219]$ are known, we know that $tk[216..219]$ is a linear subspace of dimension 64 over $\mathbb{F}_2$.
- Similarly, $tk[208..211]$ is in a linear subspace of dimension 64 ($rk[208]$ and $rk[209]$ are known).

– Moreover, there are linear relations between $tk[216..219]$ and $tk[208..211]$: we can compute $rk[240..243]$ from $tk[208..211]$ and $rk[236..239]$; $rk[244..247]$ from $rk[240..243]$ and $tk[212..215]$; $tk[216..219]$ from $rk[244..247]$ and $rk[248..251]$.
– On average, we expect one solution for $tk[216..219]$ and $tk[208..211]$.
– At this point we have computed the values of $rk[200..223]$. We can compute $tk[200.223]$ and $rk[228..255]$.
– Compute $rk[176..179]$ from $rk[201..204]$ and $rk[208..211]$
– Since $rk[224..227] = rk[176..179]$, we have a full state $rk[224..255]$. We can check consistency of the initial guess.

9.4.4 9-round Attacks

The cancellation property allows to find a key/message pair with a given value on the last 128 bits. The attack is the following: first find a message that fulfills the conditions on the subkeys, and set a, b and c at round 4 satisfying the cancellation conditions (C0). Then the second output word is:

$$T_9 \oplus T_0 = X_9 \oplus X_0 = a \oplus F_4(b) \oplus b \oplus F_3(c) \oplus F'_1\big(c \oplus F_2(d \oplus F'_3(a))\big)$$

If we set

$$d = F_2^{-1}\Big(F_1'^{-1}\big(\overline{H} \oplus a \oplus F_4(b) \oplus b \oplus F_3(c)\big) \oplus c\Big) \oplus F'_3(a)$$

we have $X_9 \oplus X_0 = \overline{H}$. Each key (message) can be used with 2^{128} different a,b,c, and the cost of finding a suitable key is 2^{96}. Hence, the amortized cost for finding a 128-bit partial preimage is one compression function evaluation. The cost of finding a full preimage for the compression function is 2^{384}, as described in Algorithm 9.1.

Algorithm 9.1 *SHAvite-3*$_{512}$ cancellation attack on 9 rounds

Input: Target value $\overline{H}$
Output: A message M and a chaining value X s.t. $F(X, M) = \overline{H}$
Running Time: 2^{384}

```
1:  loop
2:      Find $M$ such that $(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6})$ and $(k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8})$
3:      $c \leftarrow F_7^{-1}(k^0_{1,6} \oplus k^0_{0,8})$
4:      for all $a$ do
5:          $b \leftarrow F_4^{-1}(F_5^{-1}(k^0_{1,4} \oplus k^0_{0,6}) \oplus a)$
6:          Compute $d$ as $F_2^{-1}\Big(F_1'^{-1}\big(\overline{H}_4 \oplus a \oplus F_4(b) \oplus b \oplus F_3(c)\big) \oplus c\Big) \oplus F'_3(a)$
7:          Compute 4 rounds backwards and 5 rounds forwards from $a$, $b$, $c$, $d$
8:          Then $H_4 = X_0 \oplus X_9 = \overline{H}_4$
9:          if $H = \overline{H}$ then
10:             return $X, M$
11:         end if
12:     end for
13: end loop
```

9.4.5 10-round Attacks

Note that we actually have two degrees of freedom when looking for a state satisfying (C0). Using both degrees of freedom we can extend the attack to 10 rounds. In order to better explain how to use the degrees of freedom, we use a basis including $c = T_3$ and $a \oplus F_4(b) = T_5$. In Table 9.11 we compute all the state words as a function of $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$. Note that we have:

$$a = w \oplus F_4(v) \qquad b = v \qquad c = z \qquad d = u \oplus F'_3(w \oplus F_4(v))$$

Table 9.11 Values of the Register for the 10-round Cancellation Property of *SHAvite-3*$_{512}$ Written as a function of $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$

i	X_i/Y_i
X_{-1}	$z \oplus F_2(u) \oplus F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z))))$
Y_{-1}	$w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)) \oplus F_0(v \oplus F_3(z) \oplus F'_1(z \oplus F_2(u)))$
X_0	$v \oplus F_3(z) \oplus F'_1(z \oplus F_2(u))$
Y_0	$u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))$
X_1	$w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z))$
Y_1	$z \oplus F_2(u)$
X_2	u
Y_2	$v \oplus F_3(z)$
X_3	z
Y_3	$w \oplus F_4(v)$
X_4	v
Y_4	$u \oplus F'_3(Y_3)$
X_5	w
Y_5	$z \oplus F'_4(Y_4)$
X_6	$Y_4 \oplus F_5(w)$
Y_6	$v \oplus F'_5(Y_5)$
X_7	$z \oplus \cancel{F'_4(\underline{Y_4})} \oplus \cancel{F_6(\underline{Y_4} \oplus F_5(w))}$
Y_7	$w \oplus F'_6(Y_6)$
X_8	$Y_6 \oplus F_7(z)$
Y_8	$Y_4 \oplus F_5(w) \oplus F'_7(Y_7)$
X_9	$w \oplus \cancel{F'_6(\underline{Y_6})} \oplus \cancel{F_8(\underline{Y_6} \oplus F_7(z))}$

The cancellation conditions are now written as:

Round 7 We have $F'_4(\underline{Y_4}) \oplus F_6(\underline{Y_4} \oplus F_5(w))$. They cancel if:
$F_5(w) = k^0_{1,4} \oplus k^0_{0,6}$ and $(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6})$.

Round 9 We have $F'_6(\underline{Y_6}) \oplus F_8(\underline{Y_6} \oplus F_7(z))$. They cancel if:
$F_7(z) = k^0_{1,6} \oplus k^0_{0,8}$ and $(k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8})$.

Therefore the truncated differential is followed if:

$$F_5(w) = k^0_{1,4} \oplus k^0_{0,6} \qquad F_7(z) = k^0_{1,6} \oplus k^0_{0,8} \qquad \text{(C0')}$$

$$(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6}) \qquad (k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8}) \qquad \text{(C1)}$$

Given a message that satisfies (C1) and a state that satisfies (C0'), we have $X_9 = w$ independently of u and v. The third output word after 10 rounds is equal to:

$$U_0 \oplus U_{10} = X_{-1} \oplus X_9 = z \oplus F_2(u) \oplus F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))) \oplus w$$

Given a target value $\overline{H}$, we try to solve $U_0 \oplus U_{10} = \overline{H}$:

$$\begin{aligned} U_0 \oplus U_{10} = \overline{H} &\iff z \oplus F_2(u) \oplus F_0'(u \oplus F_1(w \oplus F_4(v) \oplus F_2'(v \oplus F_3(z)))) \oplus w = \overline{H} \\ &\iff F_0'(u \oplus F_1(w \oplus F_4(v) \oplus F_2'(v \oplus F_3(z)))) = \overline{H} \oplus z \oplus F_2(u) \oplus w \\ &\iff F_1(w \oplus F_4(v) \oplus F_2'(v \oplus F_3(z))) = u \oplus F_0'^{-1}(\overline{H} \oplus z \oplus F_2(u) \oplus w) \qquad (9.1) \end{aligned}$$

Note that Equation (9.1) is split with v appearing only on the left-hand side, and u appearing only on the right-hand side. Any solution u, v to the equation gives a partial preimage on 128-bit of the compression function.

We can find solutions for Equation (9.1) using a collision-finding algorithm: if we compute the left-hand side with 2^{64} random values for v, and the right-hand side with 2^{64} random values for u, we expect to find one solution due to the birthday paradox with complexity 2^{64}. Note that this can be done with a memoryless collision-finding algorithm [vOW99]. This gives a partial preimage of 128 bits of *SHAvite-3*$_{512}$ for a cost of 2^{64} assuming that a suitable message is known.

To extend this partial preimage to a full preimage attack we use the fact that we can actually find 2^{128} solutions of Equation (9.1) for a cost of 2^{196}. Therefore, we repeat the following steps:

1. Find a suitable message (cost: 2^{96});
2. Find about 2^{128} solutions u and v using a memoryless collision-search algorithm (cost: 2^{192}).

This generates 2^{128} inputs of the compression function such that 128 bits of the output are chosen. We need to repeat these steps 2^{256} times to get a full preimage on the compression function reduced to 10 rounds with a cost of 2^{448} and negligible memory requirements.

Moreover, the complexity can be reduced at the cost of higher memory requirements. If we first build a table of the left-hand side of Equation (9.1) for all values of v, we can find a solution u, v in amortized time 1 by iterating over all u. More generally, we can make a trade-off between time and memory using a distinguished point based collision-finding algorithm, as given in [vOW99, Section 4.2]. Using 2^k bits of memory ($k \leq 128$) and 2^l processors, we can generate 2^{128} solutions with complexity $2^{192-k/2-l}$. Each solution gives an input to the compression function where 128 bits of the output are chosen. If we repeat this with 2^{256} different messages satisfying the constraints, we have a preimage attack on the compression function in time $2^{448-k/2-l}$. This gives a preimage attack on the hash function with time $2^{480-k/4-l/2}$ and memory $\max(2^{32+k/4+l/2}, 2^k)$.

Note that we can alternatively shift the characteristic by one round to get a partial preimage for the second output word $T_0 \oplus T_{10} = X_0 \oplus X_{10}$.

Second Preimage Attack on the Hash Function. We can use the preimage attack on the compression function to build a second preimage attack on the hash function reduced to 10 rounds. Using a generic unbalanced meet-in-the-middle attack the complexity is about 2^{497} compression function evaluations and 2^{16} memory. Note that we cannot find preimages for the hash function because we cannot find correctly padded message blocks.

9.5 Attacks on the Compression Function of *SHAvite-3*$_{512}$

In this section, we present preimage and collision attacks for the *SHAvite-3*$_{512}$ compression function reduced to 13 and to the full 14 rounds. We first give an outline of the attack and describe the characteristic used to find partial preimages. Then, we show how to use weak salts and counters in order to satisfy the conditions of this characteristic. Our attacks are chosen-salt attacks on the compression function, which are defined as s-Pre (enhanced preimage) attacks by Reyhanitabar *et al.* [RSM10]. Note that our attacks cannot be extended to the hash function, because of the way we have to choose the salt and counter value.

Table 9.12 Cancellation Property on 13 Rounds of *SHAvite-3*$_{512}$

i	S_i	T_i	U_i	V_i	
0	?	x_2	?	x	
1	x	-	x_2	x_1	
2	x_1	x	-	-	$x_1 \to x_2$
3	-	-	x	-	$x \to x_1$
4	-	-	-	x	
5	x	-	-	y	$\underline{x \to y}$
6	y	x	-	p	$y \to p$
7	p	$\underline{\text{-}}$	x	q	$\underline{x \to y},\ \underline{p \to q}$
8	q	p	$\underline{\text{-}}$	s	$q \to s \oplus x$
9	s	$\underline{\text{-}}$	p	t	$\underline{p \to q},\ \underline{s \to t}$
10	t	s	$\underline{\text{-}}$	u	$t \to u \oplus p$
11	u	$\underline{\text{-}}$	s	v	$\underline{s \to t},\ \underline{u \to v}$
12	v	u	$\underline{\text{-}}$	?	
13	?	$\underline{\text{-}}$	u	?	$\underline{u \to v}$
FF	?	x_2	?	?	

The attacks are based on the truncated differential of Table 9.12, using four cancellations. Again, we do not specify the values of y, p, q, s, t, u, or v, and the transitions $x \to x_1$ and $x_1 \to x_2$ are known because the subkeys are known.

In order to express the constraints needed for the cancellation properties, we look at the *values* of the registers for this truncated differential. In Table 9.13, we begin with $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$, and we compute up to round 13.

The conditions for the truncated differential are the following:

Round 7 We have $F'_4(\underline{Y_4}) \oplus F_6(\underline{Y_4} \oplus F_5(w))$. They cancel if:
$F_5(w) = k^0_{1,4} \oplus k^0_{0,6}$ and $(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6})$.

Round 9 We have $F'_6(\underline{Y_6}) \oplus F_8(\underline{Y_6} \oplus F_7(z))$. They cancel if:
$F_7(z) = k^0_{1,6} \oplus k^0_{0,8}$ and $(k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8})$.

Round 11 We have $F'_8(\underline{Y_8}) \oplus F_{10}(\underline{Y_8} \oplus F_9(w))$. They cancel if:
$F_9(w) = k^0_{1,8} \oplus k^0_{0,10}$ and $(k^1_{1,8}, k^2_{1,8}, k^3_{1,8}) = (k^1_{0,10}, k^2_{0,10}, k^3_{0,10})$.
Since w is fixed by the condition at round 7, the first condition can be written as:
$F_5^{-1}(k^0_{1,4} \oplus k^0_{0,6}) = F_9^{-1}(k^0_{1,8} \oplus k^0_{0,10})$

Round 13 We have $F'_{10}(\underline{Y_{10}}) \oplus F_{12}(\underline{Y_{10}} \oplus F_{11}(z))$. They cancel if:
$F_{11}(z) = k^0_{1,10} \oplus k^0_{0,12}$ and $(k^1_{1,10}, k^2_{1,10}, k^3_{1,10}) = (k^1_{0,12}, k^2_{0,12}, k^3_{0,12})$.
Since z is fixed by the condition at round 9, the first condition can be written as:
$F_7^{-1}(k^0_{1,6} \oplus k^0_{0,8}) = F_{11}^{-1}(k^0_{1,10} \oplus k^0_{0,12})$

In order to use this cancellation differential, we have a 256-bit condition on the state, and a 1792-bit condition on the expanded message:

$$F_5(w) = k^0_{1,4} \oplus k^0_{0,6} \qquad F_7(z) = k^0_{1,6} \oplus k^0_{0,8} \tag{C0'}$$

$$(k^1_{1,4}, k^2_{1,4}, k^3_{1,4}) = (k^1_{0,6}, k^2_{0,6}, k^3_{0,6}) \qquad (k^1_{1,6}, k^2_{1,6}, k^3_{1,6}) = (k^1_{0,8}, k^2_{0,8}, k^3_{0,8}) \tag{C1}$$

$$(k^1_{1,8}, k^2_{1,8}, k^3_{1,8}) = (k^1_{0,10}, k^2_{0,10}, k^3_{0,10}) \qquad (k^1_{1,10}, k^2_{1,10}, k^3_{1,10}) = (k^1_{0,12}, k^2_{0,12}, k^3_{0,12}) \tag{C2}$$

$$F_5^{-1}(k^0_{1,4} \oplus k^0_{0,6}) = F_9^{-1}(k^0_{1,8} \oplus k^0_{0,10}) \qquad F_7^{-1}(k^0_{1,6} \oplus k^0_{0,8}) = F_{11}^{-1}(k^0_{1,10} \oplus k^0_{0,12}) \tag{C3}$$

Table 9.13 Values of the Register for the 14-round Cancellation Property of *SHAvite-3*$_{512}$ Written as function of $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$

i	X_i/Y_i
X_{-1}	$z \oplus F_2(u) \oplus F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z))))$
Y_{-1}	$w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)) \oplus F_0(v \oplus F_3(z) \oplus F'_1(z \oplus F_2(u)))$
X_0	$v \oplus F_3(z) \oplus F'_1(z \oplus F_2(u))$
Y_0	$u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))$
X_1	$w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z))$
Y_1	$z \oplus F_2(u)$
X_2	u
Y_2	$v \oplus F_3(z)$
X_3	z
Y_3	$w \oplus F_4(v)$
X_4	v
Y_4	$u \oplus F'_3(Y_3)$
X_5	w
Y_5	$z \oplus F'_4(Y_4)$
X_6	$Y_4 \oplus F_5(w)$
Y_6	$v \oplus F'_5(Y_5)$
X_7	$z \oplus \cancel{F'_4(\underline{Y_4})} \oplus \cancel{F_6(\underline{Y_4} \oplus F_5(w))}$
Y_7	$w \oplus F'_6(Y_6)$
X_8	$Y_6 \oplus F_7(z)$
Y_8	$Y_4 \oplus F_5(w) \oplus F'_7(Y_7)$
X_9	$w \oplus \cancel{F'_6(\underline{Y_6})} \oplus \cancel{F_8(\underline{Y_6} \oplus F_7(z))}$
Y_9	$z \oplus F'_8(Y_8)$
X_{10}	$Y_8 \oplus F_9(w)$
Y_{10}	$Y_6 \oplus F_7(z) \oplus F'_9(Y_9)$
X_{11}	$z \oplus \cancel{F'_8(\underline{Y_8})} \oplus \cancel{F_{10}(\underline{Y_8} \oplus F_9(w))}$
Y_{11}	$w \oplus F'_{10}(Y_{10})$
X_{12}	$Y_{10} \oplus F_{11}(z)$
Y_{12}	$Y_8 \oplus F_9(w) \oplus F'_{11}(Y_{11})$
X_{13}	$w \oplus \cancel{F'_{10}(\underline{Y_{10}})} \oplus \cancel{F_{12}(\underline{Y_{10}} \oplus F_{11}(z))}$
Y_{13}	$z \oplus F'_{12}(Y_{12})$

If we can find messages satisfying conditions (C1) to (C3), the differential can be used to mount a 13-round attack similar to the 9-round attack, and a 14-round attack similar to the 10-round attack. However since there are only 1024 message bits, we do not expect to be able to find a solution for the 1792-bit condition.

9.5.1 Using Weak Salts

As shown by Peyrin in [Pey09], the first-round version of *SHAvite-3* had a weak salt for which it was possible to set all the subkeys to zero. Note that this particular set of subkeys satisfies conditions (C1) to (C3). In this section, we first show how to adapt this to the second-round version of *SHAvite-3*, and then how to build a family of weak salts and associated messages. Using these salts we will be able to use the cancellation differential of Table 9.11.

As explained in Appendix 9.D, the message expansion of *SHAvite-3*$_{512}$ is based on a succession of AES rounds with the salt used as key, and LFSR-based linear layers, plus a few additions of the counter. Since the non-linear layer is a single AES round with the salt used as key, there exists a particular salt so that the zero value is mapped to zero by the non-linear layer (this salt has all bytes equal to `0x52`). In the first-round version of *SHAvite-3*, using this salt and the zero counter, the zero message is expanded to the all zero subkeys.

For the second round of the SHA-3 competition, *SHAvite-3*$_{512}$ has been tweaked in order to prevent all the subkeys to be zero. Each time the counter is added to the subkeys, some counter words are inverted, so that the counter cannot be zero in every positions. However, we can still choose a particular counter so that the value added to the subkeys is zero at one position. For instance, with $(cnt_3, cnt_2, cnt_1, cnt_0) = (0, 0, \overline{0}, 0)$, the value $(cnt_2, cnt_3, cnt_0, \overline{cnt_1})$ added in round 5 of the key schedule is zero. If we use the salt `0x52` and zero subkeys at round 5, we can compute the subkeys starting from round 5 of the key expansion. We can see in Figure 9.5 page 221 that we have $rk[160 \dots 439] = 0$ because no counter interact with these values. Moreover, the effect of the counter at round 3 of the key expansion diffuses slowly and several other subkeys are still equal to zero.

Table 9.14 shows the set of subkeys that are equal to zero. By computing backwards we get the message which is given in Appendix 9.E. Using this message and the salt `0x52`, conditions (C1) to (C3) are satisfied, and we can apply the cancellation attack to the full compression function of the second-round version of *SHAvite-3*$_{512}$. Note that in contrast to the attack on the round 1 version, this results in a valid counter value.

Table 9.14 Subkeys in *SHAvite-3*$_{512}$ with a weak salt.
'0' denotes an all-zero subkey, and '?' denotes a subkey which is not zero.
The counter values are xor-ed prior to the subkeys marked by $\star$.

	RK_i				RK'_i				
i	$k^0_{0,i}$	$k^1_{0,i}$	$k^2_{0,i}$	$k^3_{0,i}$	$k^0_{1,i}$	$k^1_{1,i}$	$k^2_{1,i}$	$k^3_{1,i}$	r
0	?	?	?	?	?	?	?	?	M
1	?*	?	?	?	?	?	?	0	1
2	0	?	?	?	?	0	0	0	
3	0	?	?	?	0	0	0	0	2
4	0	?	0	0	0	0	0	0	
5	0	0*	0	0	0	0	0	0	3
6	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	4
8	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	0*	5
10	0	0	0	0	0	0	0	0	
11	0	0	0	0	0	0	0	0	6
12	0	0	0	0	0	0	0	0	
13	0	0	0	0	0	0	?*	?	7

A Family of Weak Salts. The weak salt `0x52` can be used for a partial preimage attack using the cancellation property, but this does not result in a full preimage or collision in the compression function because we cannot repeat the attack with a single weak salt. In this section we show how to use a class of weak salts.

Table 9.15 Subkeys in *SHAvite-3*$_{512}$ with periodic round keys RK_i and RK'_i.

	RK_i				RK'_i			
i	$k^0_{0,i}$	$k^1_{0,i}$	$k^2_{0,i}$	$k^3_{0,i}$	$k^0_{1,i}$	$k^1_{1,i}$	$k^2_{1,i}$	$k^3_{1,i}$
i	a	b	c	d	a	b	c	d
$i+1$	e	f	g	h	e	f	g	h
$i+2$	a	b	c	d	a	b	c	d

Instead of setting all the subkeys to be zero, we try to find subkeys which are periodic (see Table 9.15). Due to the nature of the key schedule, we will then get $RK_{i+2} = RK'_{i+2} = RK_i = RK'_i$ as long as the counter does not interfere. This will be enough to satisfy the conditions (C1) to (C3) and apply the cancellation attack.

We can find salts and messages giving such a periodic expanded message by solving a linear system of equations. The message expansion alternates linear and non-linear layers, where the non-linear layer is one AES round with the salt used as the key. Note that the 512-bit salt is used twice in the 1024-bit message expansion. Hence, we look for solutions where the left subkeys are equal to the right subkeys ($rk[i] = rk[i \oplus 16]$), and both halfs will still be equal after the non-linear layer ($tk[i] = tk[i \oplus 16]$). We construct a linear system with 1024 binary variables (or 32 32-bit variables), corresponding to the inputs and outputs of the non-linear layer. More precisely, we use variables $rk[256..271]$ and $tk[256..271]$. We can express $rk[320..351]$ as a linear function of $rk[256..271]$ and $tk[256..271]$, using the constraints $rk[272..287] = rk[256..271]$ and $tk[272..287] = tk[256..271]$. Then we solve the equations $rk[320..351] = rk[256..287]$. This gives a system of 32 linear equations with 32-bit variables.

Each solution to this system gives an input and output of the non-linear layer. They can easily be connected by computing the corresponding salt value. Then, we can compute the message expansion backwards to get the message yielding a periodic expanded message for the given salt.

Surprisingly, the system has many solutions, with a kernel of dimension 9, given by the following basis:

$$\begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&1
\end{bmatrix}$$

This means that we can generate $2^{9 \cdot 32} = 2^{288}$ different salt values and corresponding message blocks such that the conditions on the subkeys are fulfilled. An example of a random salt and message block in this class is given below:

```
Salt:    7f  51  e2  fb  ca  a5  95  ac  04  42  40  19  30  0f  17  82
         6d  31  01  30  86  30  0d  18  05  dc  db  90  96  2f  2a  78
         32  71  59  03  e3  bb  97  17  ee  41  bc  97  a3  b2  5c  18
         ce  af  fd  90  d6  8d  bf  fd  ab  11  9d  62  6a  11  13  b6
Message: e1  c1  44  35  67  84  1b  18  ca  ad  ac  f8  13  4d  ea  c9
```

```
b1  a0  79  d1  e7  a9  11  ca  75  eb  96  82  cc  81  50  4f
7a  69  43  38  8f  5d  e8  e4  e0  ce  3f  6b  55  1a  27  4e
8d  e6  2d  9a  63  76  78  73  67  10  8e  d2  38  02  45  90
12  16  0b  cc  6f  ab  c8  1a  ca  0e  c8  b7  5b  e7  33  93
58  87  01  5f  09  b3  64  c3  a9  a2  5a  15  c9  70  a8  cb
a0  80  ff  a8  c7  6d  24  60  09  8e  9d  15  0b  b1  af  8d
5c  37  11  26  17  df  d7  eb  9f  bc  f0  82  2d  ad  98  e0
```

9.5.2 13-round Attacks

Using a weak salt and the associated message, the conditions (C1) to (C3) are satisfied. If we start with a state $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$ such that z and w satisfy condition (C0'), we can compute the second output word:

$$T_0 \oplus T_{13} = X_0 \oplus X_{13} = v \oplus F_3(z) \oplus F'_1(z \oplus F_2(u)) \oplus w$$

For any value of u, we can compute v as:

$$v = \overline{H} \oplus F_3(z) \oplus F'_1(z \oplus F_2(u)) \oplus w$$

in order to have $X_0 \oplus X_{13} = \overline{H}$. Hence, we can construct a partial preimage on 128 bits for the compression function of *SHAvite-3*$_{512}$ reduced to 13 rounds, with a complexity of about one compression function evaluation. In Appendix 9.E, we give several partial preimage examples. Note that with a given salt and message, we can repeat this attack 2^{128} times by iterating over u.

To construct a collision and preimage for the whole output of the compression function, we need to repeat the partial preimage algorithm 2^{192} and 2^{384} times, respectively. Note that we can construct up to $2^{128+288} = 2^{416}$ partial preimages with the family of 2^{288} weak salts.

9.5.3 14-round Attacks

The technique used to extend the 9-round attack to a 10 rounds using both degrees of freedom can be used to extend the 13-round attack to 14 rounds.

We use a weak salt and message satisfying conditions (C1) to (C3), and we start with a state $(T_2, T_3, T_4, T_5) \triangleq (u, z, v, w)$ such that z and w satisfy condition (C0'). After 14 rounds, the third output word is equal to:

$$U_0 \oplus U_{14} = X_{-1} \oplus X_{13} = z \oplus F_2(u) \oplus F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))) \oplus w$$

Like in the 10-round attack, we have:

$$\begin{aligned} U_0 \oplus U_{14} = \overline{H} &\iff z \oplus F_2(u) \oplus F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))) \oplus w = \overline{H} \\ &\iff F'_0(u \oplus F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z)))) = \overline{H} \oplus z \oplus F_2(u) \oplus w \\ &\iff F_1(w \oplus F_4(v) \oplus F'_2(v \oplus F_3(z))) = u \oplus F'^{-1}_0(\overline{H} \oplus z \oplus F_2(u) \oplus w) \qquad (9.2) \end{aligned}$$

One simple way to find solutions for Equation (9.2) is to use a collision-finding algorithm, with complexity 2^{64} and no memory requirement. Moreover, the complexity can be reduced at the cost of higher memory requirements. If we first build a table of the left-hand side for all values of v, we can find a solution u, v in amortized time 1 by iterating over all u. More generally, we can make a trade-off between time and memory using a distinguished point based collision-finding algorithm, as given in [vOW99, Section 4.2]. Using 2^k bits of memory ($k \leq 128$) and 2^l processors, we can generate 2^{128} solutions with complexity $2^{192-k/2-l}$. Each solution gives an input to the compression function where 128 bits of the output are chosen. If we repeat this with several salts, we obtain the following attacks on the compression function:

– a collision attack in time $2^{256-k/2-l}$
– a preimage attack in time $2^{448-k/2-l}$

In Appendix 9.F we show how to find partial collisions on 128 bits of the *SHAvite-3*$_{512}$ compression function in practical time. This does not give a full collision attack on the compression function, but it is still a practical distinguisher of the compression function. Moreover, by implementing this attack we can validate the techniques used in this section.

9.6 Conclusion

In this chapter we explore new ways to use efficiently degree of freedom in generalized Feistel structures. In addition to the attacks on *Lesamnta* and *SHAvite-3*$_{512}$, we describe an attack against *SMS4* in Appendix 9.B. We summarize the obtained attacks in Tables 9.16, 9.17, and 9.18.

Table 9.16 Summary of the Attacks on *Lesamnta*

				Complexity	
Version	Attack		Rounds	Time	Memory
		Generic Attacks			
Lesamnta-256	Collision	[HKY08]	16	2^{97}	-
Lesamnta-256	Second Preimage	[HKY08]	16	2^{193}	-
Lesamnta-512	Collision	[HKY08]	16	2^{193}	-
Lesamnta-512	Second Preimage	[HKY08]	16	2^{385}	-
Lesamnta-256	Collision	(Sect. 9.3.3)	22	2^{96}	-
Lesamnta-256	Second Preimage	(Sect. 9.3.3)	22	2^{192}	-
Lesamnta-512	Collision	(Sect. 9.3.3)	22	2^{192}	-
Lesamnta-512	Second Preimage	(Sect. 9.3.3)	22	2^{384}	-
Lesamnta-256	Second Preimage	(Sect. 9.3.3)	24	2^{192}	2^{128}
Lesamnta-512	Second Preimage	(Sect. 9.3.3)	24	2^{384}	2^{256}
		Specific Attacks			
Lesamnta-256	Collision	(Sect. 9.3.4)	24	2^{120}	-
Lesamnta-256	Second Preimage	(Sect. 9.3.4)	24	2^{240}	-
Lesamnta-256	Collision	(Sect. 9.3.4)	24	2^{112}	-
Lesamnta-512	Collision	(Sect. 9.3.4)	24	2^{224}	-

9.A An Integral Attack against 21-Round *Lesamnta*

In this section, we show the applicability of the cancellation technique to block ciphers analysis by giving an application to the implicit block cipher of *Lesamnta*. The main difference with the hash function attacks is that in a block cipher the adversary does not know the key, and cannot build a message satisfying the cancellation conditions. Therefore, we use an attack based on integral cryptanalysis. The basic idea is to use several sets of messages, so that the messages in one of the sets follow the cancellation path.

Table 9.17 Summary of the Attacks on *SHAvite-3*$_{512}$

Attack		Rounds	Complexity	
			Time	Memory
Hash Function Attacks				
Second Preimage	[M+09]	8	2^{448}	2^{64}
Second Preimage	(Sect. 9.4.4)	9	2^{496}	2^{16}
Second Preimage	(Sect. 9.4.5)	10	2^{496}	2^{16}
Compression Function Attacks (chosen salt & counter)				
Collision	(Sect. 9.5.2)	13	2^{192}	-
Preimage	(Sect. 9.5.2)	13	2^{384}	-
Collision	(Sect. 9.5.3)	14	2^{192}	2^{128}
Preimage	(Sect. 9.5.3)	14	2^{384}	2^{128}
Preimage	(Sect. 9.5.3)	14	2^{448}	-

Table 9.18 Summary of the Attacks on *SMS4*

Attack		Rounds	Complexity	
			Data	Time
Boomerang	[KKHS08]	18	2^{120} ACPC	2^{120}
Rectangle	[KKHS08]	18	2^{124} CP	2^{124}
Differential	[ZZW08]	21	2^{118} CP	$2^{126.6}$
Differential	[KKHS08]	22	2^{118} CP	$2^{125.7}$
Linear	[KKHS08]	22	2^{117} KP	2^{117}
Truncated Differential	(Sect. 9.B.1)	19	2^{96} CP	2^{102}

In the original submission document of *Lesamnta* [HKY08] a 19-round Square distinguisher is described. This Square distinguisher is very straightforward, and suggests an efficient distinguisher for *Lesamnta*. However, by experimenting with reduced versions, we found that the original Square distinguisher is faulty. We first give an explanation of why the Square attack does not work. Then we suggest an improved and corrected 20-round integral distinguisher attack which relies on the cancellation property. This distinguisher gives a 21-round key-recovery attack using partial decryption of the last round. We use the term integral cryptanalysis rather than Square to describe our new attack, because we use a higher order property.

In Table 9.19, the symbols b_1, b_2, b_3 are used to denote three variables that independently take all possible values. So, in the first round, T_0, U_0, V_0 take all the $2^{3n/4}$ possible values. At round 1, we have $S_1 = F_1(U_0) \oplus V_0$. We see that $F_1(U_0) \oplus V_0, T_0, U_0$ take all possible values, so we can reuse the symbol b_3 for S_1. This can be seen as a change of variables.

Starting from round 4, we have two values denoted by b_3 in the original Square. This is used to denote that R_4, S_4, T_4 take all possible values, while S_4, T_4, U_4 also take all possible values. However, this leads to a contradiction later on because there is an implicit change of variables when we reuse symbols and this cannot be done for a variable that appears twice. The first problem appears at round 7. We have that S_6, T_6, V_6 and S_6, U_6, V_6 take all possible values. The original Square suggests that this implies that S_7, S_6, T_6 take all possible values, where $S_7 = F_7(U_6) \oplus V_6$. However this is not true in general. For instance, we could have $U_6 = F_7^{-1}(T_6 \oplus V_6)$. This is compatible with the assumptions of independence but in this case we have $S_7 = T_6$ and S_7, S_6, T_6 do not take all possible values.

Actually the Square property described in this attack can be detected after 18 rounds, but not after 19 rounds.

Table 9.19 The originally suggested Square, and its actual development. We see that the independence assumptions of round 7 do not hold.

i	S_i	T_i	U_i	V_i
0	-	b_1	b_2	b_3
1	b_3	-	b_1	b_2
2	b_2	b_3	-	b_1
3	b_1	b_2	b_3	-
4	b_3	b_1	b_2	b_3
5	b_3	b_3	b_1	b_2
6	b_2	b_3	b_3	b_1
7	b_1	b_2	b_3	b_3

Suggested in [HKY08]

i	V_i	U_i	T_i	S_i
0	-	b_1	b_2	b_3
1	b_3	-	b_1	b_2
2	b_2	b_3	-	b_1
3	b_1	b_2	b_3	-
4	$F(b_3)$	b_1	b_2	b_3
5	$F(b_2) \oplus b_3$	$F(b_3)$	b_1	b_2
6	$F(b_1) \oplus b_2$	$F(b_2) \oplus b_3$	$F(b_3)$	b_1
7	$F(F(b_3)) \oplus b_1$	$F(b_1) \oplus b_2$	$F(b_2) \oplus b_3$	$F(b_3)$

Actual Square

9.A.1 The New Attack

Our new attack is based on the cancellation path of Table 9.20. Starting with $(S_0, T_0, U_0, V_0) = (a, b, c, d)$, we have the following condition:

Round 8: we have $F_7(F_4(F_1(b) \oplus c) \oplus \underline{F_0(c) \oplus d)} \oplus F_3(\underline{F_0(c) \oplus d})$. They cancel if:
$F_4(F_1(b) \oplus c) = K_3 \oplus K_7$

Table 9.20 Cancellation path for the integral attack on *Lesamnta*

i	$X_i (= S_i)$
-3	d
-2	c
-1	b
0	a
1	$F_0(c) \oplus d$
2	$F_1(b) \oplus c$
3	$F_2(a) \oplus b$
4	$F_3(F_0(c) \oplus d) \oplus a$
5	$F_4(F_1(b) \oplus c) \oplus F_0(c) \oplus d$
6	$F_5(F_2(a) \oplus b) \oplus F_1(b) \oplus c$
7	$F_6(F_3(F_0(c) \oplus d) \oplus a) \oplus F_2(a) \oplus b$
8	$\cancel{F_7(F_4(F_1(b) \oplus c) \oplus \underline{F_0(c) \oplus d)}} \oplus \cancel{F_3(\underline{F_0(c) \oplus d})} \oplus a$

Since we do not know the value of $K_3 \oplus K_7$, we cannot build a message that would satisfy the cancellation condition with probability one. However, in an integral attack, if we iterate over all values of c while b is fixed, we know that one value will be such that there is a cancellation.

Moreover, for each c, we can iterate over d and study properties of the set of ciphertexts generated in this way.

More precisely, for a random choice of A and B, we define the following sets of messages:

$$\mathcal{S}_C = \{(A, B, C, d) : d \in \mathbb{F}_{2^{64}}\} \qquad (d \in \mathbb{F}_{2^{128}} \text{ for } \textit{Lesamnta}\text{-}512)$$

We consider 2^{64} sets (2^{128} for *Lesamnta*-512) with all possible values of C, while A and B are fixed. We know that one particular set satisfies $F_4(F_1(B) \oplus C) = K_3 \oplus K_7$. For this set, we show the dependencies of the state on d in Table 9.21. Note that:

- At round 8, the values $F_7(d) \oplus F_3(d)$ cancels out for this particular set.
- At round 14 we have $F_{13}\big(\underbrace{F_6(F_3(d))}\big) \oplus F_9\big(\underbrace{F_6(F_3(d))}\big)$. This has the special property that each value is taken an even number of times, according to Property (*ii*).
- At round 17, we have $F_{16}\Big(F_{13}\big(\underbrace{F_6(F_3(d))}\big) \oplus F_9\big(\underbrace{F_6(F_3(d))}\big)\Big) \oplus F_{12}\big(F_9(F_6(F_3(d)))\big) \oplus d$. When we sum this over all d's, this gives:

$$\bigoplus_d F_{16}\Big(F_{13}\big(\underbrace{F_6(F_3(d))}\big) \oplus F_9\big(\underbrace{F_6(F_3(d))}\big)\Big) \oplus \bigoplus_d F_{12}(F_9(F_6(F_3(d)))) \oplus \bigoplus_d d$$

The first term sums to zero because each input to F_{16} is taken an even number of times (cf. Property (*iii*)), and the two last terms sum to zero because they are permutations of d.

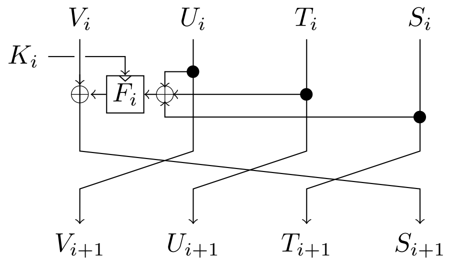
Since X_{17} is the fourth output word after 20 rounds ($X_{17} = V_{20}$), this gives an integral property on 20 rounds. This has been experimentally verified on reduced versions.

Table 9.21 Integral Attack. We only gives the dependencies in d.

i	X_i
-3	d
-2	-
-1	-
0	-
1	d
2	-
3	-
4	$F_3(d)$
5	d
6	-
7	$F_6(F_3(d))$
8	$\cancel{F_7(d)} \oplus \cancel{F_3(d)}$
9	d
10	$F_9(F_6(F_3(d)))$
11	$F_6(F_3(d))$
12	$F_{11}(d)$
13	$F_{12}(F_9(F_6(F_3(d)))) \oplus d$
14	$F_{13}\big(\underbrace{F_6(F_3(d))}\big) \oplus F_9\big(\underbrace{F_6(F_3(d))}\big)$
15	$F_{14}(F_{11}(d)) \oplus F_6(F_3(d))$
16	$F_{15}(F_{12}(F_9(F_6(F_3(d)))) \oplus d) \oplus F_{11}(d)$
17	$F_{16}(F_{13}\big(\underbrace{F_6(F_3(d))}\big) \oplus F_9\big(\underbrace{F_6(F_3(d))}\big)) \oplus F_{12}(F_9(F_6(F_3(d)))) \oplus d$

This property can be used to attack the block cipher of *Lesamnta*. One has to encipher the $2^{n/4}$ plaintexts in the sets $\mathcal{S}_C$ for each C, and to compute the sum of V_{20} over each set. If the

Figure 9.4 The Generalized Feistel structure of *SMS4*



SMS4 structure

data was generated using the compression function of *Lesamnta*, then at least one of the sets $\mathcal{S}_C$ gives a zero sum. Otherwise, there is only a probability $1/e$ that all the sums are non-zero. Moreover, when a set with a zero sum is found, it is possible to verify that this is due to the cancellation property, by building a new set $\mathcal{S}_C^*$ with the same C and B, but a different value A. As seen earlier, the cancellation condition does not depend on A, so this new set also gives a zero sum if C is the correct value for the cancellation condition.

This gives a distinguisher on 20-round *Lesamnta* with complexity $2^{n/2}$. Moreover, using partial decryption of the last round, it can be extended to a key recovery attack on 21 rounds with complexity $2^{3n/4}$.

9.B An Attack on 19-Round *SMS4*

SMS4 is a block cipher used in WAPI (the Chinese national standard for wireless networks) [Sw06, DL08], based on a generalized Feistel network. *SMS4* accepts a 128-bit plaintext and a 128-bit user key as inputs, and is composed of 32 rounds. In each round, the least significant three words of the state are XORed with the round key and the result passes the F transformation. The F transformation uses an 8-bit to 8-bit bijective SBox four times in parallel to process each byte, then the concatenated bytes are processed using a linear transformation L. The general structure is described by Figure 9.4, and can be written as:

$$S_{i+1} = V_i \oplus F(S_i \oplus T_i \oplus U_i \oplus K_i) \qquad T_{i+1} = S_i \qquad U_{i+1} = T_i \qquad V_{i+1} = U_i$$

where the K_i is the ith round subkey.

9.B.1 New Attack on *SMS4*

Our attack on *SMS4* is not based on the same cancellation property as used to attack *Lesamnta* and *SHAvite-3*$_{512}$. However, it shares the same core idea: we describe a generic attack on the Feistel structure based on a truncated differential, using available degrees of freedom to control the non-linearity. While this attack is not as efficient as the best attacks on *SMS4* (see Table 9.18), we believe it is interesting because it is generic in the round function.

Our attack is based on the following truncated differential on four rounds:

i	S_i	T_i	U_i	V_i		
0	x	x	a	b		$x = a \oplus b$
1	x	a	b	x		
2	a	b	x	x	$\underline{b \to u}$	let $c = a \oplus u$
3	b	x	x	c	$\underline{c \to u}$	let $d = b \oplus u$
4	x	x	c	d		$x = c \oplus d$

In this truncated differential, we do not specify the values of a, b, c, d and u. We start with any difference (x, x, a, b) with $x = a \oplus b$, and we end up with a difference of the same form: (x, x, c, d) with $x = c \oplus d$. The only condition for this truncated differential to be followed if that the transitions at step 2 and 3, $b \to u$ and $c \to u$ have to go to the same u. Since we do not care about the particular value of u, this happens with probability 2^{-32} (the words are 32-bit wide).

We can iterate this truncated differential four times, and add two extra rounds at the end. This gives the following characteristic:

$$(x, x, a, b)_{|a \oplus b = x} \xrightarrow{18 rounds} (e, f, x, x)_{|e \oplus f = x} \quad \text{with probability } 2^{-128}$$

For a random mapping, this characteristic has probability 2^{-96}. Therefore, to detect the bias, we need at least $2^{64+96} = 2^{160}$ input pairs: a random function would yield 2^{64} pairs following the differential, while 18-round *SMS4* will yield $2^{64} + 2^{32}$ pairs.

We can build the input pairs using structures: for a random A and B, we generate 2^{64} messages $M_{i,j} = (A \oplus i \oplus j, B \oplus i \oplus j, i, j)$, by varying i and j. This gives 2^{128} input pairs for the differential: each pair $M_{i,j}, M_{i',j'}$ has a difference $(i \oplus j \oplus i' \oplus j', i \oplus j \oplus i' \oplus j', i \oplus i', j \oplus j')$. If we repeat this with 2^{32} choices of A and B, we have enough pairs to detect the bias. This gives a distinguisher on 18 rounds of *SMS4* with 2^{96} chosen plaintexts. Experiments on reduced versions confirm this analysis.

This distinguisher can be used for a 19-round attack, using partial decryption of the last round. The corresponding differential is:

i	S_i	T_i	U_i	V_i	
0	x	x	a	b	$x = a \oplus b$
$\vdots$					
4	x	x	c	d	$x = c \oplus d$
$\vdots$					
16	x	x	e	f	$x = e \oplus f$
17	x	e	f	x	
18	e	f	x	x	
19	f	x	x	?	

We can recover the subkey of round 19 with the following algorithm:

1. Repeat 2^{40} times the following:
2. Choose a random A and B
3. Query the block cipher on the 2^{64} messages $M_{i,j} = (A \oplus i \oplus j, B \oplus i \oplus j, i, j)$
4. Let the plaintext/ciphertext be $(q_{i,j}, r_{i,j}, s_{i,j}, t_{i,j}) \to (\alpha_{i,j}, \beta_{i,j}, \gamma_{i,j}, \delta_{i,j})$, respectively. Store the plaintext/ciphertext pair of $((q_{i,j}, r_{i,j}, s_{i,j}, t_{i,j}), (\alpha_{i,j}, \beta_{i,j}, \gamma_{i,j}, \delta_{i,j}))$ in a hash table indexed by $q_{i,j} \oplus \alpha_{i,j}, q_{i,j} \oplus \beta_{i,j}$.

5. We search for collisions in the table, each offering a pair of plaintexts and ciphertexts $(q_{i,j}, r_{i,j}, s_{i,j}, t_{i,j}) \to (\alpha_{i,j}, \beta_{i,j}, \gamma_{i,j}, \delta_{i,j})$ and $(q_{i',j'}, r_{i',j'}, s_{i',j'}, t_{i',j'}) \to (\alpha_{i',j'}, \beta_{i',j'}, \gamma_{i',j'}, \delta_{i',j'})$ for which $q_{i,j} \oplus q_{i',j'} = \alpha_{i,j} \oplus \alpha_{i',j'} = \beta_{i,j} \oplus \beta_{i',j'} (= r_{i,j} \oplus r_{i',j'})$. This defines the difference x. There should be 2^{64} collisions on average.
6. For each of these pairs, obtain the input difference (f) to round 19, and the expected output difference $(e \oplus \delta_{i,j} \oplus \delta_{i',j'})$, and retrieve the one subkey suggestion (on average) for the subkey of round 19 which satisfies the differential transition [2].

The right subkey is expected to be suggested $2^{72} + 2^{40}$ times, while the wrong ones are expected to be suggested 2^{72} times. With very high probability (of more than 85%), the most suggested subkey is the correct one.

This gives a key-recovery attack on 19-round *SMS4* with 2^{104} chosen plaintexts, and a complexity of 2^{104} time.

9.C Implementation of the 24-round *Lesamnta* Attack

To verify our attacks, we implemented the attack on 24-round *Lesamnta* based on symmetry properties of the round function. We cannot find a full preimage because the complexity is too high, but we can show a partial preimage to prove the validity of our attack:

Chaining Value
`33212102 5c23803f 00957df0 94a1d777`
`4953d309 0b3b6624 8c8d523c b14eec82`

Message
`904fe6d0 cdb99073 1949261e de5b3575`
`70e209ed 1fe0a8d0 e7bc6031 6a88ceef`

Output
`03874543 a3a0eef7 3665a8bd 163bdaea`
`8a227d57 a6b6210f ffffffff ffffffff`

9.D *SHAvite-3*$_{512}$ Message Expansion

The message expansion of *SHAvite-3*$_{512}$ accepts a 1024-bit message block, a 128-bit counter, and a 512-bit salt. All are treated as arrays of 32-bit words (of 32, 4, and 16 words, respectively), which are used to generate 112 subkeys of 128 bits each, or a total of 448 32-bit words.

Let $rk[\cdot]$ be an array of 448 32-bit words whose first 32 words are initialized with the message. After the initialization of $rk[0, \ldots, 31]$, two processes are repeated, a nonlinear one (which generates 32 new words using the AES round function) and a linear one (which generates the next 32 words in a linear manner). These processes are repeated 6 times, and then the nonlinear process is repeated once more. The computation of $rk[\cdot]$ is done as follows:

Using the counter: the counter is used at 4 specific positions.

In order to simplify the description, we define a new table holding the preprocessed counter:

$ck[\ 32] = cnt[0]$,	$ck[\ 33] = cnt[1]$,	$ck[\ 34] = cnt[2]$,	$ck[\ 35] = \overline{cnt[3]}$
$ck[164] = cnt[3]$,	$ck[165] = cnt[2]$,	$ck[166] = cnt[1]$,	$ck[167] = \overline{cnt[0]}$
$ck[440] = cnt[1]$,	$ck[441] = cnt[0]$,	$ck[442] = cnt[3]$,	$ck[443] = \overline{cnt[2]}$
$ck[316] = cnt[2]$,	$ck[317] = cnt[3]$,	$ck[318] = cnt[0]$,	$ck[319] = \overline{cnt[1]}$

For all the other values, $ck[i] = 0$.

AES rounds: for $i \in \{0, 64, 128, 192, 256, 320, 384\} + \{0, 4, 8, 12, 16, 20, 24, 28\}$:

$$tk[(i, i+1, i+2, i+3)] = \mathrm{AESR}(rk[(i+1, i+2, i+3, i)] \oplus salt[(i, i+1, i+2, i+3) \bmod 16])$$

2. Note that we only need the differential table of the 8-bit S-Box for this step.

Linear Step 1: for $i \in \{32, 96, 160, 224, 288, 352, 416\} + \{0, \ldots, 31\}$:
$rk[i] = tk[i-32] \oplus rk[i-4] \oplus ck[i]$

Linear Step 2: for $i \in \{64, 128, 192, 256, 320, 384\} + \{0, \ldots, 31\}$:
$rk[i] = rk[i-32] \oplus rk[i-7]$

Figure 9.5 shows the succession of linear and non-linear steps in the full key schedule.

Once $rk[\cdot]$ is initialized, its 448 words are parsed as 112 words of 128-bit each, which are the subkeys (14 double quartets of 128-bit words each), *i.e.*:

$$\begin{aligned}
RK_{0,i} = (k^0_{0,i}, k^1_{0,i}, k^2_{0,i}, k^3_{0,i}) = \Big(&(rk[32\cdot i \quad], rk[32\cdot i + \ 1], rk[32\cdot i + \ 2], rk[32\cdot i + \ 3]),\\
&(rk[32\cdot i + \ 4], rk[32\cdot i + \ 5], rk[32\cdot i + \ 6], rk[32\cdot i + \ 7]),\\
&(rk[32\cdot i + \ 8], rk[32\cdot i + \ 9], rk[32\cdot i + 10], rk[32\cdot i + 11]),\\
&(rk[32\cdot i + 12], rk[32\cdot i + 13], rk[32\cdot i + 14], rk[32\cdot i + 15])\Big)\\
RK_{1,i} = (k^0_{1,i}, k^1_{1,i}, k^2_{1,i}, k^3_{1,i}) = \Big(&(rk[32\cdot i + 16], rk[32\cdot i + 17], rk[32\cdot i + 18], rk[32\cdot i + 19]),\\
&(rk[32\cdot i + 20], rk[32\cdot i + 21], rk[32\cdot i + 22], rk[32\cdot i + 23]),\\
&(rk[32\cdot i + 24], rk[32\cdot i + 25], rk[32\cdot i + 26], rk[32\cdot i + 27]),\\
&(rk[32\cdot i + 28], rk[32\cdot i + 29], rk[32\cdot i + 30], rk[32\cdot i + 31])\Big)
\end{aligned}$$

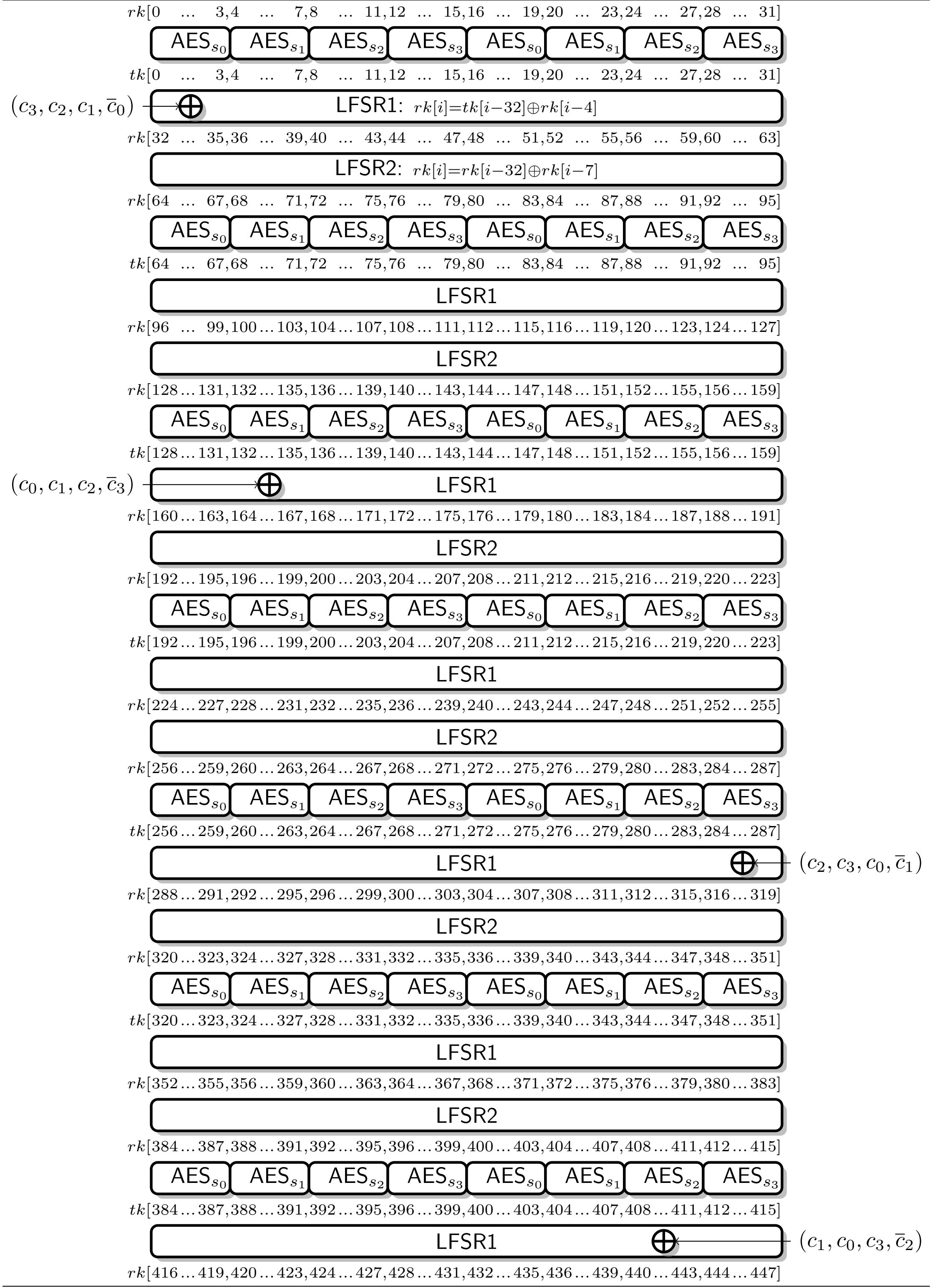
9.E Partial Preimages for 13-Round *SHAvite-3*$_{512}$

We give 3 examples for chaining inputs that all lead to a partial (128-bit) preimage of 0 for the compression function of *SHAvite-3*$_{512}$ reduced to 13 rounds. Note that the message block, the salt and counter values are the same in these examples.

```
counter:          00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00
salt:             52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
                  52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
                  52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
                  52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
message block M:  d1 58 6a 59 5f 1e ac c3 89 02 6a 23 8b 18 3d 35
                  a3 7b a6 8d 26 62 da 9a a6 8d 25 50 da 67 1e 62
                  0d fa 2b 8f a0 08 a4 97 b2 9b 25 0a 3e c3 6d c0
                  0b f7 12 3b d5 92 dd dc cf fa 79 ec 05 83 6e 9e
                  94 97 dd 03 4e e7 c1 07 8b f4 3d 9a df da 97 72
                  cc 24 50 90 0c 0a 0a b3 7c 58 d5 5d 7c 4d f9 ed
                  41 72 19 1a 8a ce 36 db ed fa 2e 40 23 66 8b d3
                  fa 1e 72 00 7b 8a 00 23 d3 00 49 88 00 96 79 19

chaining value 1: 73 ae 12 97 3f 8f 59 33 83 e5 b8 79 9f 39 3f d6
                  19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
                  3f e8 9e 31 8c 13 5b 51 05 f6 26 2f ab 50 d0 2f
                  7e d4 37 2c 7e b3 6f e2 a3 8c 10 c1 30 cb 43 1f
```

Figure 9.5 Key expansion of *SHAvite-3*$_{512}$


rk[0 … 3,4 … 7,8 … 11,12 … 15,16 … 19,20 … 23,24 … 27,28 … 31]
AES$_{s_0}$ AES$_{s_1}$ AES$_{s_2}$ AES$_{s_3}$ AES$_{s_0}$ AES$_{s_1}$ AES$_{s_2}$ AES$_{s_3}$
tk[0 … 3,4 … 7,8 … 11,12 … 15,16 … 19,20 … 23,24 … 27,28 … 31]
$(c_3, c_2, c_1, \bar{c}_0)$
LFSR1: $rk[i] = tk[i-32] \oplus rk[i-4]$
rk[32 … 35,36 … 39,40 … 43,44 … 47,48 … 51,52 … 55,56 … 59,60 … 63]
LFSR2: $rk[i] = rk[i-32] \oplus rk[i-7]$
rk[64 … 67,68 … 71,72 … 75,76 … 79,80 … 83,84 … 87,88 … 91,92 … 95]
tk[64 … 67,68 … 71,72 … 75,76 … 79,80 … 83,84 … 87,88 … 91,92 … 95]
LFSR1
rk[96 … 99,100 … 103,104 … 107,108 … 111,112 … 115,116 … 119,120 … 123,124 … 127]
LFSR2
rk[128 … 131,132 … 135,136 … 139,140 … 143,144 … 147,148 … 151,152 … 155,156 … 159]
tk[128 … 131,132 … 135,136 … 139,140 … 143,144 … 147,148 … 151,152 … 155,156 … 159]
$(c_0, c_1, c_2, \bar{c}_3)$
LFSR1
rk[160 … 163,164 … 167,168 … 171,172 … 175,176 … 179,180 … 183,184 … 187,188 … 191]
LFSR2
rk[192 … 195,196 … 199,200 … 203,204 … 207,208 … 211,212 … 215,216 … 219,220 … 223]
tk[192 … 195,196 … 199,200 … 203,204 … 207,208 … 211,212 … 215,216 … 219,220 … 223]
LFSR1
rk[224 … 227,228 … 231,232 … 235,236 … 239,240 … 243,244 … 247,248 … 251,252 … 255]
LFSR2
rk[256 … 259,260 … 263,264 … 267,268 … 271,272 … 275,276 … 279,280 … 283,284 … 287]
tk[256 … 259,260 … 263,264 … 267,268 … 271,272 … 275,276 … 279,280 … 283,284 … 287]
LFSR1
$(c_2, c_3, c_0, \bar{c}_1)$
rk[288 … 291,292 … 295,296 … 299,300 … 303,304 … 307,308 … 311,312 … 315,316 … 319]
LFSR2
rk[320 … 323,324 … 327,328 … 331,332 … 335,336 … 339,340 … 343,344 … 347,348 … 351]
tk[320 … 323,324 … 327,328 … 331,332 … 335,336 … 339,340 … 343,344 … 347,348 … 351]
LFSR1
rk[352 … 355,356 … 359,360 … 363,364 … 367,368 … 371,372 … 375,376 … 379,380 … 383]
LFSR2
rk[384 … 387,388 … 391,392 … 395,396 … 399,400 … 403,404 … 407,408 … 411,412 … 415]
tk[384 … 387,388 … 391,392 … 395,396 … 399,400 … 403,404 … 407,408 … 411,412 … 415]
LFSR1
$(c_1, c_0, c_3, \bar{c}_2)$
rk[416 … 419,420 … 423,424 … 427,428 … 431,432 … 435,436 … 439,440 … 443,444 … 447]

```
 output 1:          f5 bb 28 52 27 67 80 b5 8d 68 2d 1b 66 f2 0c 1e
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                    94 49 61 0d cc ed ea 7b 89 2a 90 ee e4 cc 49 0c
                    9c 3e 2e 17 78 f2 60 44 f5 f9 95 6c c0 dd 70 4f
 chaining value 2:  4d 96 cb 1f d7 26 9b f1 b8 84 e7 37 69 20 85 ee
                    19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
                    9c 0a 66 73 3f 9d 8e 4f 7d 15 85 71 6a cd fb 07
                    14 e6 c4 31 41 26 44 15 3a f8 a6 db b7 06 9a 4f
 output 2:          2a bf 6d c0 ef f7 78 b2 29 88 60 cc 04 63 22 6d
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                    8b 45 69 a9 7f 67 5e 20 e4 8d 9b 01 d6 74 a9 dd
                    d3 9c 37 d1 ae ed 12 4d 47 d1 7c 28 72 26 1e 97
chaining value 3:   b3 56 96 56 1a 43 91 1e 7b 0c 3f 99 9c f2 6b be
                    19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
                    70 04 fd 88 dd b1 28 f2 03 6a 04 9c c2 65 b4 7b
                    2c d9 e6 74 aa 0b c5 78 85 e0 0c 21 89 ba 7f 8e
output 3:           31 a4 76 86 fa 16 f4 41 7a 93 6b 68 33 2d 46 c9
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                    b1 a3 15 94 bc 41 2d fc 69 83 82 13 76 76 17 92
                    af 7e d8 93 c5 06 13 8e 05 2b 31 ab 65 cd 2a 51
```

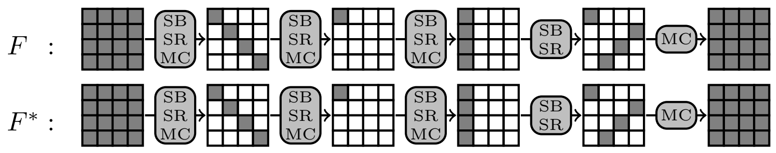
9.F Partial Collision Attack using a Rebound Technique

In this section we show a partial collision attack on the full compression function of *SHAvite-3*$_{512}$. As opposed to the attack of Section 9.5.3, this partial collision attack cannot be extended to a full collision attack, but it has the advantage of being practical. Thus we give an example of message pairs yielding a partial collision, which validates our attack strategy. This attack is based on an improvement of the rebound attack by Mendel *et al.* [MRST09]. We note that our improvement is of independent interest, and also leads to an improved collision attack on the Grøstl hash function.

Our attack uses the path of Table 9.13, like the 14-round attacks of Section 9.5.3. Using a weak salt and message such that the cancellation conditions are satisfied, we have: $X_{-1} \oplus X_{13} = F_2(u) \oplus F_0'(u \oplus \alpha) \oplus \beta$. Therefore a collision in $X_{-1} \oplus X_{13}$ is essentially a collision in $F \oplus F^*$ where F and F^* are 4 AES rounds with known keys. Such collisions can be found using the rebound attack with Super-Box cryptanalysis with a complexity of 2^{32} using 2^{32} memory [MRST10]. In this section we describe a new technique to find solutions with a complexity of 2^{32} using only 2^{16} memory, when the keys can be randomized (which is the case in our attack on *SHAvite-3*).

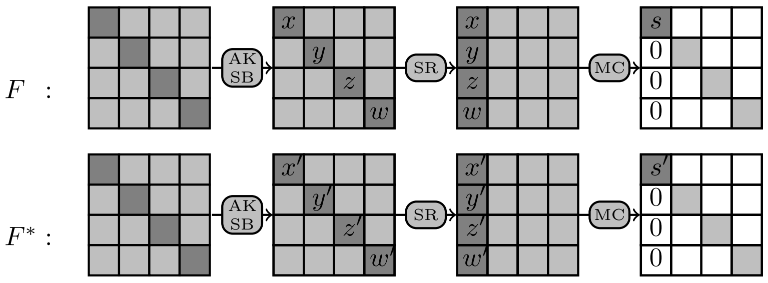
9.F.1 Finding Collisions in $F \oplus F^*$

We use the following differential trail in the AES rounds:



We start with a pair of inputs u, u' that differ in all bytes, so that the differences collapse to one active byte after two AES rounds. We will use the rebound attack to efficiently find such input pairs, with an amortized cost of 1. Then, the output difference before the last MixColumn has only 4 active bytes, so the output difference lies in a linear subspace of size 2^{32}.

First Round. We begin with the first AES round. The first diagonal is transformed into the first column:

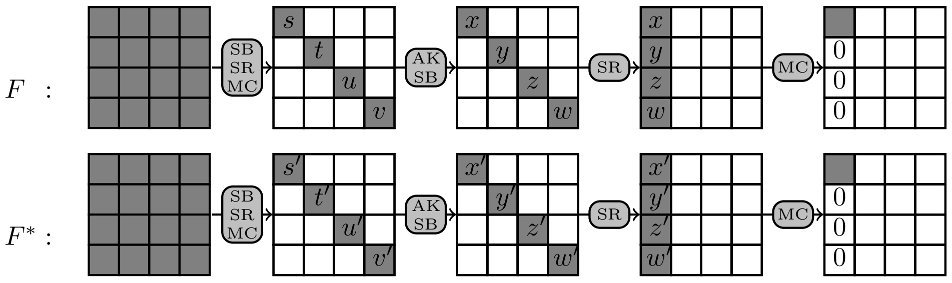


The variables inside the AES states correspond to differences.

The first step of the attack is to choose two arbitrary difference s and s'. Since the ShifRow and MixColumn operations are linear, we can compute x, y, z, w, from the first column $(s, 0, 0, 0)$ of the state at the end of the first round, and similarly in F^*. Then, given the *differences* x and x', we can compute the possible *values* for the first byte of the input state using a differential table for the pair of SBoxen with two keys in the middle. We do the same with y, y', z, z', and w, w' and we can find all the possible *values* for the first diagonal of the input. From the *values* of the input diagonal, we can compute the *values* of the column after one round.

Therefore, starting from the *differences* s and s', we can compute the *values* of the first column after one AES round. On average, we will get one solution for each choice of (s, s').

Second Round. We now study the second round of the path:



Given the *differences* for bytes t and t', we can efficiently compute the values that are possible in the first round using the same algorithm as in the previous paragraph. We will iterate over all possible *difference* pairs for bytes (t, t'), and compute the possibles *values* of (t, t'). Then we can compute the *values* of bytes (y, y') after the second SBox layer. We store those values in a table indexed by the *differences* for bytes (y, y'). We build a similar table for bytes (z, z') and (w, w').

Thanks to the first step, we know the *values* of byte s. Therefore, we can compute the values of byte x, and the corresponding differences. Because the MixColumn operation at the end is linear, the three zero differences in the state after the second round gives three linear equations between the *differences* for bytes x, y, z, w. Therefore, we can compute the *differences* for bytes y, z, w from x. Similarly, we compute the *differences* for bytes y', z', w' from x'. Now we can use the table built in the previous step to recover the *values* of bytes t, u, v, t', u', v' and the values of the full initial state.

On average we will get one solution for each choice of s, s'. Since there are 2^{16} such choices, we can amortize the cost of building the tables, and we have one solution for the differential path $F : 16 \to 8 \to 1 / F^* : 16 \to 8 \to 1$ for an amortized cost of one. This gives pairs of inputs to $F \oplus F^*$ so that the second output lies a linear subspace of size 2^{32}, which means we have a collision on 96 logical bits. This translates to pairs of inputs to the compression function of *SHAvite-3*$_{512}$, so that the second output lies a linear subspace of size 2^{32}.

Moreover, using several different keys, we can repeat this operation 2^{32} times, and we expect a collision on 128 output bits. We give an example of such a partial collision in Appendix 9.G.

9.F.2 Application to Grøstl

The same idea can be used to find collision in a version of Grøstl reduced to four rounds with complexity 2^{64} using 2^{16} memory. This is more efficient than the attack reported in [MRST10] with complexity 2^{64} using 2^{64} memory. Like in the *SHAvite-3*$_{512}$ case, we need to repeat the attack with random keys in order to get a collision in the full AES state, but we can do this using a random prefix block.

9.G Partial Collision for *SHAvite-3*$_{512}$

In this section, we give an example of a pair of chaining inputs that leads to a partial (128-bit) collision for the full compression function of *SHAvite-3*$_{512}$.

```
counter:
00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00

message block M:
d1 58 6a 59 5f 1e ac c3 89 02 6a 23 8b 18 3d 35
a3 7b a6 8d 26 62 da 9a a6 8d 25 50 da 67 1e 62
0d fa 2b 8f a0 08 a4 97 b2 9b 25 0a 3e c3 6d c0
0b f7 12 3b d5 92 dd dc cf fa 79 ec 05 83 6e 9e
94 97 dd 03 4e e7 c1 07 8b f4 3d 9a df da 97 72
cc 24 50 90 0c 0a 0a b3 7c 58 d5 5d 7c 4d f9 ed
41 72 19 1a 8a ce 36 db ed fa 2e 40 23 66 8b d3
fa 1e 72 00 7b 8a 00 23 d3 00 49 88 00 96 79 19

salt:
52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52
52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52

chaining value 1:
f1 7b 77 59 7b 44 ac 63 b4 78 d4 44 2b 30 24 9e
73 e8 12 b4 1d b4 9e f7 7d e6 99 0c 8d 8e a1 19
e6 ae 1f 7b ab 27 4a 45 44 2c 4c 4c 3e 01 0b 7e
c4 86 b9 df f5 e0 bd ab cf 99 09 bf 1e 56 9f 9a

output 1:
af 80 f0 5b 1d 41 44 34 be fc 4b 01 22 0d 82 f3
6a f1 0b ad 04 ad 87 ee 64 ff 80 15 94 97 b8 00
b4 15 c9 d3 bb 75 74 cb 87 0d 9d 6d ca b8 4f 73
ea a8 6f e0 68 9e d7 3d b6 3a b6 3f a0 fd 6d 96
```

```
chaining value 2:
47 5f bf f1 58 b2 47 8d f7 92 e1 f3 48 38 4d da
73 e8 12 b4 1d b4 9e f7 7d e6 99 0c 8d 8e a1 19
af 4f f8 a0 87 25 4b 2f 0c db 0e 19 39 cb 82 3e
01 86 5b ca b3 d1 39 67 ef 1d 91 93 47 f0 53 c7

output 2:
35 de 18 55 1c c6 00 26 df 6c 77 d9 92 f7 75 56
6a f1 0b ad 04 ad 87 ee 64 ff 80 15 94 97 b8 00
71 e9 96 ab 4c 38 f4 b2 8c 9e dd 8b 2e d6 f2 88
18 fa 29 77 bb d0 fe 4a 88 40 e3 1b 99 d7 88 96
```

Part Four

Appendix

Appendix A

Hash Function Zoo

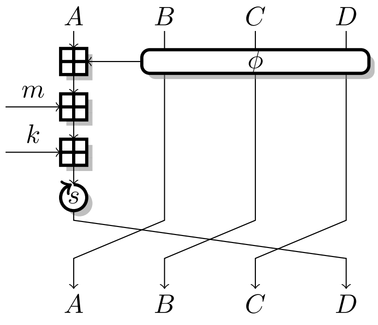
In this Appendix, we give a short description of the hash functions of the MD4 family, and other functions studied in this thesis. A quick overview of the members of the MD4 family is given in Table A.1.

Table A.1 Comparison of the hash functions of the MD4 family.
n is the output size
p is the state size
m is the size of the message blocks
l is the number of parallel lines in the compression function

Function	n	p	m	l	State	Steps
MD4	128	128	512	1	4×32 bits	48×1
MD5	128	128	512	1	4×32 bits	64×1
SHA-1	160	160	512	1	5×32 bits	80×1
SHA-224	224	256	512	2	8×32 bits	64×2
SHA-256	256	256	512	2	8×32 bits	64×2
SHA-384	384	512	1024	2	8×64 bits	80×2
SHA-512	512	512	1024	2	8×64 bits	80×2
SIMD-256	256	512	512	4	16×32 bits	36×4
SIMD-512	512	1024	1024	8	32×32 bits	36×8

A.1 MD4

Design: By Ron Rivest in 1990 [Riv90].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: $n = p = 128$, $m = 512$
Round function: 48 steps.



$$Q_i = \begin{pmatrix} Q_{i-4} \boxplus m_i \boxplus k_i \boxplus \\ \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \end{pmatrix}^{\lll s_i}$$

Step update:	$A \leftarrow (A \boxplus \Phi_i(D, C, B) \boxplus w_i \boxplus k_i)^{\lll s_i}$						
:	$(A, B, C, D) \leftarrow (B, C, D, A)$						
Input:	A	$\|$	D	$\|$	C	$\|$	B
Output:	$A \boxplus A_0$	$\|$	$D \boxplus D_0$	$\|$	$C \boxplus C_0$	$\|$	$B \boxplus B_0$

First round:	$0 \leq i < 16$	$\Phi_i =$ IF	$k_i = K_0$	$s_{i \bmod 4} = [\ 3,\ 7, 11, 19]$
Second round:	$16 \leq i < 32$	$\Phi_i =$ MAJ	$k_i = K_1$	$s_{i \bmod 4} = [\ 3,\ 5,\ 9, 13]$
Third round:	$32 \leq i < 48$	$\Phi_i =$ XOR	$k_i = K_2$	$s_{i \bmod 4} = [\ 3,\ 9, 11, 15]$

Message expansion: permutation based, $w_i = m_{\pi(i)}$.

π(0...15):	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
π(16...31):	0,	4,	8,	12,	1,	5,	9,	13,	2,	6,	10,	14,	3,	7,	11,	15
π(32...47):	0,	8,	4,	12,	2,	10,	6,	14,	1,	9,	5,	13,	3,	11,	7,	15

Alternative Description:

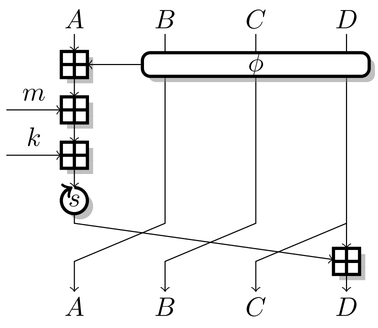
Step update:	$Q_i = (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i}$						
Input:	Q_{-4}	$\|$	Q_{-1}	$\|$	Q_{-2}	$\|$	Q_{-3}
Output:	$Q_{-4} \boxplus Q_{44}$	$\|$	$Q_{-1} \boxplus Q_{47}$	$\|$	$Q_{-2} \boxplus Q_{46}$	$\|$	$Q_{-3} \boxplus Q_{45}$

Main Weakness: Not enough rounds (wrt. weak message expansion).
Diffusion is too slow.

Main Attacks:	Data	Time	Memory	Ref
Collision	-	2^{1}	-	[WLF+05, SWOK07]
Preimage	-	2^{102}	2^{33}	[Leu08a]
HMAC key-recovery	2^{72}	2^{77}	-	[FLN07b, WOK08]

A.2 MD5

Design: By Ron Rivest in 1991 [RFC1321].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: $n = p = 128$, $m = 512$
Round function: 64 steps.



$$Q_i = \begin{pmatrix} Q_{i-4} \boxplus m_i \boxplus k_i \boxplus \\ \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \end{pmatrix}^{\lll s_i} \boxplus Q_{i-1}$$

Step update:	$A \leftarrow B \boxplus (A \boxplus \Phi_i(D, C, B) \boxplus w_i \boxplus k_i)^{\lll s_i}$						
:	$(A, B, C, D) \leftarrow (B, C, D, A)$						
Input:	A	$\|$	D	$\|$	C	$\|$	B
Output:	$A \boxplus A_0$	$\|$	$D \boxplus D_0$	$\|$	$C \boxplus C_0$	$\|$	$B \boxplus B_0$

First round:	$0 \leq i < 16$	$\Phi_i = \mathsf{IF}_x$	$s_{i \bmod 4} = [\ 7, 12, 17, 22]$
Second round:	$16 \leq i < 32$	$\Phi_i = \mathsf{IF}_z$	$s_{i \bmod 4} = [\ 5, \ 9, 14, 20]$
Third round:	$32 \leq i < 48$	$\Phi_i = \mathsf{XOR}$	$s_{i \bmod 4} = [\ 4, 11, 16, 23]$
Fourth round:	$32 \leq i < 48$	$\Phi_i = \mathsf{ONX}$	$s_{i \bmod 4} = [\ 6, 10, 15, 21]$

Constants: Each round has a unique constant: $k_i = 2^{32} \times |\sin(i)|$
Message expansion: permutation based, $w_i = m_{\pi(i)}$.

π(0...15):	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
π(16...31):	1,	6,	11,	0,	5,	10,	15,	4,	9,	14,	3,	8,	13,	2,	7,	12
π(32...47):	5,	8,	11,	14,	1,	4,	7,	10,	13,	0,	3,	6,	9,	12,	15,	2
π(48...63):	0,	7,	14,	5,	12,	3,	10,	1,	8,	15,	6,	13,	4,	11,	2,	9

Alternative Description:

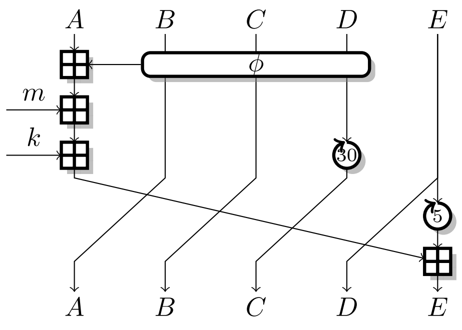
Step update:	$Q_i = Q_{i-1} \boxplus (Q_{i-4} \boxplus \Phi_i \boxplus m_{\pi(i)} \boxplus k_i)^{\lll s_i}$						
:	$\Phi_i = \phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3})$						
Input:	Q_{-4}	$\|$	Q_{-1}	$\|$	Q_{-2}	$\|$	Q_{-3}
Output:	$Q_{-4} \boxplus Q_{60}$	$\|$	$Q_{-1} \boxplus Q_{63}$	$\|$	$Q_{-2} \boxplus Q_{62}$	$\|$	$Q_{-3} \boxplus Q_{61}$

Main Weakness: flipping the MSB of the state words is likely to give a collision [dBB93].

Main Attacks:	Data	Time	Memory	Ref
Collision	-	2^{16}	-	[WY05, SSA+09]
Chosen-prefix collision	-	2^{49}	-	[SSA+09]

A.3 SHA-0

Design: By NIST and NSA in 1993 [Fips180].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: $n = p = 160$, $m = 512$
Round function: 80 steps.



$$Q_i = Q_{i-5}^{\lll 30} \boxplus m_i \boxplus k_i \boxplus \Phi_i(Q_{i-2}, Q_{i-3}^{\lll 30}, Q_{i-4}^{\lll 30}) \boxplus Q_{i-1}^{\lll 5}$$

Step update:	$A \leftarrow E^{\lll 5} \boxplus (A \boxplus \Phi_i(D, C, B) \boxplus w_i \boxplus k_i)$								
:	$(A, B, C, D, E) \leftarrow (B, C, D^{\lll 30}, E, A)$								
Input:	A	$\|\|$	E	$\|\|$	D	$\|\|$	C	$\|\|$	B
Output:	$A \boxplus A_0$	$\|\|$	$E \boxplus E_0$	$\|\|$	$D \boxplus D_0$	$\|\|$	$C \boxplus C_0$	$\|\|$	$B \boxplus B_0$

First round:	$0 \le i < 20$	$\Phi_i = \mathsf{IF}$	$k_i = K_0$
Second round:	$20 \le i < 40$	$\Phi_i = \mathsf{XOR}$	$k_i = K_1$
Third round:	$40 \le i < 60$	$\Phi_i = \mathsf{MAJ}$	$k_i = K_2$
Fourth round:	$60 \le i < 80$	$\Phi_i = \mathsf{XOR}$	$k_i = K_3$

Message expansion: linear code: $m_i = (m_{i-3} \oplus m_{i-8} \oplus m_{i-14} \oplus m_{i-16})$.
Alternative Description:

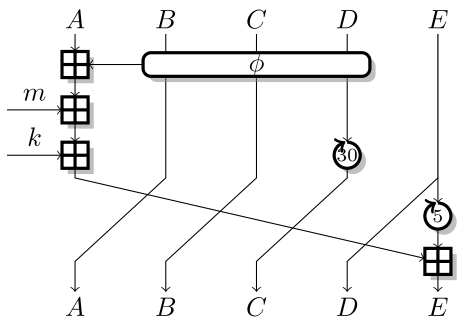
Step update:	$Q_i = Q_{i-5}^{\lll 30} \boxplus m_i \boxplus k_i \boxplus \Phi_i \boxplus Q_{i-1}^{\lll 5}$				
:	$\Phi_i = \phi_i(Q_{i-2}, Q_{i-3}^{\lll 30}, Q_{i-4}^{\lll 30})$				
Input:	$Q_{-5}^{\ggg 30}$	$\|\|$	Q_{-1}	$\|\|$	Q_{-2}
:		$\|\|$	$Q_{-3}^{\ggg 30}$	$\|\|$	$Q_{-4}^{\ggg 30}$
Output:	$Q_{-5}^{\ggg 30} \boxplus Q_{75}^{\ggg 30}$	$\|\|$	$Q_{-1} \boxplus Q_{79}$	$\|\|$	$Q_{-2} \boxplus Q_{78}$
:		$\|\|$	$Q_{-3}^{\ggg 30} \boxplus Q_{77}^{\ggg 30}$	$\|\|$	$Q_{-4}^{\ggg 30} \boxplus Q_{76}^{\ggg 30}$

Main Weakness: The code used in the message expansion does not mix the bits inside words. Moreover, the code is circulant, which allow to create local collisions.

Main Attacks:	Data	Time	Memory	Ref
Collision	-	2^{33}	-	[CJ98, WYY05c, MP08]

A.4 SHA-1

Design: By NIST and NSA in 1995 [Fips180-1].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: $n = p = 160$, $m = 512$
Round function: 80 steps.



$$Q_i = Q_{i-5}^{\lll 30} \boxplus m_i \boxplus k_i \boxplus \Phi_i(Q_{i-2}, Q_{i-3}^{\lll 30}, Q_{i-4}^{\lll 30}) \boxplus Q_{i-1}^{\lll 5}$$

Step update:	$A \leftarrow E^{\lll 5} \boxplus (A \boxplus \Phi_i(D, C, B) \boxplus w_i \boxplus k_i)$				
:	$(A, B, C, D, E) \leftarrow (B, C, D^{\lll 30}, E, A)$				
Input:	A	E	D	C	B
Output:	$A \boxplus A_0$	$E \boxplus E_0$	$D \boxplus D_0$	$C \boxplus C_0$	$B \boxplus B_0$

First round:	$0 \leq i < 20$	$\Phi_i = \mathsf{IF}$	$k_i = K_0$
Second round:	$20 \leq i < 40$	$\Phi_i = \mathsf{XOR}$	$k_i = K_1$
Third round:	$40 \leq i < 60$	$\Phi_i = \mathsf{MAJ}$	$k_i = K_2$
Fourth round:	$60 \leq i < 80$	$\Phi_i = \mathsf{XOR}$	$k_i = K_3$

Message expansion: linear code: $m_i = (m_{i-3} \oplus m_{i-8} \oplus m_{i-14} \oplus m_{i-16})^{\lll 1}$.
Alternative Description:

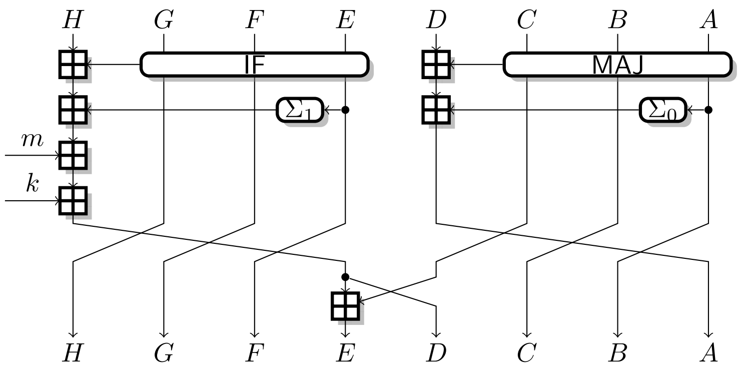
Step update:	$Q_i = Q_{i-5}^{\lll 30} \boxplus m_i \boxplus k_i \boxplus \Phi_i \boxplus Q_{i-1}^{\lll 5}$		
:	$\Phi_i = \phi_i(Q_{i-2}, Q_{i-3}^{\lll 30}, Q_{i-4}^{\lll 30})$		
Input:	$Q_{-5}^{\ggg 30}$	Q_{-1}	Q_{-2}
:		$Q_{-3}^{\ggg 30}$	$Q_{-4}^{\ggg 30}$
Output:	$Q_{-5}^{\ggg 30} \boxplus Q_{75}^{\ggg 30}$	$Q_{-1} \boxplus Q_{79}$	$Q_{-2} \boxplus Q_{78}$
:		$Q_{-3}^{\ggg 30} \boxplus Q_{77}^{\ggg 30}$	$Q_{-4}^{\ggg 30} \boxplus Q_{76}^{\ggg 30}$

Main Weakness: The code used in the message expansion is circulant. This allows to create local collisions.

Main Attacks:	Data	Time	Memory	Ref
Collision	-	2^{69}	-	[WYY05b]

A.5 SHA-2

Design: By NIST and NSA in 2001 [Fips180-2].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: SHA-224: $n = 224$, $p = 256$, $m = 512$; SHA-256: $n = 256$, $p = 256$, $m = 512$; SHA-384: $n = 384$, $p = 512$, $m = 1024$; SHA-512: $n = 512$, $p = 512$, $m = 1024$;
Round function (alternative description): SHA-256: 64 steps; SHA-512: 80 steps.



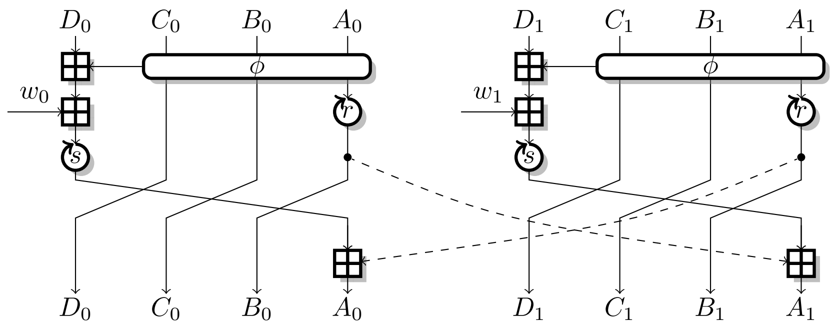
SHA-224/256:	$\Sigma_0(x) = x^{\lll 30} \oplus x^{\lll 19} \oplus x^{\lll 10}$
:	$\Sigma_1(x) = x^{\lll 26} \oplus x^{\lll 22} \oplus x^{\lll 7}$
SHA-384/512:	$\Sigma_0(x) = x^{\lll 34} \oplus x^{\lll 30} \oplus x^{\lll 25}$
:	$\Sigma_1(x) = x^{\lll 50} \oplus x^{\lll 46} \oplus x^{\lll 23}$

Message expansion: non-linear code: $m_i = \sigma_1(m_{i-2}) \boxplus m_{i-7} \boxplus \sigma_0(m_{i-15}) \boxplus m_{i-16}$

SHA-224/256:	$\sigma_0(x) = x^{\lll 25} \oplus x^{\lll 14} \oplus x^{\ggg 3}$
:	$\sigma_1(x) = x^{\lll 15} \oplus x^{\lll 13} \oplus x^{\ggg 10}$
SHA-384/512:	$\sigma_0(x) = x^{\lll 63} \oplus x^{\lll 56} \oplus x^{\ggg 7}$
:	$\sigma_1(x) = x^{\lll 45} \oplus x^{\lll\ 3} \oplus x^{\ggg 6}$

A.6 SIMD

Design: My SHA-3 candidate. Described in Part II.
Iteration mode: Wide-pipe Merkle-Damgård.
Compression function mode: Modified Davies-Meyer.
Sizes: SIMD-256: $n = 256$, $p = 512$, $m = 512$; SIMD-512: $n = 512$, $p = 1024$, $m = 1024$;
Round function: 32 steps.



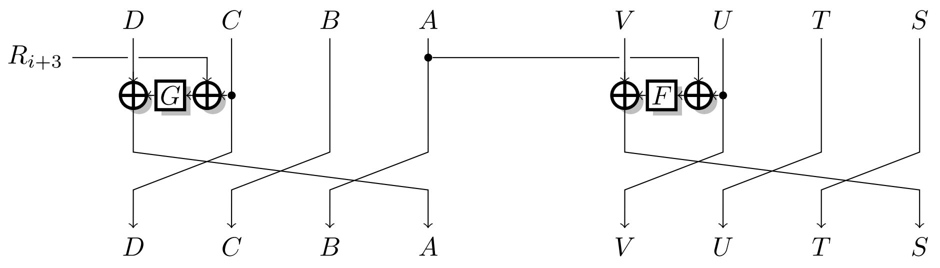
SIMD-256: 4 parallel Feistels.
SIMD-512: 8 parallel Feistels.

$$\begin{aligned} \text{Step update:} \quad & D_j \leftarrow \Big(D_j \boxplus W_j^{(i)} \boxplus \phi^{(i)}(A_j, B_j, C_j)\Big)^{\lll s^{(i)}} \boxplus A_{p^{(i)}(j)}{}^{\lll r^{(i)}} \\ : \quad & (A_j, B_j, C_j, D_j) \leftarrow (D_j, A_j^{\lll r^{(i)}}, B_j, C_j) \end{aligned}$$

Message expansion: code based: Reed-Solomon code, and multiplication by constants.

A.7 *Lesamnta*

Design: SHA-3 candidate by Hirose, Kuwakado, and Yoshida [HKY08].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Matyas-Meyer-Oseas.
Sizes: *Lesamnta*-256: $n = p = m = 256$; *Lesamnta*-512: $n = p = m = 512$;
Round function: 32 steps.



Step update:	$V = V \oplus F(U \oplus K_i)$			
:	$(S, T, U, V) \leftarrow (V, S, T, U)$			
Input:	$(A, B, C, D) \leftarrow$ CV		$(S, T, U, V) \leftarrow$ Msg	
Output:	$S \oplus S_0$ ‖	$T \oplus T_0$ ‖	$U \oplus U_0$ ‖	$V \oplus V_0$

Key expansion: Feistel based, similar to the round function:

Key derivation:	$D = D \oplus G(C \oplus R_{i+3})$
:	$(A, B, C, D) \leftarrow (D, A, B, C)$
Round constants:	$R_i = (2i, 2i + 1)$

***F* and *G* functions:** AES-based.
Lesamnta-256 F : ▦ 4 unkeyed AES-like rounds; G : ▦ 1 unkeyed AES-like round.
Lesamnta-512 F : ▦ 4 unkeyed AES rounds; G : ▦ 1 unkeyed AES-like round.
Alternative Description:

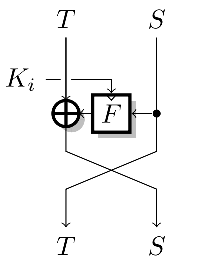
Step update:	$X_{i+4} = X_i \oplus F(X_{i+1} \oplus K_{i+3})$			
Key derivation:	$K_{i+4} = K_i \oplus G(K_{i+1} \oplus R_{i+3})$			
Output:	$X_{-3} \oplus X_{29}$ ‖	$X_{-2} \oplus X_{30}$ ‖	$X_{-1} \oplus X_{31}$ ‖	$X_0 \oplus X_{32}$

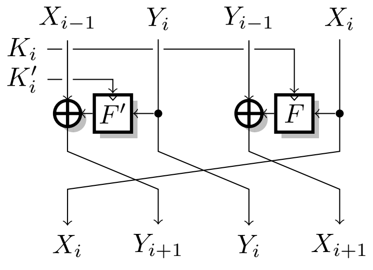
Main Weakness: Weak constants. Slow diffusion.
Tweak: New round constants to avoid the self-similarity attack [HKY09].

Main Attacks:	Time	Memory	Ref
Herding	$2^{n/2}$	-	[BDFL10, 8.3.3]
Compression Function Collision	$2^{n/4}$	-	[BDFL10, 8.3.2]
24-round *Lesamnta*-256 (tweaked) Collision	2^{112}	-	[BDLF09, 9.3.3]
24-round *Lesamnta*-256 (tweaked) 2nd Preimage	2^{240}	-	[BDLF09, 9.3.3]
24-round *Lesamnta*-512 (tweaked) Collision	2^{224}	-	[BDLF09, 9.3.3]

A.8 *SHAvite-3*

Design: SHA-3 candidate by Biham and Dunkelman [BD08].
Iteration mode: HAIFA (Narrow-pipe Merkle-Damgård with counter and salt).
Compression function mode: Davies-Meyer.
Sizes: *SHAvite-3*$_{256}$: $n = p = 256$, $m = 512$, $s = 256$, $c = 64$; *SHAvite-3*$_{512}$: $n = p = 512$, $m = 1024$, $s = 512$, $c = 128$.
Round function: *SHAvite-3*$_{256}$: 12 steps; *SHAvite-3*$_{512}$: 14 steps.





SHAvite-3$_{256}$ *SHAvite-3*$_{512}$

***F* and *F′* functions:** AES-based.
Lesamnta-256 3 keyed AES rounds; *Lesamnta*-512 4 keyed AES rounds.
Lesamnta-256: $F(k^0, k^1, k^2, x) = P(k^2 \oplus P(k^1 \oplus P(k^0 \oplus x)))$.
Lesamnta-512: $F(k^0, k^1, k^2, k^3, x) = P(k^3 \oplus P(k^2 \oplus P(k^1 \oplus P(k^0 \oplus x))))$.
Message expansion: Mix of AES-rounds and LFSR.
If we disregard the counter, defined as:
AES rounds: for $i \in \{0, 64, 128, 192, 256, 320, 384\} + \{0, 4, 8, 12, 16, 20, 24, 28\}$:

$tk[(i, i+1, i+2, i+3)] = \mathrm{AESR}(rk[(i+1, i+2, i+3, i)] \oplus salt[(i, i+1, i+2, i+3) \bmod 16])$

Linear Step 1: for $i \in \{32, 96, 160, 224, 288, 352, 416\} + \{0, \ldots, 31\}$:

$rk[i] = tk[i-32] \oplus rk[i-4]$

Linear Step 2: for $i \in \{64, 128, 192, 256, 320, 384\} + \{0, \ldots, 31\}$:

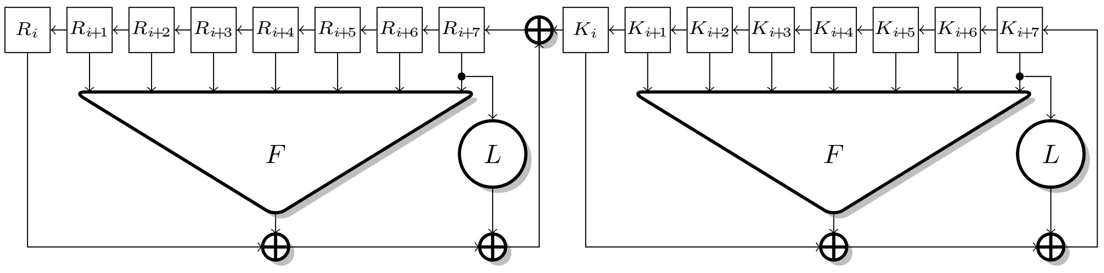
$rk[i] = rk[i-32] \oplus rk[i-7]$

Main Weakness: Relatively slow diffusion for *SHAvite-3*$_{512}$.
Chosen salt properties of the message expansion

Main Attacks:	Time	Memory	Ref
10-round *SHAvite-3*$_{512}$ 2nd Preimage	2^{496}	2^{16}	[GLM+10, 9.4.5]
SHAvite-3$_{512}$ Chosen-salt Pseudo-preimage	2^{448}	-	[GLM+10, 9.5.3]

A.9 *ESSENCE*

Design: SHA-3 candidate by Martin [Mar08].
Iteration mode: Narrow-pipe Merkle-Damgård.
Compression function mode: Davies-Meyer.
Sizes: *ESSENCE*-256: $n = p = m = 256$; *ESSENCE*-512: $n = p = m = 512$.
Round function: 32 steps.



Step update:	$R_{i+8} = R_i \oplus F\,(R_{i+1}, R_{i+2}, \ldots, R_{i+7}) \oplus L(R_{i+7}) \oplus K_i$
Key derivation:	$K_{i+8} = K_i \oplus F\,(K_{i+1}, K_{i+2}, \ldots, K_{i+7}) \oplus L(K_{i+7})$

Message expansion: LFSR based, similar to the round function.
***F* and *L* functions:** F is bitwise; L is linear, built from a LFSR.
Main Weakness: No round constants. Slow diffusion.

Main Attacks:	Time	Memory	Ref
ESSENCE-256 Collision	2^{68}	-	[NPRA+10]
ESSENCE-512 Collision	2^{135}	-	[NPRA+10]

Contents

List of Figures

List of Tables

List of Algorithms

List of Differential Paths

Bibliography

[ABM$^+$09] Jean-Philippe Aumasson, Eric Brier, Willi Meier, María Naya-Plasencia, and Thomas Peyrin. Inside the Hypercube. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 5594 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2009. 7.1, 7.2, 7.2.4, 7.3.1, 7.5.1, 8.2.2

[ABMM09] Jean-Philippe Aumasson, Tor E. Bjørstad, Willi Meier, and Florian Mendel. Observation on the PRE-MIXING step of CHI-256 and CHI-224. OFFICIAL COMMENT, 2009. 8.1, 8.2

[AH88] Hamid R. Amirazizi and Martin E. Hellman. Time-memory-processor trade-offs. *IEEE Transactions on Information Theory*, 34(3):505–512, 1988. 4.2

[AMM09] Jean-Philippe Aumasson, Atefeh Mashatan, and Willi Meier. More on Shabal's permutation. OFFICIAL COMMENT, 2009. 7.3.1, 7.5.1

[AS08] Kazumaro Aoki and Yu Sasaki. Preimage Attacks on One-Block MD4, 63-Step MD5 and More. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2008. 5.1.2

[BB02] Elad Barkan and Eli Biham. In How Many Ways Can You Write Rijndael? In Yuliang Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002. 8.2, 8.2

[BC04] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Franklin [Fra04], pages 290–305. 3.3.2

[BCCM$^+$09] Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-Francois Misarsky, María Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, and Marion Videau. Indifferentiability with Distinguishers: Why Shabal Does Not Require Ideal Ciphers. Cryptology ePrint Archive, Report 2009/199, 2009. `http://eprint.iacr.org/`. 7.3, 7.3.1, 7.5.1

[BCH06] John Black, Martin Cochran, and Trevor Highland. A Study of the MD5 Attacks: Insights and Improvements. In Robshaw [Rob06], pages 262–277. 3.1, 3.4.2

[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], pages 36–57. 1.4.1, 3.3.1

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Koblitz [Kob96], pages 1–15. 1.2.2, 4.1, 4.2, 4.2

[BCK96b] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security. In *FOCS*, pages 514–523, 1996. 6.5.1

[BD07] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Nist Hash workshop 2006, Santa Barbara – Cryptology ePrint Archive, Report 2007/278, 2007. `http://eprint.iacr.org/`. 1.3.4

[BD08] Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function. Submission to NIST, 2008. 9.4, A.8

[BD09] Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function (Tweaked Version). Submission to NIST, 2009. 2.3.2

[BDFL10] Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, and Gaëtan Leurent. Another Look at the Complementation Property. In Hong and Iwata [HI10]. 2.3.1, 2.3.4, 2.5, 7.1, 7.2, 7.2.4, 7.3.1, 7.3.2, 9.3.2, A.7

[BDLF09] Charles Bouillaguet, Orr Dunkelman, Gaëtan Leurent, and Pierre-Alain Fouque. Attacks on Hash Functions based on Generalized Feistel - Application to Reduced-Round Lesamnta and SHAvite-3-512. Cryptology ePrint Archive, Report 2009/634, 2009. `http://eprint.iacr.org/`. 2.3.1, 2.3.2, A.7

[BDLF10] Charles Bouillaguet, Orr Dunkelman, Gaëtan Leurent, and Pierre-Alain Fouque. Attacks on Hash Functions based on Generalized Feistel - Application to Reduced-Round Lesamnta and SHAvite-3-512. In Biryukov et al. [BGS10]. 2.5

[BDPA08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Smart [Sma08], pages 181–197. 1.3.2

[Bel06] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006. 4.1, 4.2, 4.A, 6.5.3

[BFL10a] Charles Bouillaguet, Pierre-Alain Fouque, and Gaëtan Leurent. Security Analysis of SIMD. Cryptology ePrint Archive, Report 2010/323, 2010. `http://eprint.iacr.org/`. 2.2.2

[BFL10b] Charles Bouillaguet, Pierre-Alain Fouque, and Gaëtan Leurent. Security Analysis of SIMD. In Biryukov et al. [BGS10]. 2.5

[BGS10] Alex Biryukov, Guang Gong, and Douglas Stinson, editors. *Selected Areas in Cryptography, 17th Annual International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2010. A.9

[Bih93] Eli Biham. New types of cryptoanalytic attacks using related keys (extended abstract). In *EUROCRYPT*, pages 398–409, 1993. 8.2, 8.2.1

[Bih94] Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. *J. Cryptology*, 7(4):229–246, 1994. 8.2

[Bir07] Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*. Springer, 2007. A.9

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In Matsui [Mat09], pages 1–18. 6.1.1, 7.1

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In Halevi [Hal09], pages 231–249. 2.2.2, 6.2.3

[BR93] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993. 1.2.2

[BR94] Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption. In *EUROCRYPT*, pages 92–111, 1994. 1.2.2

[BR96] Mihir Bellare and Phillip Rogaway. The Exact Security of Digital Signatures - HOw to Sign with RSA and Rabin. In *EUROCRYPT*, pages 399–416, 1996. 1.2.2

[Bra90] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990. A.9

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002. 6.2.2, 6.5.1

[BW99] Alex Biryukov and David Wagner. Slide Attacks. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999. 8.2

[BY03] Mihir Bellare and Bennet S. Yee. Forward-Security in Private-Key Cryptography. In Marc Joye, editor, *CT-RSA*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003. 1.2.2

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Shoup [Sho05], pages 430–448. 1.3.3, 1.3.4, 6.2.2, 6.2.3, 6.5.1, 7.1.1, 7.3, 7.B

[CJ98] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998. 1.4.1, I, 3.3.1, 7.4.4, A.3

[CJ04] Jean-Sébastien Coron and Antoine Joux. Cryptanalysis of a Provably Secure Cryptographic Hash Function. Cryptology ePrint Archive, Report 2004/013, 2004. `http://eprint.iacr.org/`. 2.3.5

[CN08] Donghoon Chang and Mridul Nandi. Improved Indifferentiability Security Analysis of chopMD Hash Function. In Nyberg [Nyb08], pages 429–443. 6.5.1

[CR06] Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Lai and Chen [LC06], pages 1–20. 3.1, 3.5, 3.5.5

[Cra05] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005. A.9

[CY06] Scott Contini and Yiqun Lisa Yin. Forgery and Partial Key-Recovery Attacks on HMAC and NMAC Using Hash Collisions. In Lai and Chen [LC06], pages 37–53. 3.1.1, 3.5.6, 4.1, 4.2, 4.2, 4.3, 4.3.1, 4.3.3, 4.3.3, 4.3.4, 4.5, 4.5.1

[Dam89] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [Bra90], pages 416–427. 1.3.1

[Dau05] M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family*. PhD thesis, Ruhr-University of Bochum, 2005. 3.5.1

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the Compressin Function of MD5. In *EUROCRYPT*, pages 293–304, 1993. 1.3.2, 1.4.1, I, 3.2.2, 3.3.1, 3.5, 4.2, 4.3.1, 4.3.3, 4.3.4, 4.5.1, 4.6.3, 6.2.2, 6.5.2, 8.1, 8.2, A.2

[Dea99] Richard D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, 1999. 1.3.4, 5.3.2, 6.2.2, 6.2.3

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976. 1.1.1

[DL05] Magnus Daum and Stefan Lucks. Hash Collisions (The Poisoned Message Attack) "The Story of Alice and her Boss". Presented at the rump session of Eurocrypt '05, 2005. `http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/`. 3.1.1

[DL08] Whitfield Diffie and George Ledin. SMS4 Encryption Algorithm for Wireless Networks. Cryptology ePrint Archive, Report 2008/329, 2008. English translation of [Sw06], `http://eprint.iacr.org/`. 9.B

[Dob96a] Hans Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996. 1.4.1, I

[Dob96b] Hans Dobbertin. The status of MD5 after a recent attack. *CryptoBytes*, 2(2):1, 3–6, Summer 1996. 1.4.1, I

[Dob98a] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998. 3.4.1, 3.1, 5.1.1

[Dob98b] Hans Dobbertin. The First Two Rounds of MD4 are Not One-Way. In Serge Vaudenay, editor, *FSE*, volume 1372 of *Lecture Notes in Computer Science*, pages 284–292. Springer, 1998. 5.1.1, 5.2.1, 5.2.1, 5.3.2

[DR98] J. Daemen and V. Rijmen. AES proposal: Rijndael. NIST AES Proposal, June 1998. 1.1.1, 6.2.3

[eBACS] eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`. 6.3.1

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. 1.1.1

[FGS07] Matthieu Finiasz, Philippe Gaborit, and Nicolas Sendrier. Improved Fast Syndrome Based Cryptographic Hash Functions. In Vincent Rijmen, editor, *ECRYPT Hash Workshop 2007*, 2007. 2.3.5

[Fips180] National Institute of Standards and Technology. *FIPS PUB 180: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, May 1993. A.3

[Fips180-1] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180. A.4

[Fips180-2] National Institute of Standards and Technology. *FIPS PUB 180-2: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, August 2002. Supersedes FIPS PUB 180-1. A.5

[Fips197] National Institute of Standards and Technology. *FIPS PUB 197: Advanced Encryption Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, November 2001. 1.1.1

[FL08] Pierre-Alain Fouque and Gaëtan Leurent. Cryptanalysis of a Hash Function Based on Quasi-cyclic Codes. In Malkin [Mal08], pages 19–35. 2.3.5, 2.5

[FLM10] Niels Ferguson, Stefan Lucks, and Kerry A. McKay. Symmetric States and their Structure: Improved Analysis of CubeHash. Cryptology ePrint Archive, Report 2010/273, 2010. `http://eprint.iacr.org/`. 7.1, 7.2, 7.2.4, 7.3.1, 7.5.1, 8.2.2

[FLN07a] Pierre-Alain Fouque, Gaëtan Leurent, and Phong Nguyen. Automatic Search of Differential Path in MD4. ECRYPT Hash Worshop 2007, Barcelona – Cryptology ePrint Archive, Report 2007/206, 2007. `http://eprint.iacr.org/`. 2.5, 3.1, 3.2, 5.2.1

[FLN07b] Pierre-Alain Fouque, Gaëtan Leurent, and Phong Q. Nguyen. Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Menezes [Men07], pages 13–30. 2.1.3, 2.5, 3.1, 3.1.1, 3.3.1, 3.5, 4.2, A.1

[FLRV09] Pierre-Alain Fouque, Gaëtan Leurent, Denis Réal, and Frédéric Valette. Practical Electromagnetic Template Attack on HMAC. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009. 2.4.1, 2.5

[Fra04] Matthew K. Franklin, editor. *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*. Springer, 2004. A.9

[GB09] Praveen Gauravaram and Nasour Bagheri. Private communication, July 2009. 6.2.3, 7.1.1

[GIS06] Max Gebhardt, Georg Illies, and Werner Schindler. A Note on the Practical Value of Single Hash Collisions for Special File Formats. In Jana Dittmann, editor, *Sicherheit*, volume 77 of *LNI*, pages 333–344. GI, 2006. 3.1.1

[GLM$^+$10] Praveen Gauravaram, Gaëtan Leurent, Florian Mendel, María Naya-Plasencia, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Cryptanalysis of the 10-Round Hash and Full Compression Function of SHAvite-3-512. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 419–436. Springer, 2010. 2.3.2, 2.5, A.8

[GØM$^+$08] Danilo Gligoroski, Rune Steinsmo Ødegård, Marija Mihova, Svein Johan Knapskog, Ljupco Kocarev, Aleš Drápal, and Vlastimil Klima. Cryptographic Hash Function EDON-R. Submission to NIST, 2008. 2.3.3

[Hal09] Shai Halevi, editor. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*. Springer, 2009. A.9

[HI10] Seokhie Hong and Tetsu Iwata, editors. *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2010. A.9

[HKY08] Shoichi Hirose, Hidenori Kuwakado, and Hirotaka Yoshida. SHA-3 Proposal: Lesamnta. Submission to NIST, 2008. 2.3.1, 8.3.1, 9.1.2, 9.2, 9.3.1, 9.3.2, 1, 9.16, 9.A, 9.19, A.7

[HKY09] Shoichi Hirose, Hidenori Kuwakado, and Hirotaka Yoshida. Security Analysis of the Compression Function of Lesamnta and its Impact. Available online, 2009. 9.3.2, A.7

[JK97] Thomas Jakobsen and Lars R. Knudsen. The Interpolation Attack on Block Ciphers. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 1997. 8.6

[Jou04] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin [Fra04], pages 306–316. 1.3.4, 6.2.3

[Jou09] Antoine Joux, editor. *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*. Springer, 2009. A.9

[JP06] Charanjit S. Jutla and Anindya C. Patthak. Provably Good Codes for Hash Function Design. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 376–393. Springer, 2006. 6.2.2, 6.2.3, 7.4

[JP07] Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Menezes [Men07], pages 244–263. 3.1, 3.3.2, 3.5

[KBN09] Dmitry Khovratovich, Alex Biryukov, and Ivica Nikolic. Speeding up Collision Search for Byte-Oriented Hash Functions. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2009. 9.4.3

[KBPH06] Jongsung Kim, Alex Biryukov, Bart Preneel, and Seokhie Hong. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract). In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2006. 3.1.1, 3.5.6, 4.2

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, jan 1883. 1.1

[KJ01] Paul Kocher and Joshua Jaffe. Leak-resistant Cryptographic Method and Apparatus. US Patent No. 6,304,658, 2001. 1.2.2

[KK06] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006. 1.2.2, 1.3.4, 6.2.3, 8.3.3

[KKHS08] Taehyun Kim, Jongsung Kim, Seokhie Hong, and Jaechul Sung. Linear and Differential Cryptanalysis of Reduced SMS4 Block Cipher. Cryptology ePrint Archive, Report 2008/281, 2008. `http://eprint.iacr.org/`. 9.18

[Kli05] Vlastimil Klima. Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications. Cryptology ePrint Archive, Report 2005/102, 2005. `http://eprint.iacr.org/`. 3.1, 3.4.2

[Kli06] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/`. 3.1, 3.3.2, 3.4.2, 3.4.3, 3.4.6

[KM05] Lars R. Knudsen and John Erik Mathiassen. Preimage and Collision Attacks on MD2. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2005. 5.1

[KMMT10] Lars Knudsen, John Mathiassen, Frédéric Muller, and Søren Thomsen. Cryptanalysis of MD2. *J. Cryptology*, 23:72–90, 2010. 5.1

[KN10] Dmitry Khovratovich and Ivica Nikolić. Rotational Cryptanalysis of ARX. In Hong and Iwata [HI10]. 2.3.4, 8.1.1, 8.2

[Knu92] Lars R. Knudsen. Cryptanalysis of LOKI91. In Jennifer Seberry and Yuliang Zheng, editors, *AUSCRYPT*, volume 718 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 1992. 8.2

[Kob96] Neal Koblitz, editor. *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer, 1996. A.9

[KP91] Matthew Kwan and Josef Pieprzyk. A General Purpose Technique for Locating Key Scheduling Weakness in DES-like Cryptosystems (Extended Abstract). In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *ASIACRYPT*, volume 739 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 1991. 8.1, 8.2

[KS05] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Cramer [Cra05], pages 474–490. 1.3.4, 5.3.2, 6.2.3

[KSW96] John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In Koblitz [Kob96], pages 237–251. 8.1, 8.2

[LBF08] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. SIMD Is a Message Digest. Submission to NIST, 2008. `http://www.di.ens.fr/~leurent/files/SIMD.pdf`. 2.2.1, 2.5

[LBF09] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. Tweaking SIMD. Submission to the second round of SHA-3 competition, 2009. 7.1.1

[LC06] Xuejia Lai and Kefei Chen, editors. *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*. Springer, 2006. A.9

[LdW05] Arjen K. Lenstra and Benne de Weger. On the Possibility of Constructing Meaningful Hash Collisions for Public Keys. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 3574 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2005. 3.1.1

[Leu07] Gaëtan Leurent. Message Freedom in MD4 and MD5 Collisions: Application to APOP. In Biryukov [Bir07], pages 309–328. 2.1.2, 2.5, 3.1.1

[Leu08a] Gaëtan Leurent. MD4 is Not One-Way. In Nyberg [Nyb08], pages 412–428. 2.1.4, 2.5, A.1

[Leu08b] Gaëtan Leurent. Practical key-recovery attack against APOP, an MD5-based challenge-response authentication. *IJACT*, 1(1):32–46, 2008. 2.5

[Leu10] Gaëtan Leurent. Practical Key Recovery Attack against Secret-IV Edon-R. In Pieprzyk [Pie10], pages 334–349. 2.3.3, 2.5

[LL05] Jie Liang and Xuejia Lai. Improved Collision Attack on Hash Function MD5. Cryptology ePrint Archive, Report 2005/425, 2005. `http://eprint.iacr.org/`. 3.1

[LM92] Xuejia Lai and James L. Massey. Hash Function Based on Block Ciphers. In *EUROCRYPT*, pages 55–70, 1992. 1.3.2, 5.3, 9.1.1

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A Modest Proposal for FFT Hashing. In Nyberg [Nyb08], pages 54–72. 6.2.3

[LN09] Gaëtan Leurent and Phong Q. Nguyen. How Risky Is the Random-Oracle Model? In Halevi [Hal09], pages 445–464. 2.4.2, 2.5

[LR85] Michael Luby and Charles Rackoff. How to Construct Pseudo-Random Permutations from Pseudo-Random Functions (Abstract). In Hugh C. Williams, editor, *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, page 447. Springer, 1985. 2.3.1

[LSWD04] Tri Van Le, Rüdiger Sparr, Ralph Wernsdorf, and Yvo Desmedt. Complementation-Like and Cyclic Properties of AES Round Functions. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 2004. 8.1, 9.3.4

[Lu09] Jiqiang Lu. Related-key rectangle attack on 36 rounds of the XTEA block cipher. *Int. J. Inf. Sec.*, 8(1):1–11, 2009. 8.5.1

[Luc05] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005. 1.3.4, 6.2.2, 6.5.1

[M$^+$09] Florian Mendel *et al.*. A preimage attack on 8-round SHAvite-3-512. Graz ECRYPT meeting 2009, May 2009. 9.2, 9.17

[Mal08] Tal Malkin, editor. *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, volume 4964 of *Lecture Notes in Computer Science*. Springer, 2008. A.9

[Mar08] Jason Worth Martin. ESSENCE: A Candidate Hashing Algorithm for the NIST Competition. Submission to NIST, 2008. 2.3.4, 8.4, A.9

[Mat09] Mitsuru Matsui, editor. *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*. Springer, 2009. A.9

[Men07] Alfred Menezes, editor. *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*. Springer, 2007. A.9

[Mer89] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [Bra90], pages 428–446. 1.3.1

[MHP09] Cameron McDonald, Philip Hawkes, and Josef Pieprzyk. Differential Path for SHA-1 with complexity $O(2^{52})$. Cryptology ePrint Archive, Report 2009/259, 2009. `http://eprint.iacr.org/`. 3.1

[MN09] Florian Mendel and Tomislav Nad. A Distinguisher for the Compression Function of SIMD-512. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT*, volume 5922 of *Lecture Notes in Computer Science*, pages 219–232. Springer, 2009. 6.4.1, 7.1.1

[MP08] Stéphane Manuel and Thomas Peyrin. Collisions on SHA-0 in One Hour. In Nyberg [Nyb08], pages 16–35. A.3

[MR07] Florian Mendel and Vincent Rijmen. Weaknesses in the HAS-V Compression Function. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2007. 5.3.2

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004. 1.2.2, 7.3

[MRR07] Florian Mendel, Christian Rechberger, and Vincent Rijmen. Update on SHA-1. Presented at the rump session of CRYPTO '07, 2007. `http://rump2007.cr.yp.to/`. 3.1

[MRS09] Florian Mendel, Christian Rechberger, and Martin Schläffer. MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In Matsui [Mat09], pages 144–161. 3.1, 3.1.1, 3.5

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009. III, 9.F

[MRST10] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Rebound Attacks on the Reduced Grøstl Hash Function. In Pieprzyk [Pie10], pages 350–365. 9.F, 9.F.2

[MT07] Ueli M. Maurer and Stefano Tessaro. Domain Extension of Public Random Functions: Beyond the Birthday Barrier. In Menezes [Men07], pages 187–204. 6.5.1

[MTT09] Nicky Mouha, Søren S. Thomsen, and Meltem Sönmez Turan. Observations of non-randomness in the ESSENCE compression function. Available online, 2009. 8.4

[Mul04] Frédéric Muller. The MD2 Hash Function Is Not One-Way. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2004. 5.1

[Ngu06] Phong Q. Nguyen, editor. *Progressin Cryptology - VIETCRYPT 2006, First International Conferenceon Cryptology in Vietnam, Hanoi, Vietnam, September 25-28, 2006, Revised Selected Papers*, volume 4341 of *Lecture Notes in Computer Science*. Springer, 2006. A.9

[NP09] Mridul Nandi and Souradyuti Paul. OFFICIAL COMMENT: SHAvite-3. Available online, 2009. 9.4.1

[NPRA$^+$10] María Naya-Plasencia, Andrea Röck, Jean-Philippe Aumasson, Yann Laigle-Chapuy, Gaëtan Leurent, Willi Meier, and Thomas Peyrin. Cryptanalysis of ESSENCE. In Hong and Iwata [HI10]. 2.3.4, 2.5, 8.4, A.9

[NPSS10] Ivica Nikolić, Josef Pieprzyk, Przemyslaw Sokolowski, and Ron Steinfeld. Rotational Cryptanalysis of (Modified) Versions of BMW and SIMD. NIST hash forum, March 2010. 7.1.1

[NSKO05] Yusuke Naito, Yu Sasaki, Noboru Kunihiro, and Kazuo Ohta. Improved Collision Attack on MD4 with Probability Almost 1. In Dongho Won and Seungjoo Kim, editors, *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2005. 3.1, 3.4.2

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 1982. 6.3.1

[Nyb08] Kaisa Nyberg, editor. *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science.* Springer, 2008. A.9

[Pey09] Thomas Peyrin. Chosen-salt, chosen-counter, pseudo-collision on SHAvite-3 compression function, 2009. Available online at `http://ehash.iaik.tugraz.at/uploads/e/ea/Peyrin-SHAvite-3.txt` (Accessed on 10/1/2010). 9.4.1, 9.5.1

[Pey10] Thomas Peyrin. Improved Differential Attacks for ECHO and Grøstl. In Tal Rabin, editor, *Crypto '10*, Lecture Notes in Computer Science. Springer, 2010. 8.1.1

[PGV93] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993. 6.2.2, 6.5.1

[Pie10] Josef Pieprzyk, editor. *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*, volume 5985 of *Lecture Notes in Computer Science.* Springer, 2010. A.9

[PSLL03] Sangwoo Park, Soo Hak Sung, Sangjin Lee, and Jongin Lim. Improving the Upper Bound on the Maximum Differential and the Maximum Linear Hull Probability for SPN Structures and AES. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 247–260. Springer, 2003. 2.2.2, 6.2.3, 7.1

[PvO95] Bart Preneel and Paul C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995. 1.3.4, 4.2, 4.2, 6.2.3

[PvO96] Bart Preneel and Paul C. van Oorschot. On the Security of Two MAC Algorithms. In *EUROCRYPT*, pages 19–32, 1996. 2.1.2, 4.1, 4.2, 4.6

[PvO99] Bart Preneel and Paul C. van Oorschot. On the Security of Iterated Message Authentication Codes. *IEEE Transactions on Information Theory*, 45(1):188–199, 1999. 4.2

[QD89] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search? Application to DES (Extended Summary). In *EUROCRYPT*, pages 429–434, 1989. 1.2.3

[Rab78] Michael O. Rabin. Digitalized signatures. In Richard J. Lipton and Richard A. DeMillo, editors, *Foundations of Secure Computation*, pages 155–168. Academic Press, New York, 1978. 1.3

[RFC1081] M.T. Rose. Post Office Protocol: Version 3. RFC 1081, November 1988. Obsoleted by RFC 1225. 4.6

[RFC1321] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992. A.2

[RFC1460] M. Rose. Post Office Protocol - Version 3. RFC 1460 (Draft Standard), June 1993. Obsoleted by RFC 1725. 4.6

[RFC1730] M. Crispin. Internet Message Access Protocol - Version 4. RFC 1730 (Proposed Standard), December 1994. Obsoleted by RFCs 2060, 2061. 4.6

[RFC1734] J. Myers. POP3 AUTHentication command. RFC 1734 (Proposed Standard), December 1994. Obsoleted by RFC 5034. 4.6, 4.6.4

[RFC1760] N. Haller. The S/KEY One-Time Password System. RFC 1760 (Informational), February 1995. 5.1

[RFC1939] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Standard), May 1996. Updated by RFCs 1957, 2449. 1.2.2, 4.6, 4.6, 4.6.2

[RFC1994] W. Simpson. PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994 (Draft Standard), August 1996. Updated by RFC 2484. 1.2.2

[RFC2104] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. 6.5.3

[RFC2195] J. Klensin, R. Catoe, and P. Krumviede. IMAP/POP AUTHorize Extension for Simple Challenge/Response. RFC 2195 (Proposed Standard), September 1997. 1.2.2

[RFC2289] N. Haller, C. Metz, P. Nesser, and M. Straw. A One-Time Password System. RFC 2289 (Standard), February 1998. 1.2.2

[RFC2617] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. 4.6.3

[RFC3501] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 3501 (Proposed Standard), March 2003. Updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738. 4.6

[Riv90] Ronald L. Rivest. The MD4 Message Digest Algorithm. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990. 1.4, I, A.1

[Rob06] Matthew J. B. Robshaw, editor. *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*. Springer, 2006. A.9

[Rog06] Phillip Rogaway. Formalizing Human Ignorance. In Nguyen [Ngu06], pages 211–228. 1.2.3

[RR06] Christian Rechberger and Vincent Rijmen. Note on Distinguishing, Forgery, and Second Preimage Attacks on HMAC-SHA-1 and a Method to Reduce the Key Entropy of NMAC. Cryptology ePrint Archive, Report 2006/290, 2006. `http://eprint.iacr.org/`. 4.2, 4.2

[RR07] Christian Rechberger and Vincent Rijmen. On Authentication with HMAC and Non-random Properties. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography*, volume 4886 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2007. 4.2, 4.2, 4.5

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004. 1.2.3

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978. The basics of trap-door functions and the famous RSA public key cryptosystem are presented in this paper. 1.1.1

[RSM10] Mohammad Reza Reyhanitabar, Willy Susilo, and Yi Mu. Enhanced Security Notions for Dedicated-Key Hash Functions: Definitions and Relationships. In Hong and Iwata [HI10]. 9.5

[SA09] Yu Sasaki and Kazumaro Aoki. Finding Preimages in Full MD5 Faster Than Exhaustive Search. In Joux [Jou09], pages 134–152. 5.1.2

[Saa07] Markku-Juhani Olavi Saarinen. Linearization Attacks Against Syndrome Based Hashes. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2007. 2.3.5

[Sch89] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In Brassard [Bra90], pages 239–252. 1.1.1

[Sho05] Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*. Springer, 2005. A.9

[SLdW07] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007. 3.1, 3.1.1, 3.5

[Sma08] Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*. Springer, 2008. A.9

[SNKO05] Yu Sasaki, Yusuke Naito, Noboru Kunihiro, and Kazuo Ohta. Improved Collision Attack on MD5. Cryptology ePrint Archive, Report 2005/400, 2005. `http://eprint.iacr.org/`. 3.1

[SNY$^+$06] Yu Sasaki, Yusuke Naito, Jun Yajima, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta. How to Construct Sufficient Conditions for Hash Functions. In Nguyen [Ngu06], pages 243–259. 3.1, 3.5.1, 3.5.3

[SO06] Martin Schläffer and Elisabeth Oswald. Searching for Differential Paths in MD4. In Robshaw [Rob06], pages 242–261. 3.1, 3.5, 3.5.5, 3.2

[Sphl] Projet RNRT SAPHIR. sphlib 1.0. `http://www.crypto-hash.fr/modules/wfdownloads/singlefile.php?cid=9&lid=5`. 6.3.1

[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], pages 55–69. 3.1, 3.1.1, A.2

[Ste05] Michael Steil. 17 Mistakes Microsoft Made in the Xbox Security System. In *22nd Chaos Communication Congress*, 2005. 8.1

[Ste06] Marc Stevens. Fast Collision Attack on MD5. Cryptology ePrint Archive, Report 2006/104, 2006. `http://eprint.iacr.org/`. 3.1, 3.4.2, 3.4.2, 3.4.6

[Ste07] Marc Stevens. On Collisions for MD5. Master's thesis, Eindhoven University of Technology, 2007. 3.1, 3.4.2, 3.4.6, 3.5.5

[Sw06] Lu Shu-wang. Specification of SMS4, Block Cipher for WLAN Products – SMS4 (in Chinese), January 2006. `http://www.oscca.gov.cn/UpFile/200621016423197990.pdf`. 9.B, A.9

[SWOK07] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New Message Difference for MD4. In Biryukov [Bir07], pages 329–348. 3.1, 3.3.1, 3.3.1, 3.5.5, 3.2, 3.5.6, 5.2, A.1

[SWOK08] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Security of MD5 Challenge and Response: Extension of APOP Password Recovery Attack. In Malkin [Mal08], pages 1–18. 3.1, 3.1.1, 3.4.1, 3.5, 4.6.3

[SY06] Michael Szydlo and Yiqun Lisa Yin. Collision-Resistant Usage of MD5 and SHA-1 Via Message Preprocessing. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2006. 3.4, 3.4.7

[SYA07] Yu Sasaki, Go Yamamoto, and Kazumaro Aoki. Practical Password Recovery on an MD5 Challenge and Response. Cryptology ePrint Archive, Report 2007/101, 2007. `http://eprint.iacr.org/`. 4.6.3

[Vau94] Serge Vaudenay. On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In Bart Preneel, editor, *FSE*, volume 1008 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 1994. 5.1.1, 5.2.1, 5.2.1

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999. 1.2.3, 9.4.5, 9.4.5, 9.5.3

[WLF$^+$05] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [Cra05], pages 1–18. 1.4.1, 2.1.1, 3.1, 3.3.1, 3.4.4, 3.4.5, 3.4.5, 3.1, 3.5, 3.5.4, 3.5.4, 3.5.6, 3.5.6, 3.2, 1, 2, 3, 3.8, 4.2, 5.1.1, 7.4.4, A.1, A.9

[WOK08] Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Smart [Sma08], pages 237–253. 3.5, 3.5.6, 6, 4.2, 4.3.4, A.1, A.9

[WSSO09] Lei Wang, Yu Sasaki, Kazuo Sakiyama, and Kazuo Ohta. Bit-Free Collision: Application to APOP Attack. In Tsuyoshi Takagi and Masahiro Mambo, editors, *IWSEC*, volume 5824 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2009. 4.6.3

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [Cra05], pages 19–35. 1.3.2, 1.4.1, 2.1.1, I, 3.1, 3.3.1, 3.3.1, 3.4.4, 3.4.6, 4.2, 4.6.2, 6.2.3, A.2

[WYW$^+$09] Xiaoyun Wang, Hongbo Yu, Wei Wang, Haina Zhang, and Tao Zhan. Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC. In Joux [Jou09], pages 121–133. 8.3.3

[WYY05a] Xiaoyun Wang, Andrew Yao, and Frances Yao. New Collision Search for SHA-1. Presented by Adi Shamir at the rump session of Crypto '05, 2005. 3.1

[WYY05b] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Shoup [Sho05], pages 17–36. 1.4.1, 2.1.1, I, 3.1, 4.2, 6.2.3, 7.4.4, A.4

[WYY05c] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Shoup [Sho05], pages 1–16. 3.1, 4.2, A.3

[XLF08] Tao Xie, FanBao Liu, and DengGuo Feng. Could The 1-MSB Input Difference Be The Fastest Collision Attack For MD5? Cryptology ePrint Archive, Report 2008/391, 2008. `http://eprint.iacr.org/`. 3.1, 3.3.1, 4.6.3

[YS05] Jun Yajima and Takeshi Shimoyama. Wang's sufficient conditions of MD5 are not sufficient. Cryptology ePrint Archive, Report 2005/263, 2005. `http://eprint.iacr.org/`. 3.1

[YWZW05] Hongbo Yu, Gaoli Wang, Guoyan Zhang, and Xiaoyun Wang. The Second-Preimage Attack on MD4. In Yvo Desmedt, Huaxiong Wang, Yi Mu, and Yongqing Li, editors, *CANS*, volume 3810 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005. 3.3.1, 3.4.1, 3.4.5, 3.1, 3.5, 3.5.4, 3.5.6, 3.5.6, 3.A, 4, 4.2, 5.1.2, A.9

[ZZW08] Lei Zhang, Wentao Zhang, and Wenling Wu. Cryptanalysis of Reduced-Round SMS4 Block Cipher. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *Lecture Notes in Computer Science*, pages 216–229. Springer, 2008. 9.18