Meet-in-the-Middle Attacks on AES

Patrick Derbez

To cite this version:

Patrick Derbez. Meet-in-the-Middle Attacks on AES. Cryptography and Security [cs.CR]. Ecole Normale Supérieure de Paris - ENS Paris, 2013. English. NNT : . tel-00918146

HAL Id: tel-00918146
https://theses.hal.science/tel-00918146

Submitted on 17 Dec 2013

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

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

École doctorale de sciences mathématiques de Paris-Centre (ED 386)

Attaques par Rencontre par le Milieu sur l'AES

THÈSE

présentée et soutenue publiquement le 9 Décembre 2013

pour l'obtention du

Doctorat de l'École Normale Supérieure
(Spécialité Informatique)

par

Patrick Derbez

Composition du jury

Directeur de thèse :	Pierre-Alain Fouque
Rapporteurs :	Gilles Barthe Jacques Patarin
Examinateurs :	Henri Gilbert Gaëtan Leurent David Pointcheval

Département d'Informatique de l'ENS — UMR 8548
École Normale Supérieure

Mis en page avec la classe thesul.

Remerciements

En premier lieu je veux remercier Pierre-Alain Fouque pour avoir été un excellent directeur tout au long de cette thèse. J'ai réellement apprécié travailler avec lui car outre son appui scientifique et la grande autonomie qu'il m'a laissée, il a également toujours été là pour me soutenir et me conseiller durant ces trois années. Je le remercie aussi ainsi que l'ENS pour m'avoir donner la chance de participer à autant de conférences internationales qui, malgré le trac que j'ai pu avoir avant mes présentations, me laisse de très bon souvenirs.

Je suis très reconnaissant envers les deux rapporteur de cette thèse, Gilles Barthe et Jacques Patarin, pour avoir examiner mon travail en le peu de temps qui leur était imparti. J'associe également à ces remerciements Henry Gilbert, Gaëtan Leurent et David Pointcheval qui ont accepté de participer à ce jury de thèse.

Merci à toutes les personnes de l'ENS et d'ailleurs avec qui j'ai pu échanger au cours de l'élaboration de cette thèse et en particulier à tous mes co-auteurs pour nos collaborations fructueuses. Des remerciements spéciaux à Jérémy Jean et Charles bouillaguet sans qui ma thèse aurait compté beaucoup moins de moments sympathiques et de travaux intéressants.

Enfin je souhaite remercier toute ma famille et tous mes amis en particulier pour le soutien qu'ils m'ont apporté pendant cette thèse. Aussi, merci beaucoup à mes parents pour m'avoir toujours encouragé dans mes études et dans la vie. Merci beaucoup à Thomas et Florent pour tous ces bons moments en France, en Irlande ou sur SC2. Et finalement, un énorme merci à ma compagne Natacha pour m'avoir supporté et soutenu quotidiennement durant toutes ces années. Je te remercie de me pardonner pour tous ces week-ends et toutes ces soirées plongé dans mes recherches. La réussite de cette thèse te doit beaucoup et c'est pourquoi je te la dédie.

À ma Pacsette.

Table des matières

1 Introduction **1**

1.1 Cryptographie Symétrique 1

1.1.1 Les Types de Chiffrements Symétriques 2

1.1.2 Sécurité des Chiffrements 3

1.2 Présentation de mes travaux 3

1.2.1 Mes Résultats 7

1.2.2 Listes de mes Publications 7

2 Automated Tool For Low Data Complexity Attacks on AES and Derivatives **11**

2.1 Guess-and-Determine Solvers 12

2.1.1 Adapting the Gaussian Elimination 13

2.1.2 Finding the Best Solver 20

2.2 Recursive Meet-in-the-Middle Solvers 29

2.2.1 Solving Subsystems Recursively 29

2.2.2 Recursive Combinations of Solvers 30

2.2.3 Finding the best solver 31

2.3 Conclusion 37

2.3.1 Other Settings and Open problems 38

3 Low Data Complexity Attacks on Round-Reduced AES-128 **41**

3.1 Low Data Complexity Attacks 41

3.2 Description of the AES 44

3.3 Observations on the Structure of AES 46

3.4 Attack on One-Round AES 47

3.4.1 Two Known Plaintexts 47

3.4.2 One Known Plaintext 48

3.5 Attacks on Two-Round AES 51
3.5.1 Two Known Plaintexts. 51
3.5.2 A Three Known Plaintext Variant 54
3.5.3 A Two Chosen Plaintext Variant 54
3.5.4 One Known Plaintext 57
3.5.5 Improved Attack When the Second MixColumns is Omitted 60
3.6 Attacks on Three-Round AES 60
3.6.1 Two Chosen Plaintexts 60
3.6.2 Nine Known Plaintexts 60
3.6.3 One Known Plaintext 62
3.7 Attacks on Four-Round AES 62
3.7.1 Ten Chosen Plaintexts 64
3.7.2 Five Chosen Plaintexts 64
3.7.3 Four Chosen Plaintexts. 65
3.8 Attack on Five-Round AES 65
3.8.1 One Known Plaintext. 66
3.9 Attack on Six-Round AES 66
3.10 Implementations 69

4 Low Data Complexity Attacks on AES-Derivatives 71
4.1 A Forgery Attack Against Pelican-MAC 71
4.2 A Key-Recovery Attack Against LEX 73
4.2.1 Prior Art 74
4.2.2 A New Attack 76

5 Fault Attacks on the AES 79
5.1 Fault Analysis 79
5.1.1 Related Works 80
5.2 Meet-in-the-Middle Fault Analysis on AES 81
5.2.1 Original Attack of Piret-Quisquater 81
5.2.2 Improvement of Piret-Quisquater Attack 81
5.2.3 Extension to One More Round 82
5.3 Impossible Differential Fault Attack on AES 84
5.4 Conclusion 86

6 Faster Chosen-Key Distinguishers on Reduced-Round AES 87

6.1 Chosen-key distinguishers 88

6.1.1 Limited Birthday Distinguishers 88

6.1.2 Distinguisher for 7-round AES-128 90

6.1.3 Distinguisher for 8-round AES-128 93

6.2 Extention to AES-256 97

6.2.1 Distinguisher for 7-round AES-256 97

6.2.2 Distinguisher for 8-round AES-256 98

6.2.3 Distinguisher for 9-round AES-256 98

6.3 Conclusion 99

7 Exhausting Demirci-Selcuk Meet-in-the-Middle Attacks against Reduced-Round AES 103

7.1 Attack of Demirci and Selçuk and Improvements 103

7.1.1 The Demirci and Selçuk Attack 104

7.1.2 Previous Improvements of the Original Attack 106

7.2 Generalization of the Demirci and Selçuk Attack 107

7.2.1 New Improvements of the Original Attack 107

7.2.2 Finding the Best Attack 108

7.3 Results 108

7.3.1 Overview of the Results 108

7.3.2 Observation on the Keyschedules 110

7.3.3 Attack on Six-Round AES-128 with 2^8 chosen-plaintexts 111

7.3.4 Attack on Seven-Round AES-256 with 2^{16} chosen-plaintexts 113

7.3.5 Attack on Seven-Round AES-192 with 2^{32} chosen-plaintexts 113

7.4 An SPN-dedicated Tool 114

7.5 Application to Low Data Complexity Attacks 114

7.5.1 Attack on Five-Round AES-128 with Eight Chosen Plaintexts . . . 115

7.5.2 Attack on Six-Round AES-128 with Thirteen Chosen Plaintexts . . 116

8 The Differential Enumeration Technique 125

8.1 Unified View of Previously Known MITM Attacks on AES 127

8.2 New Attack on AES 129

8.2.1 Efficient Tabulation 129

8.2.2 Simple Attack 131

8.2.3 Efficient Attack: New Property ★ 133
8.2.4 Turning the distinguisher into a key recovery attack 135
8.3 Extension to More Rounds 136
8.3.1 Eight-Round Attacks on AES-192 and AES-256 136
8.3.2 Improved Eight-Round Attack 138
8.3.3 Nine-Round Attack on AES-256 140
8.3.4 Nine-Round Attack on AES-192 140
8.4 Conclusion 141

Bibliographie **151**

Table des figures **161**

Chapitre 1

Introduction

1.1 Cryptographie Symétrique

La cryptographie symétrique, ou cryptographie à clé secrète, est utilisée depuis des milliers d'années et inclue tous les cas où la même clé est utilisée pour chiffrer et déchiffrer un message donné. L'un des exemples les plus connus est sans doute le « chiffre de César », aujourd'hui appelé «chiffrement par décalage », ayant, selon la légende, servi à Jules César pour pour ses correspondances secrètes et dont le principe est de décaler dans l'alphabet chaque lettre du message, la clé secrète étant le décalage.

Une variante de ce système cryptographique est de considérer une permutation quelconque de l'alphabet et non plus un décalage. Dans ce cas la clé pourrait être une longue séquence de nombres comme 5, 19, 1, 2, 11, ... indiquant que A correspond à E, B à S, C à A, D à B, E à K, ... De tels systèmes sont en réalité particulièrement faibles et la cryptographie moderne repose sur des problèmes mathématiques réputés très compliqués à résoudre.

A la différence de la cryptographie à clé publique où le chiffrement est réalisé à l'aide du clé (potentiellement) connue de tous mais où la clé déchiffrement n'est (pratiquement) jamais partagé, la cryptographie symétrique requière que la même clé soit connue et maintenue secrète par un petit groupe de personnes. Si cette clé secrète tombe entre de mauvaises mains alors la confidentialité de tous leurs échanges est immédiatement et complètement compromise. Ainsi se pose le problème de la gestion et de l'échange de clés secrètes. En particulier, l'environnement dans lequel est effectué le chiffrement doit être sous le contrôle total de l'utilisateur pour garantir le meilleur niveau de sécurité, ce qui exclu de fait les systèmes d'exploitation fermés.

On fait souvent référence à des clés de tailles particulières comme 64 ou 128 bits. Ces longueurs sont celles utilisées dans les algorithmes de chiffrement symétriques. À l'heure actuelle, on admet que personne sur Terre ne peut effectuer plus de 2^{80} opérations en un temps raisonnable. Ainsi un système de chiffrement est considéré comme sûr si il utilise une clé secrète d'au moins 80 bits et si la meilleure façon d'obtenir la clé secrète est de toutes les tester. Cependant, pour certaines applications peu sensibles comme les cartes de transport en commun, une sécurité de 64 bits est largement suffisante car réaliser 2^{64} opérations est loin d'être à la portée de tout le monde tant au niveau de la puissance de

calcul que du coût en électricité. En revanche, dans le cas de la cryptographie asymétrique la taille de clés (ou au moins de la clé privée) est beaucoup plus grande. On estime qu'avec l'algorithme de chiffrement RSA une clé privée de 1024 bits correspond approximativement à la sécurité offerte par une clé de 80 bits dans le cas symétrique.

Les algorithmes de chiffrement symétriques sont considérablement plus rapides que les méthodes à clé publique et sont donc préférés lorsque la quantité de données à chiffrer est grande. Par exemple, le DES (Data Encryption Standard) est au moins 100 fois plus rapide que le RSA en implémentation logicielle et jusqu'à 10 000 fois plus rapide sur machines dédiées. En pratique, les systèmes de chiffrement modernes sont hybrides : une clé secrète est échangée utilisant un algorithmes de chiffrement à clé publique puis les données sont chiffrées/déchiffrer avec un système symétrique.

1.1.1 Les Types de Chiffrements Symétriques

Les systèmes de chiffrement symétriques sont maintenant généralement mis en œuvre en utilisant des algorithmes de chiffrement par blocs ou par flot, qui sont examinées dans cette section. On présentera aussi ce qu'on appelle les *codes d'authentification de message* (MAC), un mécanisme de contrôle qui utilise une clé secrète.

Chiffrement par Bloc. Un algorithme de chiffrement par bloc transforme un message (claire) de taille fixée en un message (chiffré) de même taille sous l'action d'une clé secrète. Le déchiffrement est effectué en utilisant la transformation inverse et la même clé. De nos jours la taille des blocs de messages est le plus souvent de 64 ou 128 bits.

La taille des données à chiffrées est le plus souvent bien plus grande que la taille de l'entrée d'un algorithme de chiffrement par bloc et donc différentes techniques, appelés *modes d'opérations*, sont utilisées. On peut citer par exemple le mode « Dictionnaire de codes »(*Electronic code book*, ECB), « Enchaînement des blocs »(*Cipher Block Chaining, CBC*) ou encore « Chiffrement à rétroaction »(*Cipher Feedback, CFB*). avec le mode ECB le message est divisé en plusieurs blocs puis chacun est chiffré séparément. Le gros défaut de cette méthode est que deux blocs avec le même contenu seront chiffrés de la même manière et l'on peut donc tirer des informations à partir des données chiffrées en cherchant les séquences identiques et ce mode est donc à proscrire dans certains cas comme par exemple le chiffrement d'une image. Dans le mode CBC, on ajoute à chaque bloc le chiffrement du bloc précédent avant qu'il ne soit lui-même chiffré. Ainsi ce mode ajoute un niveau de sécurité supplémentaire rendant certaines attaques bien plus difficiles à réaliser. Une erreur courante faite par les développeurs non-spécialistes est d'utiliser un algorithme de chiffrement par bloc en mode ECB plutôt que d'utiliser le mode d'opération le plus adapté pouvant fournir des garanties supplémentaires.

La plupart des algorithmes de chiffrement symétriques sont aujourd'hui construit par itération. Plus précisément, le même algorithme de chiffrement (appelé «fonction de tour ») est appliqué plusieurs fois utilisant une clé de tours (ou sous-clé). Les sous-clés sont générées à partir de de la clé secrète par un algorithme, appelé « algorithme de cadencement de clé »ou « keyschedule ». Le nombre de tours choisi pour un algorithme de chiffrement est un compromis entre la sécurité et la vitesse d'exécution voulu pour le système.

Les algorithmes de chiffrement par bloc les plus connus et utilisés sont le DES, IDEA,

SAFER, Blowfish, Skipjack et l'AES qui est l'algorithme recommandé par le NIST depuis 2001.

Chiffrement par Flot. Les algorithmes de chiffrements par flot peuvent être extrêmement rapides par rapport aux chiffrement par bloc, bien que certains de ces derniers sont aussi efficaces dans certains mode d'opérations (comme le DES en CFB ou OFB). Un système de chiffrement par flot se présente souvent sous la forme d'un générateur de nombres pseudo-aléatoires avec lequel on opère un « OU Exclusif »entre un bit à la sortie du générateur et un bit provenant des donnée.

Les systèmes de chiffrement par flot sont basés sur le « chiffre de Vernam », seul algorithme de chiffrement théoriquement sûr bien que présentant d'importantes difficultés de mise en œuvre pratique. En exemple de chiffrements par flot on peut citer RC4 et SEAL, ainsi que HC-128, Rabbit, Salsa20/12, SOSEMANUK, Grain, MICKEY et Trivium du projet européen eSTREAM.

Code d'Authentification de Message. Un code d'authentification de message (*Message Authentication Code, MAC*) n'est pas à proprement parler un chiffrement mais plutôt un type particulier mécanisme de contrôle généré à partir d'une clé secrète et attaché au message. Contrairement aux fonctions de hachage qui sont publiques et plus généralement aux signatures électroniques, seuls une personne autorisée (connaissant le secret) peut valider le code.

1.1.2 Sécurité des Chiffrements

La taille de la clé secrète est l'un des nombreux facteurs qui déterminent le degré de sécurité d'un système de chiffrement. Comme avec tous les problèmes relatifs à la sécurité, ce qui est important est le compromis entre les risques, les coûts et le temps d'exécution entre autres choses. Par exemple, personne ne dépenserait 1 000 000 € dans une serrure sophistiquée pour protéger 1 000 € de biens.

Les développeurs doivent donc évaluer précisément leurs besoins en fonction des coûts de développement, la vitesse d'exécution, le paiement éventuel de droit d'exploitation, et le niveau de sécurité. En revanche, il est clair qu'à restriction équivalente il vaut mieux utiliser la solution offrant les meilleures garanties en termes de sécurité tout en restant cohérent avec le milieu dans lequel le système sera déployé.

Il est également extrêmement important d'examiner les modalités d'application d'algorithmes particuliers, certaines pouvant ne pas être très sûr. S'ajoute à cela la question de permettre l'examen public, ce qui est essentiel pour assurer la confiance dans le produit. N'importe quel développeur ou éditeur de logiciels qui refuse de rendre les éléments cryptographiques de leur application disponible publiquement ne méritent probablement pas la confiance et fournit, presque certainement, un produit de qualité inférieure.

1.2 Présentation de mes travaux

Ma thèse a été consacrée à l'algorithme de chiffrement symétrique AES, aussi connu sous le nom de Rijndael. Il a été conçu par Joan Daemen and Vincent Rijmen et est le

grand gagnant d'une compétition internationale lancée par le NIST en 1997 pour trouver un successeur au DES dont la taille de la clé de 56 bits ne permettait plus de garantir une sécurité suffisante dû à la forte augmentation de la puissance de calcul des ordinateurs.

Son statut de standard international a fait de l'AES l'un des chiffrements à clé secrète les plus utilisés dans le monde et donc tout naturellement aussi l'un des plus étudiés par les cryptanalystes. Deux principaux reproches ont été fait à l'AES, le premier concernant la simplicité algébrique de sa description et le second concernant son algorithme de cadencement de clé. En 2002, Nicolas Courtois et Josef Pieprzyk annonçaient une attaque algébrique sur l'AES ([CP02]) mais qui fût quelques temps plus tard infirmée ([CL05]) et la possibilité de monter une attaque algébrique plus rapide que la recherche exhaustive est toujours aujourd'hui un problème ouvert. En revanche, en 2009, Alex Biryukov and Dmitry Khovratovich utilisèrent les faiblesses de l'algorithme de cadencement de clé des versions 192 et 256 bits de l'AES pour obtenir des attaques plus rapide que la recherche exhaustive pour ces deux versions ([BK09]). Dès lors l'AES peut être considérer comme vulnérable mais, premièrement, la complexité théorique de ses attaques est bien trop élevée pour être praticable et, deuxièmement, ils se placent dans un modèle fort où l'adversaire peut observer les opérations d'un algorithme de chiffrement lorsqu'il est utilisé avec différentes clés, aux valeurs inconnues, mais qui sont liées entre elles par des propriétés mathématiques connues de l'attaquant. Ainsi, de l'avis de la communauté scientifique, et aux vues des connaissances actuelles, l'algorithme de chiffrement AES est toujours considéré comme sûr.

Mes travaux sur l'AES ont commencé par la remarque suivante : il est en pratique souvent plus facile d'effectuer 2^{50} opérations sur un ordinateur que de voler 50 couples claire/chiffré à quelqu'un. Ainsi je me suis intéressé au modèle d'attaque où la quantité de données en possession de l'adversaire est limitée à un maximum d'une dizaine de couples clair/chiffré. Avec une telle restriction, les attaques classiques en cryptographie symétrique telles que les attaques statistiques ne sont pas applicables car elles requièrent beaucoup plus de données. Il est bien connu que les attaques algébriques n'ont besoin que de peu données pour fonctionner et c'est donc tout naturellement que je me suis intéressé au système d'équations décrivant l'AES. Mais, au lieu d'essayer de résoudre ces équations avec un SAT-solver ou un algorithme basé sur les bases de Gröbner, j'ai utilisé des méthodes plus simples combinant l'algèbre linéaire avec les techniques algorithmiques de *la rencontre par le milieu* (Meet-in-the-Middle) et du *diviser pour régner*. Cela à donner naissance à un outil permettant de résoudre un certain type d'équations sur les corps finis ou, plus précisément, pour un système d'équations donné, de décrire une façon de le résoudre ainsi qu'une bonne estimation de la complexité en temps et en mémoire de cette résolution.

Comme il a été développé pour, cet outil a tout d'abord été utilisé pour trouver des attaques sur un nombre réduit de tours d'AES lorsque la quantité de couples claire/chiffré est très limitée, trouvant des attaques ayant échapper aux cryptanalystes. Ensuite, en tant qu'outil assez générique, il a pu être appliquer à d'autres primitives cryptographiques comme le chiffrement par flot LEX et le code d'authentification de message Pelican-MAC. Enfin, j'ai utilisé une version légèrement modifiée de cet outil pour exhauster certains types d'attaques, me permettant d'améliorer la complexité en données des attaques de Demirci

et Selçuk publiées à FSE 2008, ainsi que de trouver les meilleures attaques connues sur 7 tours d'AES toutes versions confondues, sur 8 tours d'AES-192 et 256 et sur 9 tours d'AES-256.

L'organisation de ce manuscrit suit cet ordre chronologique et dans la suite de cette section je décris plus en détail chacune de mes contributions.

Automated Tool For Low Data Complexity Attacks on AES and Derivatives.

Ce chapitre est dédié à la description de l'outil ou, plus précisément, du problème posé ainsi que de la solution apportée. L'idée principale pour résoudre un système d'équations du type considéré est de combiner ensemble par la technique du «Meet-in-the-Middle »des algorithmes énumérant toutes les solutions de sous-systèmes d'équations. Il y a plusieurs façons de construire un algorithme permettant de résoudre un système et le principal problème est de trouver la plus efficace. Malheureusement il est le plus souvent impossible d'exhauster toutes ces façons mais néanmoins les résultats obtenus sont déjà bien meilleurs que ceux trouvé précédemment à la main. Trouver de meilleurs algorithmes de recherche est aujourd'hui un problème ouvert qui m'intéresse tout particulièrement.

Low Data Complexity Attacks on Round-Reduced AES-128.

Dans ce chapitre sont décrites la majorité des attaques trouvées par l'outil sur les versions réduites de l'AES-128. Avec seulement 1 message à disposition de l'adversaire, l'outil a réussi à trouver des attaques sur 1, 2, 3, 4 et 5 tours. L'utilisation de plus de couples claire/chiffré n'a pas permis de casser plus de tours car le nombre de variables devient trop grand pour être géré par le programme. En revanche, cela a permis d'améliorer la complexité des attaques sur moins tours avec en particulier une attaque sur 2 tours en 2^8 opérations utilisant 2 claires choisies et une attaque sur 4 tours en 2^{32} opérations utilisant 4 claires choisies. Toutes les attaques trouvées ayant une complexité estimée inférieure ou égale à 2^{32} ont étaient testées en pratique, confirmant, chaque fois, les estimations faites.

Low Data Complexity Attacks on AES Derivatives.

Comme dit précédemment, l'outil a également donné des résultats sur un MAC basé sur l'AES, Pelican-MAC et sur un système de chiffrement par flot également basé sur l'AES, LEX.

Sur le code d'authentification de message Pelican-MAC proposé par Joan Daemen et Vincent Rijmen, l'outil a aidé à la construction de la meilleure attaque connue à ce jour, en mettant au point une procédure qui révèle l'état interne de la fonction en temps 2^{32} une fois qu'une collision sur l'état interne est trouvée. Cela est obtenu en résolvant l'équation $AES_4(x) - AES_4(x + \Delta_i) = \Delta_o$ en la variable x, où AES_4 désigne 4 tours complets d'AES sans addition de clef. L'attaque résultante a une complexité de 2^{64} requêtes au MAC (pour obtenir la collision).

Sur le système de chiffrement LEX proposé par Alex Biryukov, l'outil a également été utilisé pour construire une meilleure attaque. Il a d'abord aisément retrouvé tout seul la meilleure attaque connue, de complexité environ 2^{100} opérations, puis, avec l'intervention

d'un utilisateur, il a produit une attaque de complexité 2^{80} utilisant 80 tera-octets de key-stream.

Fault Attacks on the AES.

Les attaques par canaux auxiliaires sont aujourd'hui les seules à même de menacer la sécurité des systèmes de chiffrement. Le principe est d'étudier l'environnement dans lequel est exécuté le chiffrement pour obtenir des informations sur les calculs effectués et ainsi obtenir une partie ou la totalité de la clé ou d'états internes. Dans ce chapitre on s'intéresse aux attaques par fautes, modèle dans lequel l'adversaire peut modifier un ou plusieurs octets d'un état interne. L'outil a permis d'améliorer l'attaque par faute de Piret et Quisquater, en réduisant sa complexité de 2^{32} opérations à 2^{24}. Plus intéressant encore, il a trouvé une attaque en 2^{40} lorsqu'une faute est injectée 3 tours avant le dernier, soit un tour de plus que dans le cas de Piret et Quisquater, répondant ainsi à un problème ouvert. Enfin, dans ce chapitre est aussi décrite une nouvelle attaque par faute basée sur la technique de la *différentielle impossible*.

Faster Chosen-Key Distinguishers on Reduced-Round AES.

Dans ce chapitre on étudie l'AES dans un modèle complètement différent de précédemment. Ici le but n'est plus de retrouver la clé à partir de plusieurs couples claire/chiffré mais de générer rapidement des triplets composés d'une clé et d'un couple de messages vérifiant certaines propriétés particulières.

Exhausting Demirci-Selcuk Meet-in-the-Middle Attacks against Reduced-Round AES.

Dans ce chapitre je décris une amélioration de l'attaque de Demirci et Selçuk sur l'AES présentée à FSE 2008. Plus précisément je montre comment monter approximativement 2^{16} attaques similaires à la leur mais avec des complexités possiblement différentes. Pour chacune d'elles, la complexité dépend du nombre de valeurs que peuvent prendre un certains nombre de variables intermédiaires et de la vitesse à laquelle on peut les énumérer. Ainsi, en utilisant l'outil j'ai pu exhauster toutes ces attaques et trouver les meilleures. Cela a permis d'améliorer grandement la complexité en données des attaques de Demirci et Selçuk.

The Differential Enumeration Technique.

Dans ce chapitre je présente une version corrigée d'une technique introduite par Orr Dunkelman, Nathan Keller et Adi Shamir pour améliorer l' attaque de Demirci et Selçuk présentée dans le chapitre précédent. Combinée à ma propre amélioration de l' attaque de Demirci et Selçuk, cette technique m'a permis d'obtenir les meilleurs attaques connues (dans le modèle à claires choisis) sur 7 tours d'AES toutes versions confondues, sur 8 tours d'AES-192 et 256 et sur 9 tours d'AES-256.

1.2.1 Mes Résultats

Pendant ma thèse j'ai eu la chance de participer à la découverte de nouveaux résultats sur l'AES et ses dérivés souvent meilleurs que ceux précédement trouvé. Nombre d'entre eux sont décrit dans ce manuscrit et repertoriés dans les Tables 1.1 et 1.2. La Table 1.1 contient les complexités de nos attaques sur l'AES-128 dans le modèle à clairs connus ainsi que dans celui à clairs choisis. Celles des nos attaques sur les versions 192 et 256-bit de l'AES sont reportés sur la Table 1.2. Des tableaux de résultats plus complet sont donnés dans les chapitres suivant contenant aussi les résultats autres que les nôtre pour mesurer les améliorations que nous avons apportées.

1.2.2 Listes de mes Publications

- Automatic Search of Attacks on Round-Reduced AES and Applications.
  Charles Bouillaguet, Patrick Derbez, Pierre-Alain Fouque (CRYPTO 2011)
- Meet-in-the-Middle and Impossible Differential Fault Analysis on AES.
  Patrick Derbez, Pierre-Alain Fouque, Delphine Leresteux (CHES 2011)
- Faster Chosen-Key Distinguishers on Reduced-Round AES.
  Patrick Derbez, Pierre-Alain Fouque, Jérémy Jean (INDOCRYPT 2012)
- Low-Data Complexity Attacks on AES.
  Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Pierre-Alain Fouque, Nathan Keller, Vincent Rijmen (IEEE Transactions on Information Theory 58)
- Improved Key Recovery Attacks on Reduced-Round AES in the Single-Key Setting.
  Patrick Derbez, Pierre-Alain Fouque, Jérémy Jean (EUROCRYPT 2013)
- Exhausting Demirci-Selcuk Meet-in-the-Middle Attacks against Reduced-Round AES.
  Patrick Derbez, Pierre-Alain Fouque (FSE 2013)

Table 1.1 – Summary of our key-recovery attacks on AES-128 described in this thesis.

	Complexity			
#Rounds	Data	Time	Memory	Description
1	1 KP	2^{32}	2^{16}	Section 3.4.2
2	1 KP	2^{64}	2^{48}	Section 3.5.4
2	2 KP	2^{32}	2^{24}	Section 3.5.1
$2^{\star}$	2 KP	2^{24}	2^{16}	Section 3.5.5
2	2 CP	2^{8}	2^{8}	Section 3.5.3
3	1 KP	2^{96}	2^{72}	Section 3.6.3
3	2 CP	2^{16}	2^{8}	Section 3.6.1
4	4 CP	2^{32}	2^{24}	Section 3.7.3
$5^{\star}$	1 KP	2^{120}	2^{96}	Section 3.8.1
5	8 CP	2^{64}	2^{56}	Section 7.5.1
6	$2^{108.5}$ KP	2^{112}	$2^{108.5}$	Section 3.9
6	13 CP	2^{120}	2^{96}	Section 7.5.2
6	2^{8} CP	$2^{106,17}$	$2^{106,17}$	Section 7.3.3
7	2^{32} CP	$2^{126.47}$	$2^{126.47}$	Appendix 7.B.6
7	2^{105} CP	2^{99}	2^{90}	Section 8.2
7	2^{97} CP	2^{99}	2^{98}	Section 8.2.3

KP : Known-plaintext – CP : Chosen-plaintext.
$\star$: Last MixColumns omitted.
Time complexity is measured in approximate encryption units.
Memory complexity is measured approximately in 128-bit blocks.

Table 1.2 – Summary of our key-recovery attacks on AES-192 and 256 described in this thesis.

Version	#Rounds	Complexity			Description
		Data	Time	Memory	
192	6	2^{8}	$2^{109.67}$	$2^{109.67}$	Appendix 7.B.1
	7	2^{8}	2^{163}	$2^{153.34}$	Appendix 7.B.3
	7	2^{32}	$2^{129.67}$	$2^{129.67}$	Section 7.3.5
	7	2^{105}	2^{99}	2^{90}	Section 8.2
	7	2^{97}	2^{99}	2^{98}	Section 8.2.3
	8	2^{32}	$2^{182.17}$	$2^{182.17}$	Appendix 7.B.8
	8	2^{107}	2^{172}	2^{96}	Section 8.3.1
	8	$2^{104.83}$	2^{140}	$2^{138.17}$	Section 8.3.2
256	6	2^{8}	2^{122}	$2^{113.34}$	Appendix 7.B.2
	7	2^{8}	$2^{170.34}$	2^{186}	Appendix 7.B.4
	7	2^{16}	2^{178}	$2^{153.34}$	Section 7.3.4
	7	2^{32}	$2^{133.67}$	$2^{133.67}$	Appendix 7.B.7
	7	2^{105}	2^{99}	2^{90}	Section 8.2
	7	2^{97}	2^{99}	2^{98}	Section 8.2.3
	8	2^{8}	$2^{234.17}$	$2^{234.17}$	Appendix 7.B.5
	8	2^{32}	2^{195}	$2^{193.34}$	Appendix 7.B.9
	8	2^{107}	2^{196}	2^{96}	Section 8.3.1
	8	$2^{102.83}$	2^{156}	$2^{140.17}$	Section 8.3.2
	9	2^{32}	$2^{254.17}$	$2^{254.17}$	Appendix 7.B.10
	9	2^{120}	2^{203}	2^{203}	Section 8.3.3

KP : Known-plaintext – CP : Chosen-plaintext.
$\star$: Last MixColumns omitted.
Time complexity is measured in approximate encryption units.
Memory complexity is measured in 128-bit blocks.

Chapitre 2

Automated Tool For Low Data Complexity Attacks on AES and Derivatives

Breaking a good cipher should require "as much work as solving a system of simultaneous equations in a large number of unknowns of a complex type".

Since the introduction of the AES in 2001, it has been questioned whether its simple algebraic structure could be exploited by cryptanalysts. Soon after its publication as a standard [NIS01], Murphy and Robshaw showed in 2002 an interesting algebraic property: the AES encryption process can be described only with simple algebraic operations in $\mathbb{F}_{2^8}$ [MR02]. Such a result paved the way for multivariate algebraic techniques [CP02, Cid04] since the AES encryption function can also be described by a very sparse overdetermined multivariate polynomials system over $\mathbb{F}_2$. However, so far this approach has not been so promising [MV04, CL05], and the initial objective of this simple structure, providing good security protections against differential and linear cryptanalysis, has been fulfilled.

In this chapter, we describe an automated tool we originally designed to find very low data complexity attacks on the AES. The global strategy is to encode the cryptanalytic problem at hand (key-recovery, state-recovery, differential pair-finding, etc.) as a system of equations over $\mathbb{F}_{2^8}$ involving a non-linear S-box and then solve it. However, our tool is not an equations solver strictly speaking as it does not try to solve the equations directly. Actually, given a system of equations, it runs a search for an ad-hoc solver among a particular class of solvers tailored for this system and then returns the source code of the fastest found. Once compiled, the executable program (a.k.a. the attack) can enumerate all the solutions of the system. Furthermore, the tool gives a good approximation of the expected run-time the attacks it produced, which is identical for any good instantiation of the S-box. Thus, our approach is quite different from the usual algebraic attacks based on SAT-solvers or Gröbner basis algorithms [MR02, BPW06] which require to replace the S-box by its polynomial expression and lack good bounds on their complexity.

The idea of building an automated tool to find attacks of some kind on a given primitive to alleviate the task of cryptanalyst is not new and has been successful many times. Nice

examples include the cryptanalyses of Grindhal by Peyrin [Pey07] and of RadioGatùn by Fuhr and Peyrin [FP09]: in both case a custom-made program found a truncated or symmetric differential characteristic leading to a collision. Biryukov and Nikolić designed a tool to automatically find related-key attacks in the AES [BN10], and along with Khovratovich they designed a tool to search for collision attacks on byte-oriented hash functions [KBN09]. Much earlier, Matsui designed a tool to find differential characteristics and linear approximations [Mat93]. Leurent developed a tool to find good differential paths in MD4 and MD5 [FLN07], while De cannière and Rechberger developed a tool to find good differential characteristics in SHA-1 [CR06], etc.

Our tool is somewhat generic, as it is not specialized to a particular block cipher, and we applied it to reduced-round versions of the AES, to the stream cipher LEX [Bir08a], to the message authentication code Pelican-MAC [DR05c], to the block cipher SQUARE [DKR97], to SkipJack [NSA98], etc. Some of the attacks found by this algorithm are described in next chapters. Once given a very little bit of human knowledge, our tool automatically rediscovers the best known attacks on Alpha-MAC [DR05b] and LEX. It also discovered the best known attack on Pelican-MAC (a different attack with the same complexity was independently discovered at about the same time by Dunkelman, Keller and Shamir). The tool also helped us to find the best known attack on the LEX stream cipher. The tool outputs the source code of a program performing the attack, but can also provide a somewhat human-readable description of the attack procedure, which was used in some cases to understand and describe the attacks. Furthermore, it was later used as a subcomponent of an other tool described Chapters 7 and 8 leading to the discovery of the best attacks on 7, 8 and 9 rounds for the 128, 192 and 256-bit versions of the AES respectively.

All in all, this chapter is dedicated to the description of the particular class of solvers considered in our tool and to the methods we may applied to find the fastest among it. First we will describe the kind of systems of equations studied and explain our general idea. Then we will present a restricted class of solvers based on the guess-and-determine technique and see some interesting properties they share. Finally, we will show how to extend the previous class by using the meet-in-the-middle technique.

2.1 Guess-and-Determine Solvers

Let begin by properly define the kind of systems of equations studied in this chapter. Given a finite field $\mathbb{F}_q$, where q is a power of a prime number, and a non-linear function $S : \mathbb{F}_q \longrightarrow \mathbb{F}_q$, we define an AES-like equation as follows:

Definition 1 (AES-like equation)**.** *An AES-like equation in variables* $\mathbf{X} = \{x_1, \ldots, x_n\}$ *is an equation of the form:*

$$\sum_{i=1}^{n} a_i x_i + \sum_{i=1}^{n} b_i S(x_i) + c = 0,$$

where $a_1, \ldots, a_n, b_1, \ldots, b_n, c \in \mathbb{F}_q$.

AES-like equations enjoy some very interesting properties. First the set of all the AES-like equations in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$ is a vector space over $\mathbb{F}_q$. Indeed, this

set is stable by the multiplication by a scalar and the sum of two AES-like equations is still an AES-like equation.

Definition 2 (AES-like system). *We denote by $\mathcal{V}(\mathbf{X})$ the vector space spanned by all the AES-like equations in variable $\mathbf{X}$.*

As a consequence a system of AES-like equations is a subspace of $\mathcal{V}(\mathbf{X})$ and this chapter is dedicated to solving such systems. This structure allows us to use some of the resolution techniques used in linear algebra. Indeed, faced to a system of equations (possibly describing a cryptographic problem), the most naive way to obtain its solutions consists in enumerating all the possible values of the variables and retaining only the ones satisfying all the equations. But in the case of a system of linear equations, the best method is to first put the system into an echelon form by performing a Gaussian elimination and then enumerate all the solutions (linearly in the number of solutions). For instance, suppose the goal is to find and describe the set of solutions of the following system of linear equations:

$$\begin{cases} x + y - z = a \\ 2x + y + z = b \end{cases}, \text{where } a \text{ and } b \text{ are some constants.}$$

Then performing a Gaussian elimination leads to the equivalent system:

$$\begin{cases} x + y - z = a \\ \quad\; y - 3z = 2a - b \end{cases}$$

We note that there is a first (polynomial) step where the system of equations is refined independently of the two constants a and b and then, for each values of the constants, we can enumerate all the solutions of the system linearly (in the number of solutions). This resolution process is well-adapted to cryptanalysis because a and b may be seen as the information owned the adversary when she will run the attack. For instance a and b may be a plaintext and its corresponding ciphertext and as a consequence the attack will work whatever the couple in his possession. Our idea is to apply a similar resolution scheme on systems of AES-like equations.

2.1.1 Adapting the Gaussian Elimination

Let $\mathbb{E}$ be (the vector space spanned by) a system of AES-like equations in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$. Our first goal is to put this system into an "echelon form" in order to build a fast solver to enumerate all the solutions of $\mathbb{E}$ (denoted $\mathcal{S}ol(\mathbb{E})$). We stress that there is no reason for $\mathcal{S}ol(\mathbb{E})$ to be a vector space. The classical method to echelonize the system is to eliminate the variables one by one and therefore we need to be able to extract from the system $\mathbb{E}$ the biggest subsystem in any subset $\mathbf{Y}$ of $\mathbf{X}$. The most natural way to achieve this operation is to perform the intersection of the two subspaces $\mathbb{E}$ and $\mathcal{V}(\mathbf{Y})$, leading to the following definition:

Definition 3 (subsystem). *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and let $\mathbf{Y}$ be a subset of $\mathbf{X}$. We denote by $\mathbb{E}(\mathbf{Y})$ the subspace $\mathbb{E} \cap \mathcal{V}(\mathbf{Y})$. This subspace is the biggest subsystem of $\mathbb{E}$ composed of AES-like equations in variables $\mathbf{Y}$.*

Indeed, by definition $\mathcal{V}(\mathbf{Y})$ is the subspace of all AES-like equations in variables $\mathbf{Y}$ so $\mathbb{E}(\mathbf{Y})$ is obviously the biggest subspace of $\mathbb{E}$ in variables $\mathbf{Y}$ that we can reach by performing linear combinations on vectors of $\mathbb{E}$. However, if we replace the function S by its polynomial expression then a Gröbner basis of $\mathbb{E}$ for a well-chosen elimination order may contain more polynomials in variables $\mathbf{Y}$ than $\dim \mathbb{E}(\mathbf{Y})$, leading to more equations than contained in $\mathbb{E}(\mathbf{Y})$. But first those equations will probably not be AES-like equations, then they do depend on the choice of the function S, and finally obtaining them may be a very hard task depending on the number of variables and on the degree of S.

Now it is natural to consider the following sequence of subspaces:

$$\mathbb{E}(\varnothing) \subseteq \mathbb{E}(\{x_1\}) \subseteq \mathbb{E}(\{x_1, x_2\}) \subseteq \ldots \subseteq \mathbb{E}(\{x_1, \ldots, x_{n-1}\}) \subseteq \mathbb{E}(\{x_1, \ldots, x_n\}) = \mathbb{E}.$$

Performing a Gaussian elimination on the system $\mathbb{E}$ is equivalent to building a basis of $\mathbb{E}$ by first taking a basis of $\mathbb{E}(\varnothing)$, then by completing it into a basis of $\mathbb{E}(\{x_1\})$, then into a basis of $\mathbb{E}(\{x_1, x_2\})$, and so on. Thus, the first step is to look at $\mathbb{E}(\varnothing) = \mathbb{E} \cap \mathcal{V}(\varnothing)$. The vector space $\mathcal{V}(\varnothing)$ is the set of equations with the form $0 = c$ where c is a known constant or a parameter that will be required to run the solver as for instance the variables representing the plaintexts and the ciphertexts (in the sequel we will just say parameter). If $\mathbb{E}(\varnothing) \neq \{0 = 0\}$ then it contains such an equation with c being a non-zero constant or a parameter. In the first case the system has no solution so we can stop the analysis here. In the second case we replace c by 0 in the system of equations and add to the solver this condition to check. So without loss of generality we can assume that $\mathbb{E}(\varnothing) = \{0 = 0\}$. Now let assume that we know how to solve $\mathbb{E}(\{x_1, \ldots, x_k\})$ for some constant $k \in \{0, \ldots, n-1\}$. In that case we can solve $\mathbb{E}(\{x_1, \ldots, x_{k+1}\})$ by first enumerating all the solutions of $\mathbb{E}(\{x_1, \ldots, x_k\})$ and then by computing for each of them all the possible values of the variable x_{k+1} such that $(x_1, \ldots, x_{k+1})$ is a solution of $\mathbb{E}(\{x_1, \ldots, x_{k+1}\})$. As the assumption is obviously true for $k = 0$, this algorithm allows us to solve the system $\mathbb{E}$.

System of Linear Equations. If the system $\mathbb{E}$ is linear then it is known that for any $k \in \{1, \ldots, n-1\}$, $\dim \mathbb{E}(\{x_1, \ldots, x_{k+1}\}) - \dim \mathbb{E}(\{x_1, \ldots, x_k\}) \in \{0, 1\}$ depending on whether there is or not an equation involving x_{k+1} and (eventually) some variables of $\{x_1, \ldots, x_k\}$. More precisely, at each step the variable x_{k+1} is either guessed or uniquely determined by the previous ones. Furthermore, the number of guessed variables does not depend on the order we chose for $\mathbf{X}$ as we have the equation:

$$\begin{aligned} \sum_{k=1}^{n} \dim \mathbb{E}(\{x_1, \ldots, x_k\}) - \dim \mathbb{E}(\{x_1, \ldots, x_{k-1}\}) &= \dim \mathbb{E}(\{x_1, \ldots, x_n\}) - \dim \mathbb{E}(\varnothing) \\ &= \dim \mathbb{E}. \end{aligned} \tag{2.1}$$

Indeed, if we denote by n_i the number of variables x_k such that

$$\dim \mathbb{E}(\{x_1, \ldots, x_k\}) - \dim \mathbb{E}(\{x_1, \ldots, x_{k-1}\}) = i,$$

then we have $n_0 + n_1 = n$ and the above equation implies $n_0 = n - \dim \mathbb{E}$ which is constant (for a given system of equations).

System of AES-like Equations. Things are a bit different for systems of AES-like equations and the following property is the main reason:

Property 1. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$. If $x \in \mathbf{X}$ and $\mathbf{Y} = \mathbf{X} - \{x\}$, $\mathcal{V}(\mathbf{X}) = \langle \mathcal{V}(\mathbf{Y}), x = 0, S(x) = 0 \rangle$ and so $\dim \mathcal{V}(\mathbf{X}) = \dim \mathcal{V}(\mathbf{Y}) + 2$. As a consequence*

$$\dim \mathbb{E}(\mathbf{X}) - \dim \mathbb{E}(\mathbf{Y}) \in \{0, 1, 2\}.$$

Two equations like $x + f(\mathbf{Y}) = 0$ and $S(x) + g(\mathbf{Y})$ are linearly independent and thus the elimination of a variable may result in the loss of two equations instead of one in the case of linear equations. This directly affects the number of variables guessed during the solving process because Equation 2.1 now implies $n_1 + 2n_2 = \dim \mathbb{E}$ so $n_0 = n + n_2 - \dim \mathbb{E}$ which may vary. In particular, modifying the order in which variables are enumerated may increase the number of variables to guess, potentially resulting in an increase of the time complexity. It turns out that our set of solvers for $\mathbb{E}$ will be identified to the set of all the permutation of $\mathbf{X}$. But before that we have to understand how a variable can be deduced from other ones. There is two cases to consider:

i) $\underline{\dim \mathbb{E}(\{x_1, \dots, x_k\}) - \dim \mathbb{E}(\{x_1, \dots, x_{k-1}\}) = 1}$: there is an AES-like equation

$$f(x_k) = g(x_1, \dots, x_{k-1}),$$

with $f : x \mapsto \alpha x + \beta S(x)$ where α or β is non-zero. For each value of $(x_1, \dots, x_{k-1})$, the variable x_k can assume any value of the set $f^{-1}(g(x_1, \dots, x_{k-1}))$. Note that this set may contain more or less than one element since the function f is not necessary one-to-one. The inverse of the function f can be precomputed and stored in a hash table to access each solution in constant time. As a result, finding all the solutions of $\mathbb{E}(\{x_1, \dots, x_k\})$ from the ones of $\mathbb{E}(\{x_1, \dots, x_{k-1}\})$ requires exactly $|\mathcal{S}ol\,(\mathbb{E}(\{x_1, \dots, x_{k-1}\}))\,|$ evaluations of g, as much lookup in the hash table and the reading of $|\mathcal{S}ol\,(\mathbb{E}(\{x_1, \dots, x_k\}))\,|$ solutions.

ii) $\underline{\dim \mathbb{E}(\{x_1, \dots, x_k\}) - \dim \mathbb{E}(\{x_1, \dots, x_{k-1}\}) = 2}$: there are two AES-like equations

$$x_k = g_1(x_1, \dots, x_{k-1}), \; S(x_k) = g_2(x_1, \dots, x_{k-1}).$$

For each value of $(x_1, \dots, x_{k-1})$, the first equation is used to uniquely determine the value of variable x_k and then $(x_1, \dots, x_k)$ is a solution if and only if the second one holds. In particular, $|\mathcal{S}ol\,(\mathbb{E}(\{x_1, \dots, x_k\}))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\{x_1, \dots, x_{k-1}\}))\,|$.

All in all, this leads to the following definition of a solver:

Definition 4 (class of guess-and-determine solvers)**.** *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$. The class of solvers considered to solve $\mathbb{E}$ is identified to the set of all the permutations of $\mathbf{X}$: given such a permutation σ, we associate to it the solver corresponding to the chain:*

$$\mathbb{E}(\{\sigma(1)\}) \subseteq \mathbb{E}(\{\sigma(1), \sigma(2)\}) \subseteq \dots \subseteq \mathbb{E}(\{\sigma(1), \dots, \sigma(n-1)\}) \subseteq \mathbb{E}(\{\sigma(1), \dots, \sigma(n)\}) = \mathbb{E}.$$

We denote its time complexity by T_σ which is approximated by:

$$T \approx \max(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1)\}))\,|, |\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1), \sigma(2)\}))\,|, \dots, |\mathcal{S}ol\,(\mathbb{E})\,|).$$

Note that we omit the time spent to inverse the functions $x \mapsto \alpha x + \beta S(x)$ because they can be precomputed and only q of them have to be processed, bounding the complexity of this part to q^2 elementary operations and approximately $q^2 \log_2(q)$ bits of storage. Indeed, for any non-zero λ and any subset X of $\mathbb{F}_q$, we know that:

$$(x \mapsto \lambda\alpha x + \lambda\beta S(x))^{-1}(X) = (x \mapsto \alpha x + \beta S(x))^{-1}(\lambda^{-1}X).$$

As a consequence, we only have to process the functions such that $\alpha = 1$ and $\beta \neq 0$ (the case $(\alpha, \beta) = (1, 0)$ is obviously not required) or $\alpha = 0$ and $\beta = 1$. As the value of q considered in practice is 256, the complexity of this part is negligible.

Assumption on the Number of Solutions

Computing the complexity of the solver requires evaluating the number of solutions of various subsystems. This is a difficult problem in general, and in order to be able to quickly evaluate it, we use the following *heuristic assumption*:

$$|\mathcal{S}ol\,(\mathbb{E}(\mathbf{Y}))| \approx q^{|\mathbf{Y}|-\dim \mathbb{E}(\mathbf{Y})}, \text{ for any subset } \mathbf{Y} \text{ of } \mathbf{X}.$$

This heuristic assumption comes from the intuition that any function $f(x_1, \ldots, x_k) = \sum_{i=1}^{k} a_i x_i + b_i S(x_i)$ reaches each element of $\mathbb{F}_q$ almost as much times. Given two sequences $(a_n)_{n\in\mathbb{N}}$ and $(b_n)_{n\in\mathbb{N}}$ it is easy to show that for any k:

$$\max_{c\in\mathbb{F}_q} \text{Prob}\left(\sum_{i=1}^{k+1} a_i x_i + b_i S(x_i) = c\right) \leqslant \max_{c\in\mathbb{F}_q} \text{Prob}\left(\sum_{i=1}^{k} a_i x_i + b_i S(x_i) = c\right),$$

where the x_i's are pick randomly, following a uniform distribution.

Indeed, let $p_{1,c} = \text{Prob}\left(\sum_{i=1}^{k} a_i x_i + b_i S(x_i) = c\right)$ and $p_{2,c} = \text{Prob}\left(a_{k+1} x_{k+1} + b_{k+1} S(x_{k+1}) = c\right)$. Then for any $\alpha \in \mathbb{F}_q$ we have $\text{Prob}\left(\sum_{i=1}^{k+1} a_i x_i + b_i S(x_i) = \alpha\right) = \sum_{c\in\mathbb{F}_q} p_{1,\alpha-c} p_{2,c}$ leading to the expected result:

$$\sum_{c\in\mathbb{F}_q} p_{1,\alpha-c} p_{2,c} \leqslant \max_{c\in\mathbb{F}_q} p_{1,\alpha-c} \sum_{c\in\mathbb{F}_q} p_{2,c} = \max_{c\in\mathbb{F}_q} p_{1,c}.$$

As a consequence our assumption makes sense as long as the equation $ax + bS(x) = c$ doesn't have too many solutions whatever the value of $\{a, b, c\}$ (excepted $\{a, b\} = \{0, 0\}$). However it introduces a risk of failure, or of wrong estimation of the complexity. In particular, a difficulty that we encountered in practice stems from the following "differential" system:

$$\left\{\begin{array}{rcl} x - y & = & \Delta_i \\ S(x) - S(y) & = & \Delta_o \end{array}\right. .$$

If S is the S-box of the AES, then this system has one solution on average (over the random choice of the differences), and the hypothesis holds. However, in degenerate situations, for instance when $\Delta_i = \Delta_o = 0$, then the system has 2^8 solutions... Surprisingly, an S-box with very bad differential properties would make life more difficult for us. This follows

from the fact that on a good S-box, there are very few pairs of input/output values that generate a given input/output difference, and this makes our assumption more likely to hold in "differential" situations. However this case is easily handled in practice: if there is an equation like $x - y = 0$ then we will replace each occurrence of y and $S(y)$ by x and $S(x)$ respectively in the system of equations. More generally, if there is an equation like $ax + bS(x) - ay - bS(y) = c$ then the three cases $\{c \neq 0\}$, $\{c = 0, x = y\}$ and $\{c = 0, x \neq y\}$ are considered. A more vicious case failing the heuristic assumption comes from over-determined systems that leads to a non-integer number of solutions. In itself it is not a real issue since in that case $q^{|\mathbf{Y}|-\dim \mathbb{E}(\mathbf{Y})}$ can be seen as the probability that there is a solution. But it may lead to pathological cases such that:

$$\begin{cases} x &= a \\ S(x) &= b \\ y + z &= c \end{cases} .$$

This system has three equations in three variables so we expect 1 solution. But in fact this system has exactly q solutions if $S(a) = b$ and 0 otherwise. Such cases are more delicate to handle without computing $\dim \mathbb{E}(\mathbf{Y})$ for each subset $\mathbf{Y}$ of $\mathbf{X}$. Indeed, if we echelonize this system according to the chain:

$$\mathbb{E}(\{x\}) \subseteq \mathbb{E}(\{x, y\}) \subseteq \mathbb{E}(\{x, y, z\}),$$

then we will see that $|\{x\}| - \dim \mathbb{E}(\{x\}) < 0$ and we will take an appropriate action. But if we consider the chain:

$$\mathbb{E}(\{y\}) \subseteq \mathbb{E}(\{x, y\}) \subseteq \mathbb{E}(\{x, y, z\}),$$

then no problems will be detected leading to a possible wrong estimation of the time complexity. When such a problem is encountered for a subsystem $\mathbb{E}(\mathbf{Y})$, our idea is to stop the analysis at this point and to restart an analysis of $\mathbb{E}$ but where the variables $\mathbf{Y}$ will be seen as parameters (denoted by $\mathbb{E}_{\mathbf{Y}}$). Then, by combining the solver for $\mathbb{E}(\mathbf{Y})$ and the one for $\mathbb{E}_{\mathbf{Y}}$, we will obtain a solver for the whole system $\mathbb{E}$. In the sequel we will assume that this pathological case never happens.

In any case, this assumption makes it very easy to evaluate the time complexity of a solver: it boils down to computing a vector-space intersection.

A Toy Example

To fully understand how works the solver, let study the following system of equations:

$$\begin{cases} x + y + S(y) + z - S(z) + t + S(t) = 0 \\ S(x) + y - S(y) + z - S(z) + t + S(t) = 0 \\ x + S(x) + 2y + S(y) + 3z - 3S(z) + 2t + 3S(t) = 0 \end{cases} .$$

This system is composed of three AES-like equations involving four variables x, y, z and t. If the equations were linear and independent then a solver for such a system would have

exactly q solutions and enumerate all of them would require q operations. Let see what happens in the case of AES-like equations. First, let consider the permutation $[t, z, y, x]$ corresponding to the chain:

$$\mathbb{E}(\{t\}) \subseteq \mathbb{E}(\{z,t\}) \subseteq \mathbb{E}(\{y,z,t\}) \subseteq \mathbb{E}(\{x,y,z,t\}).$$

In that case the system is refined into:

$$\left\{\begin{array}{cccccccccccccccc} x & + & & & y & + & S(y) & + & z & - & S(z) & + & t & + & S(t) & = 0 \\ & & S(x) & + & y & - & S(y) & + & z & - & S(z) & + & t & + & S(t) & = 0 \\ \hline \\ & & & & & & S(y) & + & z & - & S(z) & & & + & S(t) & = 0 \\ \hline \\ \hline \end{array}\right. .$$

As we can see, $\mathbb{E}(\{t\}) = \mathbb{E}(\{z,t\}) = \{0 = 0\}$ so we begin by guessing the two variables z and t. Then we observe that the only equation contained in $\mathbb{E}(\{y,z,t\})$ is

$$S(y) + z - S(z) + S(t) = 0.$$

So the variable y can be *deduced* from z and t, the set of possible values for y being the set $S^{-1}(\{-z + S(z) - S(t)\})$. Finally, for each solution of $\mathbb{E}(\{y,z,t\})$ we have to find the possible values for x according to the two equations:

$$\left\{\begin{array}{rcl} x + y + S(y) + z - S(z) + t + S(t) & = & 0 \\ S(x) + y - S(y) + z - S(z) + t + S(t) & = & 0 \end{array}\right. .$$

To do so, the first equation is used to deduced x from y, z and t and then we check whether the second equation holds. This solver is described in an algorithmic manner in Algorithm 1.

Algorithm 1: Solver of the toy example system of AES-like equations

Data: A function $S : \mathbb{F}_q \longrightarrow \mathbb{F}_q$.
Result: All the solutions of the system of equations.

```
T ⟵ table of q empty sets;
forall the v ∈ F_q do                              // Inverse the function S
    T[S(v)] ⟵ T[S(v)] ∪ {v}
end
Sol ⟵ empty set;
forall the t ∈ F_q do                              // solutions of E({t})
    forall the z ∈ F_q do                          // solutions of E({z,t})
        forall the y ∈ T[−z + S(z) − S(t)] do      // solutions of E({y,z,t})
            x ⟵ −(y + S(y) + z − S(z) + t + S(t));
            if S(x) + y − S(y) + z − S(z) + t + S(t) = 0 then
                Sol ⟵ Sol ∪ {(x,y,z,t)}            // solutions of E({x,y,z,t})
            end
        end
    end
end
return Sol
```

First a table representing the inverse of the function S is constructed in order to be able to quickly deduced the possible values of y from z and t. Performing this part requires q elementary operations and approximately $q\log_2(q)$ bits to store the table. Then, the solutions of the system are computed in time T depending only on the number of solutions of the considered subsystems. Indeed, the number of operations performed is:

$$\max\left(|\mathcal{S}ol\left(\mathbb{E}(\{t\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{z,t\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{y,z,t\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{x,y,z,t\})\right)|\right).$$

According to the heuristic assumption it is equal to:

$$\max\left(q, q^2, q^2, q\right) = q^2.$$

However, this assumption is used only to determine the number of solutions of $\mathbb{E}(\{y,z,t\})$. Indeed, the two subsystems $\mathbb{E}(\{t\})$ and $\mathbb{E}(\{z,t\})$ contain the equation $0 = 0$ only and we are sure that $\mathbb{E}(\{x,y,z,t\})$ cannot have more solutions than $\mathbb{E}(\{y,z,t\})$. As a consequence $T = \max\left(q^2, |\mathcal{S}ol\left(\mathbb{E}(\{y,z,t\})\right)|\right)$ and will be between q^2 and q^3 for any S-box which is higher than expected in the linear case.

Now let consider the solver obtained with the permutation $[y,x,t,z]$ corresponding to the chain:

$$\mathbb{E}(\{y\}) \subseteq \mathbb{E}(\{x,y\}) \subseteq \mathbb{E}(\{x,y,t\}) \subseteq \mathbb{E}(\{x,y,z,t\}).$$

In that case the system is refined into:

$$\left\{\begin{array}{cccccccccccccc}
z & - & S(z) & & + & S(t) & & & & & + & S(y) & = & 0 \\ \hline
 & & & t & & & & + & S(x) & + & y & - 2S(y) & = & 0 \\ \hline
 & & & & & & x & - & S(x) & & + & 2S(y) & = & 0 \\ \hline
 \\ \hline
\end{array}\right. .$$

The corresponding solver begins by guessing the variable y and then the variables x, t and z are deduced successively. Its complexity is

$$\begin{aligned}
T &= \max\left(|\mathcal{S}ol\left(\mathbb{E}(\{y\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{x,y\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{x,y,t\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{x,y,z,t\})\right)|\right) \\
&= \max\left(q, |\mathcal{S}ol\left(\mathbb{E}(\{z,t\})\right)|, |\mathcal{S}ol\left(\mathbb{E}(\{x,y,z,t\})\right)|\right)
\end{aligned}$$

because $\mathbb{E}(\{y\}) = \{0 = 0\}$ and $|\mathcal{S}ol\left(\mathbb{E}(\{x,y\})\right)| = |\mathcal{S}ol\left(\mathbb{E}(\{x,y,t\})\right)|$. According to the heuristic assumption we have $T = q$ and for instance if the function $v \mapsto v - S(v)$ is a bijection this complexity is reached. In that case this solver will be faster than the previous one, proving that the complexity of such solvers does depend on the permutation used to echelonize the system. So now our goal is clearly identified: find the permutation corresponding to the fastest solver.

2.1.2 Finding the Best Solver

The most naive way to find the best solver is to try all the permutations. But $|\mathbf{S}_n| = n!$ and for each permutation computing its complexity require computing the dimension of n subspaces. Furthermore those subspaces are obtained by performing the intersection between $\mathbb{E}$ and various subspaces. As a consequence such an algorithm requires approximately $(n+1)!$ non-elementary operations and only systems involving a very small number of variables can be handled in practice. However it is possible to improve this algorithm by using some simple observations. Actually, there are two possible methods to tackle the problem depending on whether it is seen as finding the best guess-and-determine attacks or as finding the best way to echelonize the system. Indeed, a bottom-up approach seems more adapted to the first case while a top-down approach is the most natural in the second one. For each of them we will show how to reduce the search space and describe an algorithm that solves the problem.

From the Bottom

Let begin by the bottom-up approach. It is natural to think that an optimal solver can be constructed by first guessing a variable, then deducing all the variables we can, then guessing a new one and so on until we obtain all of them. More generally, given a subset of the variables we are interested by the variables that can be deduced from it, leading to the following definition:

Definition 5 (Propagate). *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and let be $\mathbf{Y} \subseteq \mathbf{X}$. We say that a variable x can be deduced directly from $\mathbf{Y}$ if and only if $x \in \mathbf{Y}$ or $\dim \mathbb{E}(\mathbf{Y} \cup \{x\}) - \dim \mathbb{E}(\mathbf{Y}) \geqslant 1$ (i.e., there is at least one equation involving the variable x and (possibly) some of $\mathbf{Y}$). We denote by $\text{Propagate}_{\mathbb{E}}(\mathbf{Y})$ the set of all such variables.*

In the sequel we will omit to specify the system of equations and write only $\text{Propagate}(\mathbf{Y})$. The variable x is said to be deduced from $\mathbf{Y}$ because according to the heuristic assumption on the number of solutions, we have the equivalence:

$$x \in \text{Propagate}(\mathbf{Y}) \Longleftrightarrow |\mathcal{S}ol(\mathbb{E}(\mathbf{Y} \cup \{x\}))| \leqslant |\mathcal{S}ol(\mathbb{E}(\mathbf{Y}))|.$$

The operator Propagate has some interesting properties. First it is fairly easy to see that it is monotonic:

$$\mathbf{Y} \subseteq \mathbf{Z} \Longrightarrow \text{Propagate}(\mathbf{Y}) \subseteq \text{Propagate}(\mathbf{Z}).$$

Furthermore, and as the time complexity of a solver only depends on the number of solutions of subsystems, we can build from a solver σ of a subsystem $\mathbb{E}(\mathbf{Y})$ a solver for $\mathbb{E}(\text{Propagate}(\mathbf{Y}))$ denoted $\text{Propagate}(\sigma)$ such that $T_{\text{Propagate}(\sigma)} = T_\sigma$.

Property 2. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$, $\mathbf{Y}$ be a subset of $\mathbf{X}$ and $\mathbf{Z} = \text{Propagate}(\mathbf{Y}) - \mathbf{Y}$. Let $\sigma_\mathbf{Y}$ be a permutation of $\mathbf{Y}$. Then, for any permutation $\sigma_\mathbf{Z}$ of $\mathbf{Z}$, the permutation $\sigma = \sigma_\mathbf{Y} \parallel \sigma_\mathbf{Z}$ of $\text{Propagate}(\mathbf{Y})$ is such that*

$$|\mathcal{S}ol(\mathbb{E}(\{\sigma(1), \dots, \sigma(k)\}))| \geqslant |\mathcal{S}ol(\mathbb{E}(\{\sigma(1), \dots, \sigma(k+1)\}))|,$$

for all $k \in \{|\mathbf{Y}|, \dots, |\text{Propagate}(\mathbf{Y})| - 1\}$. In particular,

$$T_\sigma = T_{\sigma_\mathbf{Y}} \text{ and } |\mathcal{S}ol(\text{Propagate}(\mathbf{Y}))| \leqslant |\mathcal{S}ol(\mathbf{Y})|.$$

Here a permutation is seen as an ordered sequence and then $\sigma = \sigma_\mathbf{Y} \parallel \sigma_\mathbf{Z}$ means that $\sigma(k)$ is equal to $\sigma_\mathbf{Y}(k)$ (resp. $\sigma_\mathbf{Z}(k - |\mathbf{Y}|)$) for any $k \leqslant |\mathbf{Y}|$ (resp. $k > |\mathbf{Y}|$). The definition of $\text{Propagate}(\sigma)$ seems unclear as we have many choices for it but makes sense as all the choices for $\text{Propagate}(\sigma)$ have the same time complexity and solve the same subsystem. Indeed it is fairly easy to show the following property:

Property 3. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and $\mathbf{Y}$ be a subset of $\mathbf{X}$. Let σ and σ' be two permutations of $\mathbf{Y}$ such that $T_\sigma \leqslant T_{\sigma'}$. Then, for any permutation σ'' of $\mathbf{X} - \mathbf{Y}$, we have $T_{\sigma\parallel\sigma''} \leqslant T_{\sigma'\parallel\sigma''}$.*

Proof. By definition, we have:
- $T_{\sigma\parallel\sigma''} = \max(T_\sigma, |\mathcal{S}ol(\mathbb{E}(\mathbf{Y} \cup \{\sigma''(1)\}))|, \dots, |\mathcal{S}ol(\mathbb{E})|)$
- $T_{\sigma'\parallel\sigma''} = \max(T_{\sigma'}, |\mathcal{S}ol(\mathbb{E}(\mathbf{Y} \cup \{\sigma''(1)\}))|, \dots, |\mathcal{S}ol(\mathbb{E})|)$

Thus, $T_\sigma \leqslant T_{\sigma'}$ implies $T_{\sigma\parallel\sigma''} \leqslant T_{\sigma'\parallel\sigma''}$. □

A natural extension of this property is to expect that if a solver enumerates as fast the solutions of a bigger subsystem than an other solver then we could use it instead. While it seems to hold in practice, unfortunately it does not in general and Theorem 1 is the best we can get.

Theorem 1. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and $\mathbf{Y}, \mathbf{Z}$ be two subsets of $\mathbf{X}$ such that $\mathbf{Y} \subseteq \mathbf{Z}$. Let $\sigma_{\mathbf{Y}}$, $\sigma_{\mathbf{Z}}$ and $\sigma_{\mathbf{X}-\mathbf{Y}}$ be permutations of $\mathbf{Y}$, $\mathbf{Z}$ and $\mathbf{X} - \mathbf{Y}$ respectively. Let $\sigma_{\mathbf{X}-\mathbf{Z}}$ be the permutation of $\mathbf{X} - \mathbf{Z}$ conserving the order induced by $\sigma_{\mathbf{X}-\mathbf{Y}}$. If for any subset $\mathbf{V}$ such that $\mathbf{Y} \subseteq \mathbf{V} \subseteq \mathbf{Z}$ then $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{V}))\,| \geqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{Z}))\,|$ and if $T_{\sigma_{\mathbf{Z}}} \leqslant T_{\sigma_{\mathbf{Y}}}$, then*

$$T_{[\sigma_{\mathbf{Z}} \| \sigma_{\mathbf{X}-\mathbf{Z}}]} \leqslant T_{[\sigma_{\mathbf{Y}} \| \sigma_{\mathbf{X}-\mathbf{Y}}]}.$$

In particular, if there is an optimal solver going through $\mathbf{Y}$ then there is one going through $\mathbf{Z}$.

The proof is fairly easy but notations introduced in this section make it ugly. However a proof of a more general theorem is given Section 2.2.3.0. It relies on the following lemma about the evolution of the number of solutions:

Lemma 1. *Let $\mathbf{X}_1$, $\mathbf{X}_2$ and $\mathbf{X}_3$ be three subsets of $\mathbf{X}$ such that:*

i) $\mathbf{X}_2 \subseteq \mathbf{X}_1$

ii) $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2))\,|$

iii) $\mathbf{X}_1 \cap \mathbf{X}_3 = \mathbf{X}_2 \cap \mathbf{X}_3$

Then $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3))\,|$. In addition, if the inequality in ii) is strict, then the resulting inequality is also strict.

Proof. Let us prove the case where $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2))\,|$, the other being treated in the same way. According to the heuristic assumption on the number of solutions we have:

$$|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3))\,|$$
$$\Longleftrightarrow$$
$$|\mathbf{X}_1 \cup \mathbf{X}_3| - |\mathbf{X}_2 \cup \mathbf{X}_3| \leqslant \dim \mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3) - \dim \mathbb{E}(\mathbf{X}_2 \cup \mathbf{X}_3).$$

In an other hand, as $\mathbf{X}_1 \cap \mathbf{X}_3 = \mathbf{X}_2 \cap \mathbf{X}_3$ and $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2))\,|$, we obtain

$$|\mathbf{X}_1 \cup \mathbf{X}_3| - |\mathbf{X}_2 \cup \mathbf{X}_3| = |\mathbf{X}_1| - |\mathbf{X}_2| \leqslant \dim \mathbb{E}(\mathbf{X}_1) - \dim \mathbb{E}(\mathbf{X}_2).$$

Consequently, it remains to prove that $\dim \mathbb{E}(\mathbf{X}_2 \cup \mathbf{X}_3) - \dim \mathbb{E}(\mathbf{X}_2) \leqslant \dim \mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3) - \dim \mathbb{E}(\mathbf{X}_1)$ to establish the result. In other words, we have to prove that there are more equations in $\mathbb{E}$ involving at least one variable of $\mathbf{X}_3$ and possibly some of $\mathbf{X}_1$ than equations involving at least one variable of $\mathbf{X}_3$ and possibly some of $\mathbf{X}_2$ which is trivial since $\mathbf{X}_2 \subseteq \mathbf{X}_1$. □

The condition about the number solutions of the intermediate subsytems may be hard to check in practice and the interest of this theorem is mainly theoretical. However, it allows us to show the following useful property about Propagate:

Property 4. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$ and let σ be a permutation of $\mathbf{X}$. Let $k, l \in \{1, \ldots, n\}$ such that $k < l$ and let σ' be the permutation defined as follows:*

- $\sigma'(i) = \sigma(i)$ *for* $1 \leqslant i < k$.
- $\sigma'(k) = \sigma(l)$.

– $\sigma'(i) = \sigma(i-1)$ *for* $k < i \leqslant l$.
– $\sigma'(i) = \sigma(i)$ *for* $l < i \leqslant n$.

Then $\sigma(l) \in \textsc{Propagate}(\{\sigma(1), \ldots, \sigma(k-1)\})$ *implies* $T_{\sigma'} \leqslant T_\sigma$. *In particular, if* σ *leads to an optimal solver then* σ' *too.*

Proof. The proof of this theorem is very simple. By definition and according to the heuristic assumption on the number of solutions, we have:

$$\begin{cases} T_\sigma &= \max(q^{1-\dim \mathbb{E}(\{\sigma(1)\})}, q^{2-\dim \mathbb{E}(\{\sigma(1),\sigma(2)\})}, \ldots, q^{n-\dim \mathbb{E}}) \\ T_{\sigma'} &= \max(q^{1-\dim \mathbb{E}(\{\sigma'(1)\})}, q^{2-\dim \mathbb{E}(\{\sigma'(1),\sigma'(2)\})}, \ldots, q^{n-\dim \mathbb{E}}) \end{cases}.$$

If $\sigma(l) \in \textsc{Propagate}(\{\sigma(1), \ldots, \sigma(k-1)\})$ then $\sigma(l) \in \textsc{Propagate}(\{\sigma(1), \ldots, \sigma(i)\})$ for any $i \geqslant k-1$ since $\textsc{Propagate}$ is monotonic, and as a consequence,

$$\dim \mathbb{E}(\{\sigma(1), \ldots, \sigma(i), \sigma(l)\}) - \dim \mathbb{E}(\{\sigma(1), \ldots, \sigma(i)\}) \leqslant 1.$$

But by definition $\{\sigma(1), \ldots, \sigma(i), \sigma(l)\} = \{\sigma'(1), \ldots, \sigma'(i+1)\}$ and then

$$i + 1 - \dim \mathbb{E}(\{\sigma'(1), \ldots, \sigma'(i+1)\}) \geqslant i - \dim \mathbb{E}(\{\sigma(1), \ldots, \sigma(i)\}).$$

By combining this to the fact that $\sigma(\{1, \ldots, i\}) = \sigma'(\{1, \ldots, i\})$ for any $i \in \{1, \ldots, k-1, l+1, \ldots, n\}$ we easily obtain $T_{\sigma'} \leqslant T_\sigma$. □

In other words, this theorem means that to find an optimal solver we can only consider the permutations such that for each $k \in \{1, \ldots, n\}$ either $\sigma(k) \in \textsc{Propagate}(\{\sigma(1), \ldots, \sigma(k-1)\})$ or $\textsc{Propagate}(\{\sigma(1), \ldots, \sigma(k-1)\}) = \{\sigma(1), \ldots, \sigma(k-1)\}$. In particular we should begin by the variables belonging to $\textsc{Propagate}(\varnothing)$ which is the set of the variables that can be deduced from the parameters. More precisely, $\textsc{Propagate}(\varnothing)$ contains a variable x if and only if $\dim \mathbb{E}(\{x\}) > 0$ so that we know at least an equation involving only x, $S(x)$ and eventually some parameters. To be consistent with the heuristic assumption on the number of solutions it is assumed that x takes a single value that will be computed at the beginning of the solving process in a negligible time, and thus x can be seen as a parameter. So without loss of generality we can assume that $\dim \mathbb{E}(\{x_k\}) = 0$ for all $k \in \{1, \ldots, n\}$.

Let come back on $\textsc{Propagate}$. By definition $\mathbf{Y} \subseteq \textsc{Propagate}(\mathbf{Y})$ which combined to the fact that $\textsc{Propagate}$ is monotonic implies that the sequence $(\textsc{Propagate}^n(\mathbf{Y}))_{n\in\mathbb{N}}$ is increasing. As there is only a finite number of subsets of $\mathbf{X}$, this is a stationary sequence (from a particular rank n_0). Thus the set $\textsc{Propagate}^{n_0}(\mathbf{Y})$ contains all the variables deduced from $\mathbf{Y}$ and we denote it by $\textsc{Propagate}^\star(\mathbf{Y})$. As a result the sequence $(\textsc{Propagate}^n(\sigma))_{n\in\mathbb{N}}$ is also constant from the same rank allowing us to uniquely define the solver $\textsc{Propagate}^\star(\sigma)$. This solver has similar properties with $\textsc{Propagate}(\sigma)$:

Property 5. *Let* $\mathbb{E}$ *be a system of AES-like equations in variables* $\mathbf{X}$*,* $\mathbf{Y}$ *be a subset of* $\mathbf{X}$ *and* $\mathbf{Z} = \textsc{Propagate}^\star(\mathbf{Y})$. *Let* $\sigma_\mathbf{Y}$ *be a permutation of* $\mathbf{Y}$ *and* $\sigma = \textsc{Propagate}^\star(\sigma_\mathbf{Y})$. *Then* $T_\sigma = T_{\sigma_\mathbf{Y}}$ *and* $|\mathcal{S}ol(\textsc{Propagate}^\star(\mathbf{Y}))| \leqslant |\mathcal{S}ol(\mathbf{Y})|$.

Finally, and related to Property 4, the following property shows how the move of one variable affects the complexity of a solver:

Property 6. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$ and let σ be a permutation of $\mathbf{X}$. Let $k, l \in \{1, \ldots, n\}$ such that $k \leqslant l$ and let σ' be the permutation defined as follows:*
- $\sigma'(i) = \sigma(i)$ *for* $1 \leqslant i < k$.
- $\sigma'(k) = \sigma(l)$.
- $\sigma'(i) = \sigma(i-1)$ *for* $k < i \leqslant l$.
- $\sigma'(i) = \sigma(i)$ *for* $l < i \leqslant n$.

Then for any $i \in \{1, \ldots, n\}$,

$$|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1), \ldots, \sigma(i)\}))\,| - |\mathcal{S}ol\,(\mathbb{E}(\{\sigma'(1), \ldots, \sigma'(i)\}))\,| \in \{-1, 0, 1\}.$$

In particular, $\log_q(T_{\sigma'}) - \log_q(T_\sigma) \in \{-1, 0, 1\}$.

All in all, our combination of those results leads to Algorithm 2. It takes as input a system of equations $\mathbb{E}$ in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$ and an integer $t_{up} \geqslant 1$, and returns a solver for $\mathbb{E}$ with a time complexity equals to at most $q^{t_{up}}$ if any exists. The complexity of this algorithm is hard to evaluate but if the case $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{Z}))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{Y}))\,|$ never occurs then it only looks for a minimal set of variables sufficient to deduce the other ones, bounding the complexity by $t_{up}\binom{|\mathbf{X}|}{t_{up}}$ applications of PROPAGATE*.

From the Top

The theorems we just saw are well-adapted to a bottom-up approach which is usual when looking for guess-and-determine attacks. However systems of linear equations are commonly handled by a top-down approach and it could be powerful for system of AES-like equations as well. Actually we will see that both approaches are quite similar. First let define a particular class of variables called the *linear variables*:

Definition 6 (linear variable)**.** *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and let be $x \in \mathbf{X}$. The variable x is a linear variable if and only if $\dim \mathbb{E} - \dim \mathbb{E}(\mathbf{X} - \{x\}) \leqslant 1$. The set of all the linear variables is denoted by $\text{LIN}\,(\mathbb{E})$. Furthermore, for each subset $\mathbf{Y}$ of $\mathbf{X}$ the set $\text{LIN}\,(\mathbb{E}(\mathbf{X} - \mathbf{Y})) \cup \mathbf{Y}$ is denoted by $\text{LIN}\,(\mathbf{Y})$.*

This definition may seem abstract and the following proposition clarifies it:

Property 7. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and let $x \in \text{LIN}\,(\mathbb{E})$. Then it exists $(a, b) \in \mathbb{F}_q^2$ such that each equation of $\mathbb{E}$ involving the variable x involves in fact a multiple of $ax + bS(x)$. In other words, if we replace $ax + bS(x)$ by X in the system of equations then x and $S(x)$ do not appear any more. In particular, $\text{LIN}\,(\mathbb{E}) \cap \mathbf{Y} \subseteq \text{LIN}\,(\mathbb{E}(\mathbf{Y}))$ for any subset $\mathbf{Y}$ of $\mathbf{X}$.*

This class of variables is interesting because actually the operator LIN acts exactly as PROPAGATE but for an other measurement of the time complexity. Let $\mathbf{Y}$ be a subset of $\mathbf{X}$ and σ a permutation of $\mathbf{Y}$. We saw that σ can be seen as a solver for $\mathbb{E}(\mathbf{Y})$ and its

Algorithm 2: Exhaustive search of an optimal solver (from the bottom)

```
Function BottomSearch(𝔼,t_up)
    Data: A system of equations 𝔼 in variables X and an upper bound t_up
    Result: A permutation of X leading to a solver with time complexity equals to
            at most q^{t_up} if any and ∅ otherwise.
    𝔼 ⟵ 𝔼 without equations between the parameters;
    if Propagate(∅) = ∅ then
        return tmp-Bottom(𝔼, ∅, ∅, ∅, t_up)
    else
        σ ⟵ a permutation of Propagate(∅);
        return σ ‖ BottomSearch(𝔼_{Propagate(∅)}, t_up)
    end
end

Function tmp-Bottom(𝔼,Y,G_not,σ,t_up)
    Data:
    – a system of equations 𝔼 in variables X,
    – a subset Y of X stable by Propagate,
    – a subset G_not containing Y and the variables not allowed to be guessed,
    – a permutation σ of Y such that T_σ < q^{t_up},
    – a bound t_up ⩾ 1.
    Result: A permutation of X leading to a solver with time complexity equals to
            at most q^{t_up} and which begins by solving 𝔼(Y) according to σ if any
            and ∅ otherwise.
    if X − G_not = ∅ then return ∅;
    x ⟵ Pick one variable from X − G_not;
    Z ⟵ Propagate*(Y ∪ {x});
    σ′ ⟵ Propagate*(σ||[x]) ;
    if Z = X then return σ′;
    if |Sol(𝔼(Z))| ⩽ |Sol(𝔼(Y))| then
        if |Sol(𝔼(Z))| ⩽ 1 then return σ′ ‖ BottomSearch(𝔼_Z,t_up) ;
        if log_q(|Sol(𝔼(Y))|) = t_up − 1 then return tmp-Bottom(𝔼,Z,Z,σ′,t_up) ;
        if (Z − Y) ∩ G_not = ∅ then return tmp-Bottom(𝔼,Z,G_not ∪ Z,σ′,t_up) ;
    else
        if |Sol(𝔼(Z))| < t_up and (Z − Y) ∩ G_not = ∅ then
            σ″ ⟵ tmp-Bottom(𝔼,Z,G_not ∪ Z,σ′,t_up) ;
            if σ″ ≠ ∅ then return σ″;
        end
    end
    return tmp-Bottom(𝔼,Y,G_not ∪ Z,σ′,t_up)
end
```

complexity is defined by $T_\sigma = \max(1 - \dim \mathbb{E}(\{\sigma(1)\}), \ldots, |\mathbf{Y}| - \dim \mathbb{E}(\mathbf{Y}))$. However σ can also be seen as an *ending solver* for $\mathbb{E}$ with a time complexity defined by:

$$T^*_\sigma = \max\left(|\mathbf{X}| - \dim \mathbb{E}, |\mathbf{X}| - 1 - \dim \mathbb{E}(\mathbf{X} - \{\sigma(|\mathbf{Y}|)\}), \ldots, |\mathbf{X}| - |\mathbf{Y}| - \dim \mathbb{E}(\mathbf{X} - \mathbf{Y})\right).$$

More precisely T^*_σ is the minimal complexity that a solver of $\mathbb{E}$ ending by σ could have.

Property 8. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and $\mathbf{X}_1 \cup \mathbf{X}_2$ be a partition of $\mathbf{X}$. Then, for any permutations σ_1 of $\mathbf{X}_1$ and σ_2 of $\mathbf{X}_2$, the permutation $\sigma = \sigma_1 \parallel \sigma_2$ of $\mathbf{X}$ is such that*

$$T_\sigma = \max(T_{\sigma_1}, T^*_{\sigma_2}) = T^*_\sigma.$$

Then, given a permutation $\sigma_\mathbf{Y}$ of a subset $\mathbf{Y}$ and for $\mathbf{Z} = \textsc{Lin}(\mathbf{Y}) - Y$, for any permutation $\sigma_\mathbf{Z}$ of $\mathbf{Z}$ the permutation $\sigma = \sigma_\mathbf{Z} \parallel \sigma_\mathbf{Y}$ of $\textsc{Lin}(\mathbf{Y})$ is such that $T^*_\sigma = T^*_{\sigma_\mathbf{Y}}$. We pick one of those permutations and denote it by $\textsc{Lin}(\sigma_\mathbf{Y})$. Thanks to Property 8, this choice does not affect the complexity of any solver builds from it. Furthermore, we can show a property of $\mathcal{L}$ comparable to Property 4 concerning PROPAGATE:

Property 9. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X} = \{x_1, \ldots, x_n\}$ and let σ be a permutation of $\mathbf{X}$. Let be $k \in \{1, \ldots, n\}$ and let σ' be the permutation defined as follows:*
- $\sigma'(i) = \sigma(i)$ *for* $1 \leqslant i < k$.
- $\sigma'(i) = \sigma(i+1)$ *for* $k \leqslant i < n$.
- $\sigma'(n) = \sigma(k)$.

If $\sigma(k)$ is a linear variable then $T_{\sigma'} \leqslant T_\sigma$. In particular, if σ leads to an optimal solver then σ' too.

This means that there is an optimal solver σ such that for all $k \in \{1, \ldots, n\}$ either $\sigma(k) \in \textsc{Lin}(\{\sigma(k+1), \ldots, \sigma(n)\})$ or $\textsc{Lin}(\{\sigma(k+1), \ldots, \sigma(n)\}) = \{\sigma(k+1), \ldots, \sigma(n)\}$. In particular, we know that for any permutation of $\textsc{Lin}(\varnothing)$ there is an optimal solver ending by it and so only the subsystem $\mathbb{E}(\mathbf{X} - \textsc{Lin}(\varnothing))$ has to be studied. Thus, without loss of generality, we can assume that $\textsc{Lin}(\varnothing) = \varnothing$. Actually, it allows us to answer a non-trivial question. Indeed, our solvers alternate between guessing and deducing variables but one could ask why restrict ourselves to variables and not to linear combinations of them. Allowing a solver to guess a particular linear combination l can be done by adding to the system a new variable x and the equation $x - l = 0$. But then this new variable occurs linearly in the system of equations and thanks to the property we are sure there is an optimal solver which does not require to guess it.

As previously, Property 9 comes from a more general theorem about the replacement of a solver by an other and is stated as follows:

Theorem 2. *Let $\mathbb{E}$ be a system of AES-like equations in variables $\mathbf{X}$ and $\mathbf{Y}, \mathbf{Z}$ be two subsets of $\mathbf{X}$ such that $\mathbf{Y} \subseteq \mathbf{Z}$. Let $\sigma_\mathbf{Y}$, $\sigma_\mathbf{Z}$ and $\sigma_{\mathbf{X}-\mathbf{Y}}$ be permutations of $\mathbf{Y}$, $\mathbf{Z}$ and $\mathbf{X} - \mathbf{Y}$ respectively. Let $\sigma_{\mathbf{X}-\mathbf{Z}}$ be the permutation of $\mathbf{X} - \mathbf{Z}$ conserving the order induced by $\sigma_{\mathbf{X}-\mathbf{Y}}$. If for any subset $\mathbf{V}$ such that $\mathbf{Y} \subseteq \mathbf{V} \subseteq \mathbf{Z}$ then $|Sol(\mathbb{E}(\mathbf{V}))| \geqslant |Sol(\mathbb{E}(\mathbf{Z}))|$ and if $T^*_{\sigma_\mathbf{Z}} \leqslant T^*_{\sigma_\mathbf{Y}}$, then*

$$T^*_{[\sigma_{\mathbf{X}-\mathbf{Z}} \parallel \sigma_\mathbf{Z}]} \leqslant T^*_{[\sigma_{\mathbf{X}-\mathbf{Y}} \parallel \sigma_\mathbf{Y}]}.$$

In particular, if there is an optimal solver going through $\mathbf{Y}$ *then there is one going through* $\mathbf{Z}$.

All in all, as for PROPAGATE we define $\mathcal{L}^\star$ as the least fixed point of LIN, leading to Algorithm 4 which is very close to the previous one. The only difference concerns the detection of $\mathbb{E}(\mathbf{X}-\mathbf{Z})$ as an overdetermined subsystem failing the heuristic assumption on the number of solutions. In that case, and unlike to Algorithm 2, we do not have a solver for this subsystem and the complexity of the ending solver has been underestimated. Thus we first try to find a solver fast enough for the subsystem. If there is no such solver then we can continue the search. But if such a solver is found then the existence of solver for $\mathbb{E}$ fast enough only depends on the existence of such one for $\mathbb{E}_{\mathbf{X}-\mathbf{Z}}$.

As previously the complexity of this algorithm is hard to evaluate. However if the case $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}-\mathbf{Z}))\,| \leqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}-\mathbf{Y}))\,|$ never occurs then the complexity is bounded by $t_{up}\binom{|\mathbf{X}|}{t_{up}-|X|+\dim\mathbb{E}}$ applications of $\mathcal{L}^\star$. As a consequence this top-down approach seems faster than the bottom-up one as long as the system is underdefined. In practice we noticed that to be true even for systems with few more equations than variables. Our feeling is that once we know enough variables the other ones can be deduced in almost any order.

Bounding Complexity of Optimal Solvers

It is always interesting to bound the complexity of an optimal solver before looking for it. First, the solvers we consider enumerate all the solutions of the system of equations and thus the time complexity of a solver is greater than the number of solutions enumerated.

Property 10. *Let* $\mathbb{E}$ *be a system of AES-like equations and* σ *be (a permutation corresponding to) a solver for* $\mathbb{E}$. *Then* $T_\sigma \geqslant |\mathcal{S}ol\,(\mathbb{E})\,|$. *In particular, if* $T_\sigma = |\mathcal{S}ol\,(\mathbb{E})\,|$ *then* σ *is optimal.*

This gives us a lower bound on the time complexity of an optimal solver. Thanks to Property 1 and under the heuristic assumption, we can obtain an upper bound by noticing that for any permutation σ and any $k \in \{1,\ldots,n-1\}$, we have

$$\log_q\left(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1),\ldots,\sigma(k+1)\}))\,|\right) - \log_q\left(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1),\ldots,\sigma(k)\}))\,|\right) \in \{-1,0,1\}.$$

As a consequence, and as $\log_q\left(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1)\}))\,|\right) = 1$ and $\log_q\left(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1),\ldots,\sigma(n)\}))\,|\right) = n-\dim\mathbb{E}$, we obtain

$$\log_q\left(|\mathcal{S}ol\,(\mathbb{E}(\{\sigma(1),\ldots,\sigma(k)\}))\,|\right) \leqslant \min(k, n-\dim\mathbb{E}+n-k) \leqslant n-\dim\mathbb{E}/2.$$

Thus the time complexity of an optimal solver is at most $q^{n-\dim\mathbb{E}/2}$.

Algorithm 3: Exhaustive search of an optimal solver (from the top)

Function `TopSearch` ($\mathbb{E}$,t_{up})

  Data: A system of equations $\mathbb{E}$ in variables $\mathbf{X}$ and an upper bound t_{up}

  Result: A permutation of $\mathbf{X}$ leading to a solver with time complexity equals to at most $q^{t_{up}}$ if any and $\varnothing$ otherwise.

  $\mathbb{E} \longleftarrow \mathbb{E}$ without equations between the parameters;
  if $\textsc{Lin}(\varnothing) = \varnothing$ **then**
    if $\textsc{Propagate}(\varnothing) = \varnothing$ **then**
      return `tmp-Top` ($\mathbb{E}, \varnothing, \varnothing, \varnothing, t_{up}$)
    else
      $\sigma \longleftarrow$ a permutation of $\textsc{Propagate}(\varnothing)$;
      return $\sigma \parallel$ `TopSearch` ($\mathbb{E}_{\textsc{Propagate}(\varnothing)}, t_{up}$)
    end
  else
    $\sigma \longleftarrow$ a permutation of $\textsc{Lin}(\varnothing)$;
    return `TopSearch` ($\mathbb{E}(\mathbf{X} - \textsc{Lin}(\varnothing)), t_{up}$) $\parallel \sigma$
  end
end

Function `tmp-Top` ($\mathbb{E}$,$\mathbf{Y}$,G_{not},σ,t_{up})

  Data:
  - a system of equations $\mathbb{E}$ in variables $\mathbf{X}$,
  - a subset $\mathbf{Y}$ of $\mathbf{X}$ stable by $\mathcal{L}$,
  - a subset G_{not} containing $\mathbf{Y}$ and variables not allowed to be guessed,
  - a permutation σ of $\mathbf{Y}$,
  - a bound $t_{up} \geqslant 1$.

  Result: A permutation of $\mathbf{X}$ leading to a solver with time complexity equals to at most $q^{t_{up}}$ and which ends by solving $\mathbb{E}(\mathbf{Y})$ according to σ if any and $\varnothing$ otherwise.

  if $\mathbf{X} - G_{not} = \varnothing$ **then return** $\varnothing$;
  $x \longleftarrow$ Pick one variable from $\mathbf{X} - G_{not}$;
  $\mathbf{Z} \longleftarrow \textsc{Lin}^{\star}(\mathbf{Y} \cup \{x\})$; $\sigma' \longleftarrow \textsc{Lin}^{\star}([x] \parallel \sigma)$;
  if $\mathbf{Z} = \mathbf{X}$ **then return** σ';
  if $|\mathcal{S}ol(\mathbb{E}(\mathbf{X} - \mathbf{Z}))| \leqslant |\mathcal{S}ol(\mathbb{E}(\mathbf{X} - \mathbf{Y}))|$ **then**
    if $|\mathcal{S}ol(\mathbb{E}(\mathbf{X} - \mathbf{Z}))| < 1$ **then**
      $\sigma_1 \longleftarrow$ `TopSearch` ($\mathbb{E}(\mathbf{X} - \mathbf{Z})$,$t_{up}$) ;
      if $\sigma_1 \neq \varnothing$ **then return** $\sigma_1 \parallel$ `TopSearch` ($\mathbb{E}_{(\mathbf{X}-\mathbf{Z})}$,$t_{up}$) ;
    else
      if $|\mathcal{S}ol(\mathbb{E}(\mathbf{X} - \mathbf{Y}))| = t_{up} - 1$ **then return** `tmp-Top` ($\mathbb{E}$,$\mathbf{Z}$,$\mathbf{Z}$,σ',t_{up}) ;
      if $(\mathbf{Z} - \mathbf{Y}) \cap G_{not} = \varnothing$ **then return** `tmp-Top` ($\mathbb{E}$,$\mathbf{Z}$,$G_{not} \cup \mathbf{Z}$,σ',t_{up}) ;
    end
  else
    if $|\mathcal{S}ol(\mathbb{E}(\mathbf{X} - \mathbf{Z}))| < t_{up}$ *and* $(\mathbf{Z} - \mathbf{Y}) \cap G_{not} = \varnothing$ **then**
      $\sigma'' \longleftarrow$ `tmp-Top` ($\mathbb{E}$,$\mathbf{Z}$,$G_{not} \cup \mathbf{Z}$,σ',t_{up}) ;
      if $\sigma'' \neq \varnothing$ **then return** σ'';
    end
  end
  return `tmp-Top` ($\mathbb{E}$,$\mathbf{Y}$,$G_{not} \cup \mathbf{Z}$,σ',t_{up})
end

2.2 Recursive Meet-in-the-Middle Solvers

Given a system of equations $\mathbb{E}$ in variables $\mathbf{X}$, we saw in the previous section how to build from a solver of a subsystem $\mathbb{E}(\mathbf{Y})$ a solver for $\mathbb{E}(\mathbf{Y} \cup \{x\})$, allowing us to build various guess-and-determine solvers for $\mathbb{E}$. But actually this is a particular case of a more general framework and in this section we will see how to extend the previous solvers to get faster ones by using a "divide-and-conquer" meet-in-the-middle approach.

2.2.1 Solving Subsystems Recursively

Given a partition $\mathbf{X} = \mathbf{X}_1 \cup \mathbf{X}_2$ and two *black-box solvers* $\mathcal{A}_1$ and $\mathcal{A}_2$ that find all the solutions of $\mathbb{E}(\mathbf{X}_1)$ and $\mathbb{E}(\mathbf{X}_2)$ respectively, we seek to use the two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ to find the solutions $\mathcal{S}ol\,(\mathbb{E})$ of the full problem. An obvious way would be to compute the solutions $\mathcal{S}_1$ of $\mathbb{E}(\mathbf{X}_1)$ and $\mathcal{S}_2$ of $\mathbb{E}(\mathbf{X}_2)$, and to test all the solutions in the Cartesian product $\mathcal{S}_1 \times \mathcal{S}_2$. This would require checking $|\mathcal{S}_1| \cdot |\mathcal{S}_2|$ candidates against the equations.

It is possible to do better though. Firstly, we observe that the vectors in $\mathcal{S}_1 \times \mathcal{S}_2$ automatically satisfy the equations in $\mathbb{E}(\mathbf{X}_1) + \mathbb{E}(\mathbf{X}_2)$. Therefore we first compute a supplementary of $\mathbb{E}(\mathbf{X}_1) + \mathbb{E}(\mathbf{X}_2)$ inside $\mathbb{E}$ (let us call it $\mathcal{M}$). The solutions of $\mathbb{E}$ are in fact the elements of $\mathcal{S}_1 \times \mathcal{S}_2$ satisfying the equations of $\mathcal{M}$. This already makes less constraints to check. Second, sieving the elements satisfying constraints from $\mathcal{M}$ can be done in roughly $|\mathcal{S}_1| + |\mathcal{S}_2|$ operations, using variable separation and a table. Let $(f_i)_{1 \leqslant i \leqslant m}$ be a basis of $\mathcal{M}$. Thanks to the structure of AES-like equations, each of those equations can be written $g_i(\mathbf{X}_1) = h_i(\mathbf{X}_2)$. If the values of all the variables in $\mathbf{X}_1$ (resp. $\mathbf{X}_2$) are available, then the g_i's (resp. h_i) may be evaluated. We denote by G (resp. H) the function that evaluates all the g_i (resp. h_i) on its input. If $\ell = |\mathbf{X}_1|$, then:

$$G : (x_1, \ldots, x_\ell) \mapsto \Big(g_1(x_1, \ldots, x_\ell), \ldots, g_m(x_1, \ldots, x_\ell)\Big)$$

We build two tables:

$$\begin{aligned} L_1 &\longleftarrow \{(G(x_1), x_1) \mid x_1 \text{ solution of } \mathbb{E}(\mathbf{X}_1)\} \\ L_2 &\longleftarrow \{(H(x_2), x_2) \mid x_2 \text{ solution of } \mathbb{E}(\mathbf{X}_2)\} \end{aligned}$$

Then, the solutions of $\mathbb{E}$ are the pairs (x, y) for which there exist a z such that $(z, x) \in L_1$ and $(z, y) \in L_2$. They can be identified efficiently by various methods (sorting the tables, using a hash index, etc.). We have just combined $\mathcal{A}_1$ and $\mathcal{A}_2$ to form a new solver, $\mathcal{A} = \mathcal{A}_1 \bowtie \mathcal{A}_2$, that enumerates the solutions of $\mathbb{E}$. Note that to extend this work at a cover of $\mathbf{X}$ we just have to perform the match also on variables common to $\mathbf{X}_1$ and $\mathbf{X}_2$.

Complexity of the Combination.

Given a cover $\mathbf{X} = \mathbf{X}_1 \cup \mathbf{X}_2$, and two sub-solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively computing $\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1))$ and $\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2))$, the complexity and the properties of $\mathcal{A} = \mathcal{A}_1 \bowtie \mathcal{A}_2$ are easy to determine. Let us denote by $T(\mathcal{A})$ the running time of $\mathcal{A}$, by $M(\mathcal{A})$ its memory consumption, and by $V(\mathcal{A})$ the set of variables occurring in the corresponding equations. The set of solutions returned by a solver $\mathcal{A}$ only depends on $V(\mathcal{A})$, as it is the set of

solutions of $\mathbb{E}(V(\mathcal{A}))$. For the sake of simplicity, we denote it by $\mathcal{S}ol(\mathcal{A})$. Note that the number of solutions found by a solver cannot be greater than its running time, so that $|\mathcal{S}ol(\mathcal{A})| \leqslant T(\mathcal{A})$.

The number of operations performed by the combination is the sum of the number of operations produced by the sub-solvers plus the number of solutions (the time required to scan the tables, namely $|\mathcal{S}_1| + |\mathcal{S}_2|$, is in the worst case of the same order as the running time of the two sub-solvers), so that

$$T(\mathcal{A}_1 \bowtie \mathcal{A}_2) = T(\mathcal{A}_1) + T(\mathcal{A}_2) + |\mathcal{S}ol(\mathcal{A}_1 \bowtie \mathcal{A}_2)|.$$

However, we use the following approximation

$$T(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \max\Big\{T(\mathcal{A}_1), T(\mathcal{A}_2), |\mathcal{S}ol(\mathcal{A}_1 \bowtie \mathcal{A}_2)|\Big\}.$$

It is possible to store only the smallest table, and to enumerate the content of the other "on the fly", while looking for a collision. This reduces the memory complexity to the maximum of the memory complexity of the sub-solvers, and the size of the smaller table. This yields:

$$M(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \max\Big\{M(\mathcal{A}_1), M(\mathcal{A}_2), \min\Big(|\mathcal{S}ol(\mathcal{A}_1)|, |\mathcal{S}ol(\mathcal{A}_2)|\Big)\Big\}.$$

2.2.2 Recursive Combinations of Solvers

Given a system of equations $\mathbb{E}$, we would like to build an efficient solver by breaking the problem down to smaller and smaller subsystems, recursively generating efficient sub-solver for the sub-problems and combining them back.

Recursively combining solvers yields *solving trees* of various shapes. In such a tree, all the nodes are labelled by a set of variables: the leaves are labelled by single variables and each node is labelled by the union of the labels of its children. Each node is in fact a *solver* that solves the sub-system $\mathbb{E}(\mathbf{Y})$, where $\mathbf{Y}$ is the label of the node. For obvious reasons, we enforce that the label of each node is strictly larger than the labels of its children.

The leaves of a solving tree are the "base solvers" associated to variables of $\mathbf{X}$. Note that for any variable x the subsystem $\mathbb{E}(\{x\})$ cannot be further broken down because obviously $\{x\}$ cannot be partitioned anymore. It is a "base case" of the decomposition. As previously, to be consistent with the heuristic assumption on the number of solutions if $\mathbb{E}(\{x\}) \neq \{0\}$ then it is assumed that x takes a single value, and then x can be seen as a parameter. So without a loss of generality we only consider the case $\mathbb{E}(\{x\}) = \{0\}$. As a consequence, a base solver essentially guesses a variable and its complexity is:

- $T(\mathsf{BaseSolver}(x)) = q$.
- $M(\mathsf{BaseSolver}(x)) = 1$.
- $|\mathcal{S}ol(\mathsf{BaseSolver}(x))| = q$.

This implies that, for considered solvers (*i.e.*, those generated by base solvers), time, memory and number of solutions are powers of q. In the sequel, unless otherwise stated explicitly, a "solver" always designates the recursive combinations with base solvers at the end.

Note that the guess-and-determine solvers discussed in the previous section can be described by a recursive combination where, at each step of the decomposition, one of the two solvers is a base solver. However, it turns out that allowing more general tree shapes may results in better solvers.

Comparing Solvers.

It is always possible to construct several solving trees for the same problem in different ways, and sometimes more or less efficiently. Indeed, a quick calculation, with $|\mathbf{X}| = n$, gives the number of distinct covers of X:

$$|\{\{X_1, X_2\} \,|X_1 \cup X_2 = X, X_1 \neq X, X_2 \neq X\}| = \frac{3^n + 1}{2} - 2^n.$$

The actual number of different solvers is then necessarily even larger. In addition, because our solvers are at least as fast as exhaustive search, we observe that our approximation of the time complexity of a solver for $\mathbb{E}\,(X)$ can take only n different values. So we deduce that there are many solvers with the same approximate complexity solving the same system. We will therefore introduce a (quasi-)order relation over solvers. A natural candidate is:

$$\mathcal{A}_1 \succeq_1 \mathcal{A}_2 \Longleftrightarrow \begin{cases} V(\mathcal{A}_1) & = & V(\mathcal{A}_2) \\ T(\mathcal{A}_1) & \leqslant & T(\mathcal{A}_2) \end{cases}.$$

In other words, a solver is better than an other if it solves the same system in less time. Just like any other partial quasi-order, it induces an equivalence relation:

$$\mathcal{A}_1 \equiv \mathcal{A}_2 \text{ if and only if } \mathcal{A}_1 \succeq_1 \mathcal{A}_2 \text{ and } \mathcal{A}_2 \succeq_1 \mathcal{A}_1.$$

This quasi-order has the advantage of being compatible with the combination operation (*i.e.*, $\mathcal{A}_1 \succeq_1 \mathcal{A}_2$ implies $\mathcal{A}_1 \bowtie \mathcal{A}_3 \succeq_1 \mathcal{A}_2 \bowtie \mathcal{A}_3$), and it is therefore also the case of the equivalence relation. We observe that given a set of variables $\mathbf{X}_1$, there can be only one maximal solver (up to equivalence) for $\mathbb{E}(\mathbf{X}_1)$. Thus, our objective is now clearly identified: find a maximal (*i.e.*, the best) solver for $\mathbb{E}$ (up to equivalence).

2.2.3 Finding the best solver

In this part, we present two different approaches to find an optimal solver, and theorems of reducing the search space.

From The Top.

We begin by presenting an algorithm that find a solver with a time T_{up} for a system of equations $\mathbb{E}$ in the variables $\mathbf{X}$. First, we check that the number of solutions of the system is not higher than the time T_{up}, otherwise we could not list them all in less than T_{up}. Then, we just try all possible decompositions of $\mathbf{X}$.

Algorithm 4: Exhaustive search of an optimal solver (from the top)

Function `ExhaustiveSearchTop(`$\mathbb{E}, T_{up}$`)`
  Data: A system of equations $\mathbb{E}$ in variables $\mathbf{X}$ and a bound T_{up}.
  Result: A solver $\mathcal{A}$ for $\mathbb{E}$ such that $T(\mathcal{A}) \leqslant T_{up}$ (if any).
  if $|Sol\,(\mathbf{X})\,| > T_{up}$ **then return** $NULL$;
  if $|\mathbf{X}| = 1$ **then return** BaseSolver($\mathbf{X}$);
  foreach *cover* $\mathbf{X}_1 \cup \mathbf{X}_2 = \mathbf{X}$ **do**
    $\mathcal{A}_1 \leftarrow$`ExhaustiveSearchTop(`$\mathbb{E}(\mathbf{X}_1), T_{up}$`);`
    $\mathcal{A}_2 \leftarrow$`ExhaustiveSearchTop(`$\mathbb{E}(\mathbf{X}_2), T_{up}$`);`
    if $\mathcal{A}_1 \neq NULL$ *and* $\mathcal{A}_2 \neq NULL$ **then return** $\mathcal{A}_1 \bowtie \mathcal{A}_2$;
  end
  return $NULL$
end

The complexity of the procedure ExhaustiveSearchTop is hard to evaluate but can be bounded below by the number of covers of $\mathbf{X}$, which is $\frac{3^n+1}{2} - 2^n$, where $n = |\mathbf{X}|$. It can still be improved by using dynamic programming techniques to avoid searching several times a solver for the same system but this requires storing at most 2^n solvers. Unfortunately, this algorithm can not even be applied to the system of equations from one AES round (the number of variables and equations being 64), and need to be improved. Hopefully, Property 9 about linear variables can be transposed to this new set of solvers:

Theorem 3. *Let $\mathbb{E}$ be a system of equations in variables $\mathbf{X}$. For any $x \in$ Lin $(\mathbb{E})$ (i.e., x linearly occurs in the system $\mathbb{E}$), if $\mathcal{A}$ is an optimal solver for $\mathbb{E}(\mathbf{X} - \{x\})$ then $\mathcal{A} \bowtie$ BaseSolver(x) is an optimal solver for $\mathbb{E}$.*

The proof is easy and comes from the fact that given a solver $\mathcal{A}$ for $\mathbb{E}(\mathbf{X})$ and $x \in$ Lin $(\mathbb{E})$ then the solver $\mathcal{A}'$ for $\mathbb{E}(\mathbf{X} - \{x\})$ built following the same decomposition than $\mathcal{A}$ but without the variable x verifies $T(\mathcal{A}') \leqslant T(\mathcal{A})$. As previously, this theorem allows us to focus on a smaller system of equations in our research of an optimal solver. On systems of equations from the AES, this improvement significantly reduces the search space but it is still infeasible for more than one round of AES.

From The Bottom.

After the failure of the previous approach, it is natural to try to start from the base solvers and test different combinations until obtaining an optimal solver.

The procedure ExhaustiveSearch in Algorithm 5 computes the set of all maximal solvers for all sub-systems of a given system of equations $\mathbb{E}$ (up to equivalence). In particular, it will construct a maximal solver for $\mathbb{E}$ itself. The algorithm is reminiscent of (and inspired by) the Buchberger algorithm for Gröbner bases [Buc65]. More generally Algorithm 5 is a saturation procedure, and this also makes it similar to many automated deduction procedures (such a Resolution-based theorem provers or the Knuth-Bendix completion algorithm). At each step, the algorithm maintains a list G of solvers for

subsystems of the original system $\mathbb{E}$. It also maintains a list $\mathcal{P}$ of pairs of solver that remain to be processed. When a new solver is found, all the solvers that are worse (according to $\succeq_1$) are removed from G (and all pairs containing it are removed as well). Then, new pairs containing the new solver are scheduled for processing.

Algorithm 5: Exhaustive Search from the top for an optimal solver (from the bottom)

Function `Update-Queue` $(G, \mathcal{P}, \mathcal{A})$
  if $\mathcal{A}' \nsucceq_1 \mathcal{A}$ *for all* $\mathcal{A}' \in G$ **then**
    $G' \leftarrow \{\mathcal{A}\} \cup G - \{\mathcal{A}' \in G \ : \ \mathcal{A} \succeq_1 \mathcal{A}'\}$;
    $\mathcal{P}' \leftarrow \mathcal{P} - \big\{(\mathcal{A}_1, \mathcal{A}_2) \in \mathcal{P} \ : \ \mathcal{A} \succeq_1 \mathcal{A}_1 \text{ or } \mathcal{A} \succeq_1 \mathcal{A}_2\big\}$;
    $\mathcal{P}' \leftarrow \mathcal{P}' \cup \big\{(\mathcal{A}, \mathcal{A}') \ : \ \mathcal{A}' \in G',\ V(\mathcal{A}) \not\subseteq V(\mathcal{A}'),\ V(\mathcal{A}') \not\subseteq V(\mathcal{A})\big\}$;
    return $(G', \mathcal{P}')$
  end
  return $(G, \mathcal{P})$
end

Function `ExhaustiveSearch` $(\mathbb{E}, T_{up})$
  Data: A system of equations $\mathbb{E}$ in variables $\mathbf{X}$ and a bound T_{up}.
  Result: The set of all maximal solvers for all subsystems of $\mathbb{E}$ with time complexity smaller or equal to T_{up}.
  $G \leftarrow \big\{\mathsf{BaseSolver}(x) : x \in \mathbf{X}\big\}$;
  $\mathcal{P} \leftarrow \big\{(G_i, G_j) : 1 \leqslant i < j \leqslant |G|\big\}$;
  while $\mathcal{P} \neq \varnothing$ **do**
    Pick $(\mathcal{A}_1, \mathcal{A}_2) \in \mathcal{P}$ and remove it from $\mathcal{P}$;
    $\mathcal{C} \leftarrow \mathcal{A}_1 \bowtie \mathcal{A}_2$;
    if $T(\mathcal{C}) \leqslant T_{up}$ **then** $(G, \mathcal{P}) \leftarrow$ `Update-Queue` $(G, \mathcal{P}, \mathcal{C})$;
  end
  return G
end

Termination. This search procedure only uses the compatibility of $\succeq_1$ with the combination operation $\bowtie$. First, we notice that, at each step of the algorithm, G can contain at most one solver by subset of $\mathbf{X}$ (the best found so far). It follows that $|G| \leqslant 2^{|\mathbf{X}|}$. Next, for a subset $\mathbf{Y}$ of $\mathbf{X}$, there exist at most $|\mathbf{Y}|$ distinct solvers (up to equivalence). It follows that the number time G will be modified by UpdateQueue is upper bounded by $|\mathbf{X}| \cdot 2^{|\mathbf{X}|}$. Next, there can be only a finite number of steps between two updates of G, because each iteration of the loop consumes an element of $\mathcal{P}$, and only an actual modification of G can increase $\mathcal{P}$. As a result, the ExhaustiveSearch procedure terminates in finite time.

Correction. One of the invariants of this algorithm comes from the compatibility of $\succeq_1$ with the combination operation $\bowtie$ and is the property: "if $\mathcal{A}_1$, $\mathcal{A}_2 \in G$ and $T(\mathcal{A}_1 \bowtie \mathcal{A}_2) \leqslant T_{up}$ then either there is $(\mathcal{A}_3, \mathcal{A}_4) \in \mathcal{P}$ such that $\mathcal{A}_3 \bowtie \mathcal{A}_4 \succeq_1 \mathcal{A}_1 \bowtie \mathcal{A}_2$ or there is $\mathcal{A}_3 \in G$ such that $\mathcal{A}_3 \succeq_1 \mathcal{A}_1 \bowtie \mathcal{A}_2$". But, when the algorithm terminates, $\mathcal{P}$ is empty and so we always are in the second case of the previous property. This means that for each solver with an approximate time complexity smaller than T_{up} and generated from solvers of G, there is a solver in G solving the same system with at least the same approximate time complexity. But the base solvers allow to generate all solvers and G contains them, so G also allows it. In particular G allows to generate the best solver for $\mathbb{E}(\mathbf{X})$ and, as a consequence, if T_{up} is high enough then G contains it.

Complexity. The complexity of this algorithm seems difficult to evaluate. It depends on the equations, and on the order in which the combinations are performed. The parameter T_{up} allows the user to enforce an upper-bound on the time complexity of the generated solvers (by discarding the ones that are too slow). For small values of T_{up}, this may for instance allow to prove the non-existence of recursive solvers with complexity lower than a threshold. The running time of the exhaustive search also gets smaller with lower values of T_{up}.

In practice, what dominates the execution of this algorithm is the computation of the dimension of the combination $\mathcal{C}$, and the bookkeeping required to update G ($\mathcal{P}$ can be handled implicitly).

Comparing Solver II.

The complexity of this algorithm depends directly on the quasi-order relation: more the solvers are comparable between them, more G (and by the way $\mathcal{P}$) will be small. But we note that many solvers are not comparable with this quasi-order. In particular, two solvers cannot be compared if they do not enumerate the exact same set of variables. It would seem natural that if a solver is faster and enumerates more variables, then it should be better. This prompts for the relaxation of the $V(\mathcal{A}_1) = V(\mathcal{A}_2)$ condition into $V(\mathcal{A}_1) \supseteq V(\mathcal{A}_2)$ in the definition of $\succeq_1$. However, a problem is that this relaxed quasi-order relation is incompatible with the $\bowtie$ operation (explicit counter-examples exist). The problem is that a faster solver that enumerates more variables may generate more solutions, and this can slow down the subsequent combination operations. Trying to fix the problem leads to the definition of:

$$\mathcal{A}_1 \succeq_2 \mathcal{A}_2 \Longleftrightarrow \begin{cases} T(\mathcal{A}_1) & \leqslant & T(\mathcal{A}_2) \\ V(\mathcal{A}_1) & \supseteq & V(\mathcal{A}_2) \\ |\mathcal{S}ol(\mathcal{A}_1)| & \leqslant & |\mathcal{S}ol(\mathcal{A}_2)| \end{cases} .$$

Unfortunately, and as for guess-and-determine solvers (see Theorem 1), this new condition is not enough to ensure compatibility with the $\bowtie$ operation (explicit yet subtler examples exist). To solve this problem once and for all, we impose the compatibility in the definition.

Definition 7. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two solvers. We denote by $\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2$ the following property:*

For any sequence of solvers $(\mathcal{B}_n)_n$, the two sequences defined by:

$$\left\{ \begin{array}{rcl} \mathcal{A}_i^0 & = & \mathcal{A}_i \\ \mathcal{A}_i^{n+1} & = & \mathcal{A}_i^n \bowtie \mathcal{B}_n \end{array} \right. \qquad \text{for } i = 1, 2.$$

verify $\mathcal{A}_1^n \succeq_2 \mathcal{A}_2^n$ for all $n \geqslant 0$.

In other words, $\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2$ if for any solver built from $\mathcal{A}_2$ we can construct a better solver (according to $\succeq_2$) from $\mathcal{A}_1$. We easily verify that $\succeq_2^\star$ is a quasi-order compatible with the combination operation. But now the problem is that there is no obvious algorithm to compare solvers according to this relation. The next property shows a necessary and sufficient condition on $\mathcal{A}_1$ and $\mathcal{A}_2$ to imply $\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2$.

Property 11. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two solvers. Then:*

$$\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2 \Longleftrightarrow \left\{ \begin{array}{l} \mathcal{A}_1 \succeq_2 \mathcal{A}_2 \\ V(\mathcal{A}_2) \subseteq \mathbf{Y} \subseteq V(\mathcal{A}_1) \Longrightarrow |\mathcal{S}ol\,(\mathbb{E}(\mathbf{Y}))\,| \geqslant \mathcal{S}ol\,(\mathcal{A}_1)\,| \end{array} \right.$$

Proof. ($\Rightarrow$) First, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two solvers such that $\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2$. Then, by definition we have $\mathcal{A}_1 \succeq_2 \mathcal{A}_2$. Next, let $\mathbf{Y}$ be subset of $V(\mathcal{A}_1)$ such that $V(\mathcal{A}_2) \subseteq \mathbf{Y}$. Let $\mathcal{B}$ be a solver such that $V(\mathcal{B}) = \mathbf{Y}$. Then, by hypothesis, we have $\mathcal{A}_1 \bowtie \mathcal{B} \succeq_2 \mathcal{A}_2 \bowtie \mathcal{B}$. In particular $|\mathcal{S}ol\,(\mathcal{A}_1 \bowtie \mathcal{B})\,| \leqslant |\mathcal{S}ol\,(\mathcal{A}_2 \bowtie \mathcal{B})\,|$. But $V(\mathcal{A}_1 \bowtie \mathcal{B}) = V(\mathcal{A}_1)$ and $V(\mathcal{A}_2 \bowtie \mathcal{B}) = \mathbf{Y}$. In consequence, $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{Y}))\,| \geqslant |\mathcal{S}ol\,(\mathcal{A}_1)\,|$.

($\Leftarrow$) Now, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two solvers such that $\mathcal{A}_1 \succeq_2 \mathcal{A}_2$. We assume that for any set $\mathbf{Y}$ such that $V(\mathcal{A}_2) \subseteq \mathbf{Y} \subseteq V(\mathcal{A}_1)$ we have $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{Y}))\,| \geqslant |\mathcal{S}ol\,(\mathcal{A}_1)\,|$. We will show that $\mathcal{A}_1 \succeq_2^\star \mathcal{A}_2$. Let $(\mathcal{B}_n)_n$ be a sequence of solvers. We show the result by induction on n.

- By hypothesis, we have $\mathcal{A}_1 \succeq_2 \mathcal{A}_2$, so the theorem is true when $n = 0$.
- Next, we assume that $\mathcal{A}_1^n \succeq_2 \mathcal{A}_2^n$ and we will prove that this implies $\mathcal{A}_1^{n+1} \succeq_2 \mathcal{A}_2^{n+1}$. First, we notice that by hypothesis we know that $V(\mathcal{A}_1^n) \supseteq V(\mathcal{A}_2^n)$, and this easily implies $V(\mathcal{A}_1^{n+1}) \supseteq V(\mathcal{A}_2^{n+1})$. Next, because $T\,(\mathcal{A}_1^n) \leqslant T\,(\mathcal{A}_2^n)$ and

$$T\,(\mathcal{A}_i^n \bowtie \mathcal{B}_n) = \max\left(T\,(\mathcal{A}_i^n)\,, T\,(\mathcal{B}_n)\,, |\mathcal{S}ol\,(\mathcal{A}_i^n \bowtie \mathcal{B}_n)\,|\right),$$

  it follows that we just have to prove that $|\mathcal{S}ol\,(\mathcal{A}_1^n \bowtie \mathcal{B}_n)\,| \leqslant |\mathcal{S}ol\,(\mathcal{A}_2^n \bowtie \mathcal{B}_n)\,|$ to establish the result.
  Let us define $\mathbf{Y} = \bigcup_{i=0}^n V(\mathcal{B}_i)$. We use the lemma 1 with $\mathbf{X}_1 = V(\mathcal{A}_1)$, $\mathbf{X}_2 = V(\mathcal{A}_2) \cup (V(\mathcal{A}_1) \cap \mathbf{Y})$ and $\mathbf{X}_3 = \mathbf{Y}$. Indeed, we easily verify that $\mathbf{X}_2 \subseteq \mathbf{X}_1$ and $\mathbf{X}_1 \cap \mathbf{X}_3 = \mathbf{X}_2 \cap \mathbf{X}_3$. Then, as $V(\mathcal{A}_2) \subseteq \mathbf{X}_2 \subseteq V(\mathcal{A}_1)$, by hypothesis we know that $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2))\,| \geqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1))\,|$. So the conditions to apply the lemma are satisfied, and we deduce that $|\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_2 \cup \mathbf{X}_3))\,| \geqslant |\mathcal{S}ol\,(\mathbb{E}(\mathbf{X}_1 \cup \mathbf{X}_3))\,|$. But $\mathbf{X}_i \cup \mathbf{X}_3 = V(\mathcal{A}_i^{n+1})$, so we obtain the expected result.

□

As said in the previous section, it is difficult in practice to check whether the assumptions of this property are satisfied. We therefore use the relation $\succeq_2^\star$ rarely, preferring to use $\succeq_1$ or $\succeq_2$ which are easier to compute.

Finally, to end this part, we note that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are two solvers then:

$$\mathcal{A}_1 \succeq_1 \mathcal{A}_2 \Longrightarrow \mathcal{A}_1 \succeq_2^\star \mathcal{A}_2 \Longrightarrow \mathcal{A}_1 \succeq_2 \mathcal{A}_2.$$

If, in addition, they satisfy $V(\mathcal{A}_1) = V(\mathcal{A}_2)$ then:

$$\mathcal{A}_1 \succeq_1 \mathcal{A}_2 \Longleftrightarrow \mathcal{A}_1 \succeq_2^\star \mathcal{A}_2 \Longleftrightarrow \mathcal{A}_1 \succeq_2 \mathcal{A}_2.$$

In particular, an optimal solver for $\mathbb{E}(X)$ is a maximal solver for each one of the quasi-orders. In addition, equivalence relations induced by all introduced quasi-orders are identical.

Search Space Reduction.

Even if the above results can be seen as a failure, we have two theorems that bypass partially this compatibility problem.

The first theorem is used to restrict the shape of solvers to consider and, in practice, greatly reduces the complexity of the search procedure.

Theorem 4. *Let $\mathcal{A}$ be a solver with running time T and let $s \geqslant T$, s and T being power of q. Then there exists a solver solving the same system with a time complexity of at most s and verifying the following property: at each step of its decomposition, one of the solvers is a base solver or the two solvers that are combined together both give s solutions.*

Proof. Let $(\mathcal{A}_1, \mathcal{A}_2)$ be a pair of solver combined together in the construction of $\mathcal{A}$. As $T(\mathcal{A}) = T$, necessary both of them have a running time of at most T. Let assume that the property is true for solvers enumerating less variables than $\mathcal{A}$, so in particular for $\mathcal{A}_1$ and $\mathcal{A}_2$. If $\mathcal{A}_1$ gives less than s solutions then we combine it with base solvers of $\mathbb{E}(V(\mathcal{A}_2))$ until either the resulting solver $\mathcal{A}'_1$ gives exactly (according to the heuristic assumption) s solutions or solves $\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2))$. Similarly we construct $\mathcal{A}'_2$ from $\mathcal{A}_2$ and then we can replace $\mathcal{A}_1 \bowtie \mathcal{A}_2$ in $\mathcal{A}$ by $\mathcal{A}'_1$ if $V(\mathcal{A}'_1) = V(\mathcal{A}_1) \cup V(\mathcal{A}_2)$, or by $\mathcal{A}'_2$ if $V(\mathcal{A}'_2) = V(\mathcal{A}_1) \cup V(\mathcal{A}_2)$ or by $\mathcal{A}'_1 \bowtie \mathcal{A}'_2$. □

This theorem shows the importance to combine solver with base solvers and, by the way, the importance of guess-and-determine solvers. In particular, a solver with time and memory equal to T can be generated by guess-and-determine solvers with time T. We can therefore reuse the ideas of the previous sections and in particular the operator Propagate.

Theorem 5. *Let $\mathcal{A}$ be a solver. There exists another solver $\mathcal{B}$ such that:*

- $V(\mathcal{B}) = \text{Propagate}^\star(V(\mathcal{A}))$,
- $T(\mathcal{B}) = T(\mathcal{A})$,
- $\mathcal{B} \succeq_2^\star \mathcal{A}$.

Such a solver is unique up to equivalence, so we denote it by PROPAGATE$^\star(\mathcal{A})$ and its construction is straightforward. As PROPAGATE$^\star(\mathcal{A}) \succeq_2^\star \mathcal{A}$, it will always be more interesting to process solvers through PROPAGATE*.

These two theorems allow to improve the EXHAUSTIVESEARCH procedure by using the function IMPROVED-UPDATE-QUEUE instead of UPDATE-QUEUE to reduce the number of tested pairs. The fact that we can replace $\mathcal{A}$ by PROPAGATE$^\star(\mathcal{A})$ comes from Theorem 5. The others changes come from Theorem 4. Then, we can use the Theorem 3 to find an optimal solver for a subsystem of $\mathbb{E}(\mathbf{X})$ before extending it to an optimal solver for the entire system. Finally, as a last improvement, we perform a first pass using $\succeq_2$ instead of $\succeq_1$.

Algorithm 6: Improved UPDATE-QUEUE procedure.

```
Function Improved-Update-Queue (G, P, A)
    A ← PROPAGATE*(A);
    if there is A' ∈ G such that A' ⪰₁ A then return (G, P);
    G' ← {A} ∪ G − {A' ∈ G : A ⪰₁ A'};
    P' ← P − {(A₁, A₂) ∈ P : A ⪰₁ A₁ or A ⪰₁ A₂};
    P' ← P' ∪ {(A, BASESOLVER(x)) : x ∈ X − V(A)};
    if T(A) = |Sol(A)| then
        P' ← P' ∪ {(A, A') : A' ∈ G', V(A) ⊄ V(A'), V(A') ⊄ V(A), T(A') =
        |Sol(A')| = T(A)};
    end
    return (G', P')
end
```

In practice, when two guess-and-determine solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $T(\mathcal{A}_1) = |\mathcal{S}ol\,(\mathcal{A}_1)| = T(\mathcal{A}_2) = |\mathcal{S}ol\,(\mathcal{A}_2)| \geqslant |\mathcal{S}ol\,(\mathcal{A}_1 \bowtie \mathcal{A}_2)|$ are found, we are able, in most cases, to build very quickly an optimal solver from them.

2.3 Conclusion

Algorithms presented in this chapter have been developed and implemented in C. The running time is dominated by the computation of the time-complexity of a combination of solvers, which involves computing the dimension of a vector-space intersection. Various tricks can also be used to speed this operation up (using a sparse representation, precomputing partially echelonized forms, not computing an intersection but a sum, etc.

The complexity of the exhaustive search is inherently exponential, and exploring the whole space might not be feasible. In that case, a non-exhaustive randomized search might find good results, without offering the guarantee that they are the best possible. There are many possible ways to perform a randomized search and this presently seems to be more of an art than a science.

When an interesting solver for $\mathbb{E}$ is found by the search procedure, it is not particularly complicated to recursively generate a C++ implementation thereof (*i.e.*, a function that takes as input the parameters, and returns the solutions of the system of equations), or a text file that describes which variables to enumerate, which tables to join, in a nearly human-readable language. The generated C++ files are not very optimized.

2.3.1 Other Settings and Open problems

In its present form, our algorithms are suited to situations where all the solutions of the given equations are wanted. However some other problems concerning AES-like equations are interesting as well.

Reducing the Memory Complexity

We saw how to combine two solvers $\mathcal{A}_1$ and $\mathcal{A}_2$ of $\mathbb{E}(V(\mathcal{A}_1))$ and $\mathbb{E}(V(\mathcal{A}_2))$ into a solver for $\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2))$. However we can also combine $\mathcal{A}_1$ with a solver $\mathcal{A}_3$ of $\mathbb{E}_{V(\mathcal{A}_1)}(V(\mathcal{A}_3))$ (*i.e.*, the subsystem $\mathbb{E}(V(\mathcal{A}_3))$ where the variables $V(\mathcal{A}_1)$ are seen as parameters) into a solver for $\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_3))$ denoted by $\mathcal{A}_1 \to \mathcal{A}_3$. This is done by first running $\mathcal{A}_1$ and then for each solution returned running $\mathcal{A}_3$. It is fairly easy to see that this sequential join cannot lead to faster solvers than the original one but may save some memory. Unfortunately, if the sequential join is allowed then Theorem 3 about variables occurring linearly does not hold anymore. As a consequence, to be sure to find the best solver we now have q^{2n} base solvers: one for each linear combination of the x_i's and $S(x_i)$'s. As a result, the exhaustive search seems far from practicality.

Note that for guess-and-determine solvers, both the sequential and parallel joins give the same solvers.

Underdefined System of Equations

If there are much more variables than equations, the number of solutions will be overwhelming, and returning them all will be very expensive (and often unnecessary). A typical example is the case of collisions in hash functions (there are many, yet a single one is sufficient). A possible workaround is to allow solvers returning only a part of the solutions. More precisely, we can assume that $|\mathcal{S}ol\,(\mathcal{A}_1 \bowtie \mathcal{A}_2)\,| \approx |\mathcal{S}ol\,(\mathcal{A}_1)\,| \times |\mathcal{S}ol\,(\mathcal{A}_2)\,| \times p$, where p is the probability that a solution of $\mathbb{E}(V(\mathcal{A}_1)) \times \mathbb{E}(V(\mathcal{A}_2))$ is a solution of $\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2))$:

$$p = \frac{|\mathcal{S}ol\,(\mathbb{E}(V(\mathcal{A}_1) \cup V(\mathcal{A}_2)))\,|}{|\mathcal{S}ol\,(\mathbb{E}(V(\mathcal{A}_1)))\,| \times |\mathcal{S}ol\,(\mathbb{E}(V(\mathcal{A}_2)))\,|}.$$

As a base solver only guesses a variable we can obtain $0 \leqslant t \leqslant q$ solutions of $\mathbb{E}(\{x\})$ in t operations. Then, following a similar strategy than previously we can build the fastest solver giving t solutions for any $0 \leqslant t \leqslant |\mathcal{S}ol\,(\mathbb{E})\,|$. However some issues come up. First the heuristic assumption on the number of solutions holds globally but not locally, and the resulting complexity is often underestimated, in particular when the number of solutions wanted is low. Then, unlike what happens when looking for all the solutions, the sequential join may lead to better solvers, making the exhaustive search infeasible.

Restricting Possible Values of Some Variables

Given a subset $\mathbf{Y}$ of $\mathbf{X}$, an other interesting problem is to find the best solvers $\mathcal{A}$ such that $\mathbf{Y} \subseteq V(\mathcal{A})$. For instance, it is common that an attack partially encrypts or decrypts some data requiring to enumerate all the values of some key bytes related by a keyschedule. A solver enumerating them can be used as sub-component of this attack and the resulting complexity depends on how many values they can assume and how fast we can enumerate them.

The most naive way to find such solvers is to process each subset between $\mathbf{Y}$ and $\mathbf{X}$. However, only the ones stable by Propagate and without linear variables (excepted $\mathbf{Y}$) have to be considered. Furthermore, in most cases we can assume that the solver cannot solve the full system since otherwise it seems pointless to use it as a sub-component of an other attack.

Algorithm 7: TweakedTool (naive implementation)

Data: System of equations $\mathbb{E}$ in variables $\mathbf{X}$ involving some S-boxes and a subset $\mathbf{Y} \subsetneq \mathbf{X}$.

Result: A list of optimal algorithms to enumerate all the possible values of Y according to the system of equations E with predictable time and memory complexities.

$L \leftarrow \emptyset$;
foreach $\mathbf{Y} \subseteq \mathbf{Z} \subsetneq X$ **do**
  $\mathbf{V} \leftarrow \text{Propagate}^{\star}_{\mathbb{E}}(\mathbf{Z})$;
  if $\mathbf{V} \neq \mathbf{X}$ **then** $L \leftarrow L \cup \{\texttt{OriginalTool}(\mathbb{E}(V))\}$;
end
return L

Making a System Easier to Solve

Using the tool requires some knowledge of the primitive under scrutiny. For instance, on two AES rounds, two truncated differential paths with probability one yield two very different results: if the 4 active byte are on the same column, the tool finds an attack of complexity about 2^8, whereas if the active bytes are on a diagonal, the best attack found by the tool has complexity 2^{32}. More generally, it would be interesting to know which equations add to a system in order to make it easier to solve, leading to a trade-off between time and data complexities.

Chapitre 3

Low Data Complexity Attacks on Round-Reduced AES-128

We present a collection of low-data complexity attacks on round-reduced versions of the AES-128. These attacks have been found in the course of a joint work with Bouillaguet, Dunkelman, Keller and Rijmen, during which the automated tool of the previous chapter have been developed. Some of these results led to the submission of a journal paper [BDD+12], while some others were presented in [BDF11].

During the course of 2009 and 2010, Dunkelman and Keller announced in several occasions that they were investigating low-data complexity attacks against the AES, and announced interesting results. We developed the automated tool described in the previous chapter hoping to catch up on their results. This effort has been gratifying, as the tool could often improve on the manually-found attacks. When it is the case, it is interesting to compare both the manually-found and the automatically-found attacks.

3.1 Low Data Complexity Attacks

The field of block cipher design has advanced greatly in the last two decades. New strategies of designing secure block ciphers were proposed, and following the increase in computing power, designers could other larger security margins with reduced performance penalties. As a result, practical attacks on block ciphers became extremely rare, and even "certificational attacks" (that is, attacks which are not practical but are still faster than exhaustive key search on the full version of the cipher), are not very common, while the situation of hash functions and also stream ciphers is dramatically different. Several commonly used hash functions were practically broken in recent years [SSA+09, MP08], and practical attacks on new stream cipher designs appear every several months [SHJ09, HJ11].

This led to two approaches in the block-cipher cryptanalysis community. The first one is to concentrate on attacking reduced-round variants of block ciphers, where the usual goal of the adversary is to maximize the number of rounds that can be broken, using less data than the entire codebook and less time than exhaustive key search. This approach usually leads to attacks with extremely high data and time complexities, such as those

shown in [DS08, LDKK08]. The second approach is to allow the adversary more degrees of freedom in his control. Examples of this approach are attacks requiring adaptive chosen plaintext and ciphertext queries, the related-key model, the related-subkey model, and even the known-key model [BK09, DKS10b, KR07]. This approach allows to achieve practical complexities even against widely used block ciphers such as the AES, but the practicality of the models themselves in real-life situations can be questioned. Attacks following each of these approaches are of great importance, as they ensure that the block ciphers are strong enough, almost independently of the way in which they are deployed. Moreover, they help to establish the security margins offered by the ciphers. A block cipher which is resistant to attacks when the adversary has a strong control and almost unrestricted resources offers larger security margins than a block cipher which does not possess this resistance. At the same time, concentrating the cryptanalytic attention only on such attacks may prove insufficient to truly understand the security of the analyzed block cipher. It seems desirable to also consider other approaches, such as restricting the resources available to the adversary in order to adhere to "real-life" scenarios. For example, one may study the maximal number of rounds that can be broken with practical data and time complexity, as considered for instance in [BDK+10] with respect to the related-key model.

Low Data Complexity Attacks. In this part we pursue this direction of research, but we concentrate on another restriction of the adversary's resources. In the attacks we consider, the time complexity is not restricted (besides the natural bound of exhaustive search), but the data complexity is restricted to only a few known or chosen plaintexts. At first glance, this scenario may seem far fetched. However, it makes sense to question whether it is easier in practice for an attacker to perform 2^{50} elementary operations or to acquire 50 plaintext/ciphertext pairs. We are inclined to believe that in many actual-life situations, performing the computation is easier. In addition, the EMV protocol for credit cards specifies that at most 2^{16} signatures shall be issued by a given chip, which practically restricts the quantity of available data. It also turns out that this setup is very natural in the context of several classes of attacks:

1. *Slide attacks [BW99]*: This class of attacks is especially designed against block ciphers whose rounds are very similar to each other. The main feature of slide attacks is that they are independent of the number of rounds, and thus, the common countermeasure of increasing the number of rounds is not effective against them. Since most other attack techniques can be easily undermined by adding a few rounds, this makes the slide attacks one of the most powerful attacks against modern block cipher designs. The main idea of the slide attacks is to reduce the attack on the entire block cipher to an attack on a single round, where the data available to the adversary is only two known plaintext/ciphertext pairs. Hence, the scenario considered in our paper is exactly the one faced by the adversary in the slide attack. We note that several variants of slide attacks suggested methods to increase the amount of data available to the adversary [BW00, BDK07, Fur01]. However, all these methods either require the knowledge of large portion of the codebook or perform in the adaptively chosen plaintext model.
2. *Attacks based on fixed point properties [CBW08]*: In this class of attacks, the adver-

sary looks for a fixed point of some part of the encryption process. For such a fixed point, the cipher is reduced to a smaller variant, which can (sometimes) be attacked efficiently. Since usually the number of fixed points is extremely small (e.g., one or two), the adversary's goal is to attack a reduced-round variant of the cipher given a few known plaintexts.

3. *Side channel attacks*: In this class of attacks, the adversary has access to some information on the internal states during the encryption process. Usually, due to practical restrictions, the amount of data available to the adversary is extremely low. In the (somewhat unlikely) case where the information available to the adversary is the full intermediate state after a few rounds, the scenario the adversary faces is exactly the one considered in our paper. We note that the complementary scenario, where the adversary has access to a small part of the internal state in multiple encryptions, was studied in [DS09].

4. *Building block in more complex attacks*: As we will demonstrate on the example of AES, an attack on 2-round AES with two known plaintexts can be leveraged to a known plaintext attack on 6-round AES. The attack uses a meet-in-the-middle approach combined with a low probability differential. Also, the block cipher GOST was recently broken by such an attack, where an attack against the full 32 rounds is reduced to an attack against 8 rounds using four known plaintext [Iso11]. We expect that such "leveraging" attacks are applicable against other block ciphers as well.

5. *Attacks on other primitives based on the block cipher*: In recent years, many designs of stream ciphers (e.g., Sosemanuk [BBC+08]), hash functions (e.g., Hamsi [Kuc09]), MACs (e.g., Alpha-MAC [DR05b]), etc., use a small number of rounds of a block cipher as one of their components. Some of these constructions can be broken when internal collisions are found and thus the attackers are in a setting in which they have to exploit a very limited quantity of data (the colliding inputs). The attacks we consider can be used against these primitives, once collisions are found (which may require a large quantity of data though).

The AES, a Natural Target. In order to make our results concrete, we have chosen to concentrate on a single block cipher – the AES, the *Advanced Encryption Standard* [NIS01]. The AES is a 128-bit block cipher with a variable key length (128, 192, and 256-bit keys are supported). Since its selection, AES gradually became one of the most widely used block ciphers and received a great deal of cryptanalytic attention, both during the AES process, and even more after its selection. Studying reduced-round versions of AES is also motivated by the recent blossom of many AES-based primitives for hashing or authentication, such as the Grostl, ECHO, SHAVite-3 and LANE hash functions, the LEX [Bir08a] stream cipher, or the Alpha-MAC [DR05b] and Pelican-MAC [DR05c] message authentication codes. In these construction, AES rounds (and sometimes the full AES) are used as internal permutations. A possible explanation of this fancy is that the AES enjoys very interesting security properties against statistical attacks: two rounds achieve full diffusion, and there exist very good differential and linear lower bounds for the best differential on four rounds [KMT01a, KMT01b, Kel04]. It results that in some

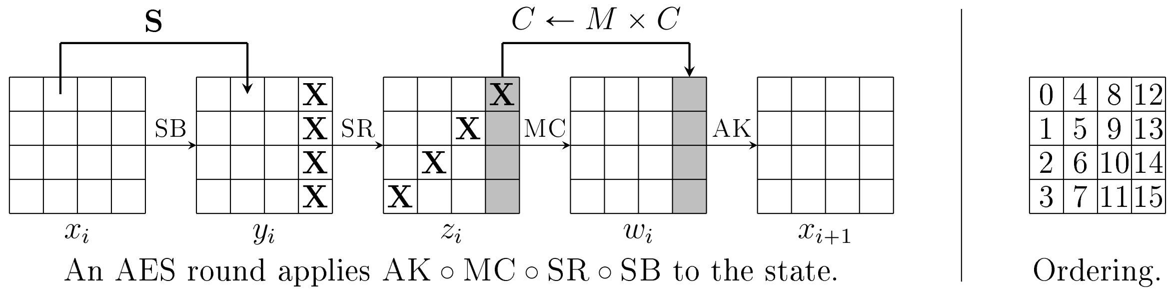


Figure 3.1: Description of one AES round and the ordering of bytes in an internal state.

applications (such as authentication or hashing), only a small number of AES rounds are sufficient to yield a reasonable internal permutation. The relative weakness of the permutation is then usually compensated by the fact that the internal state is either hidden from the adversary (in MACs, because of the secret key), or only partially accessible to the adversary (because only certain parts can be seen and modified). Furthermore, in some particular attacks, such as side-channel attacks, only a small number of rounds of the cipher needs to be studied [PQ03, BK07a]. Lastly, much attention has been recently devoted to the AES block cipher as a by-product of the NIST SHA-3 competition. The low diffusion property of the key schedule has been used to mount several related-key attacks [BKN09, BK09, BDK+10, KBN09] and differential characteristic developed for hash functions have been used to also improve single-key attacks [DKS10a]. The AES is therefore a relevant and interesting case study to demonstrate our techniques.

3.2 Description of the AES

The Advanced Encryption Standard [NIS01] is a Substitution-Permutation network that supports key sizes of 128, 192 and 256 bits. The 128-bit plaintext initializes the internal state viewed as a 4×4 matrix of bytes, where each byte represents an element from $\mathbb{F}_{2^8}$. This field is defined using the irreducible polynomial $X^8 + X^4 + X^3 + X + 1$ over $\mathbb{F}_2$. Depending on the key length, N_r rounds are applied to that state: $N_r = 10$ for 128-bit keys, $N_r = 12$ for 192-bit keys, and $N_r = 14$ rounds for 256-bit keys. An AES round applies four operations to the state matrix:

- `SubBytes` (SB) applies the same 8-bit to 8-bit invertible S-Box S 16 times in parallel on each byte of the state,
- `ShiftRows` (SR) shifts the i-th row left by i positions,
- `MixColumns` (MC) replaces each of the four column C of the state by $M \times C$ where M is a constant 4×4 maximum distance separable matrix over $\mathbb{F}_{2^8}$,
- `AddRoundKey` (AK) adds a 128-bit subkey to the state.

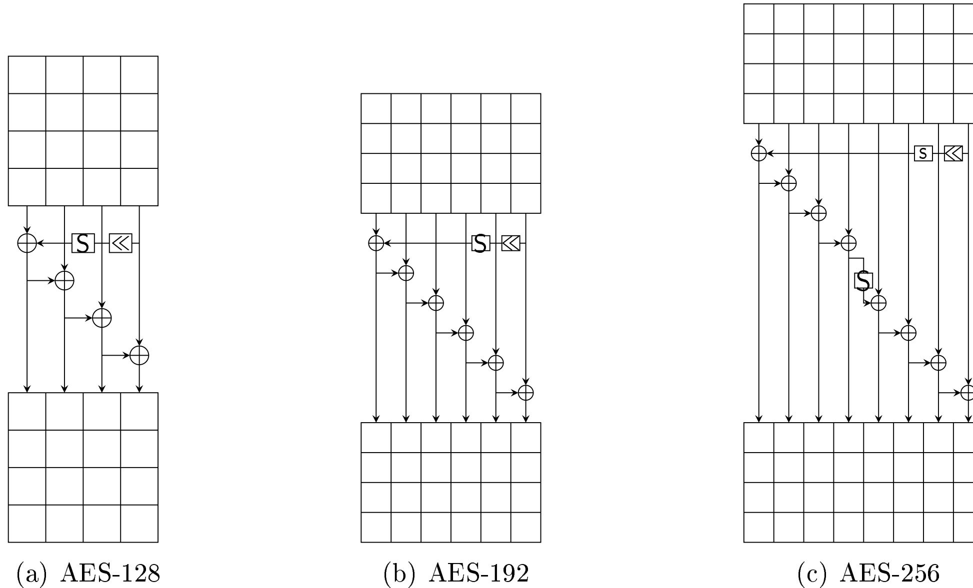
We outline an AES round in Figure 3.1. Before the first round, an additional `AddRoundKey` operation (using a whitening key) is applied, and in the last round the `MixColumns` operation is omitted.

Let $\mathbb{F}_{2^8}$ be the finite field with 256 elements used in the AES. We represent the S-box of the `SubBytes` transformation by $S : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$. In a 4×4 matrix, we use the following

numbering of bytes: byte zero is the top-left corner, the first column is made of bytes 0-3, while the last column is made of bytes 12-15, with byte 15 in the bottom-right corner (this is illustrated by Figure 3.1). We denote the four columns of a 4×4 matrix M by $M[0..3], M[4..7], M[8..11]$ and $M[12..15]$ respectively. We count the AES rounds from 0 and, in the i-th round, we denote the internal state after `AddRoundKey` by x_i, after `SubBytes` by y_i, after `ShiftRows` by z_i and after `MixColumns` by w_i. To refer to the difference in a state x, we use the notation Δx. Because the final AES round is different from the others, we use the term "r.5 rounds AES" to denote the AES reduced to $(r+1)$ rounds, including the final round. We use "r rounds AES" to denote the AES reduced to r identical full rounds. In our terminology, the "normal" 128-bit AES has 9.5 rounds.

The key expansion algorithms to produce the N_r+1 subkeys are described in Figure 3.2 for each keysize. As the AES-128 will be studied more intensively we shall describe this version more precisely and refer to the original publication [NIS01] for further details about the two other versions. The key schedule of AES-128 takes the 128-bit master key k_0 and extends it into 10 subkeys $k_1, \ldots, k_{10}$ of 128 bits each using a key-schedule algorithm given by the following equations:

$$KS_i : \begin{cases} k_i[j] & + \quad k_i[j-4] + k_{i-1}[j] = 0, \qquad\qquad j = 4, \ldots, 15 \\ k_i[0] & + \quad k_{i-1}[0] + S\left(k_{i-1}[13]\right) + \mathtt{RCON}_i = 0 \\ k_i[1] & + \quad k_{i-1}[1] + S\left(k_{i-1}[14]\right) = 0 \\ k_i[2] & + \quad k_{i-1}[2] + S\left(k_{i-1}[15]\right) = 0 \\ k_i[3] & + \quad k_{i-1}[3] + S\left(k_{i-1}[12]\right) = 0 \end{cases}$$



(a) AES-128 (b) AES-192 (c) AES-256

Figure 3.2: Key schedules of the variants of the AES: AES-128, AES-192 and AES-256.

In some cases, we are interested in interchanging the order of the `MixColumns` and `AddRoundKey` operations. As these operations are linear they can be interchanged, by

first XORing the data with an equivalent key and only then applying the `MixColumns` operation. We denote the equivalent subkey for the altered version by:

$$u_i = MC^{-1}(k_i) = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \times k_i$$

3.3 Observations on the Structure of AES

In this section we present well-known observations on the structure of AES, that we use in our attacks. We first consider the propagation of differences through `SubBytes`, which is the only non-linear operation in AES.

Property 12 (the `SubBytes` property)**.** *Consider pairs $(\alpha \neq 0, \beta)$ of input/output differences for a single S-box in the* `SubBytes` *operation. For 129/256 of such pairs, the differential transition is impossible, i.e., there is no pair (x, y) such that $x \oplus y = \alpha$ and $S(x) \oplus S(y) = \beta$. For 126/256 of the pairs (α, β), there exist two ordered pairs (x, y) such that $x \oplus y = \alpha$ and $S(x) \oplus S(y) = \beta$, and for the remaining 1/256 of the pairs (α, β) there exist four ordered pairs (x, y) that satisfy the input/output differences. Moreover, the pairs (x, y) of actual input values corresponding to a given difference pattern (α, β) can be found instantly from the difference distribution table of the S-box. We recall that the time required to construct the table is 2^{16} evaluations of the S-box, and the memory required to store the table is about 2^{17} bytes.*

Property 12 means that given the input and output difference of an S-box, we can find in constant time the possible absolute values of the input, and there is only a single one on average.

The second observation uses the linearity of the `MixColumns` operation, and follows from the structure of the matrix used in `MixColumns`:

Property 13 (the `MixColumns` property)**.** *Consider a pair (a, b) of 4-byte vectors, such that $a = MC(b)$, i.e., the input and the output of a* `MixColumns` *operation applied to one column. Denote $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$ where a_i and b_j are elements of $\mathbb{F}_{256}$. The knowledge of* any *four out of the eight bytes $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$ is sufficient to* uniquely *determine the value of the remaining four bytes.*

The third observation is concerned with the keyschedule of AES-128, and exploits the fact that most of the operations in the algorithm are linear. It allows the adversary to get relations between bytes of non-consecutive subkeys (*e.g.*, k_r, k_{r+3} and k_{r+4}), while "skipping" the intermediate subkeys. The observation extends previous observations of the same nature made in [FKL+00, DK10a].

Property 14 (the key-schedule properties). *Consider a series of consecutive subkeys $k_r, k_{r+1}, \ldots$, and denote $k_r = (a, b, c, d)$ and:*

$$\begin{aligned} u &= \textsf{RotBytes}(\texttt{SubBytes}(k_r[12..15])) \oplus RCON[r+1] \\ v &= \textsf{RotBytes}(\texttt{SubBytes}(k_{r+1}[12..15])) \oplus RCON[r+2] \\ w &= \textsf{RotBytes}(\texttt{SubBytes}(k_{r+2}[12..15])) \oplus RCON[r+3] \\ x &= \textsf{RotBytes}(\texttt{SubBytes}(k_{r+3}[12..15])) \oplus RCON[r+4] \end{aligned}$$

Then, the subkeys $k_{r+1}, k_{r+2}, \ldots$ can be represented as linear combinations of (a, b, c, d) (the columns of k_r) and the 32-bit words u, v, w, x, as shown in the following table:

Round	$k[0..3]$	$k[4..7]$	$k[8..11]$	$k[12..16]$
r	a	b	c	d
$r+1$	$a \oplus u$	$a \oplus b \oplus u$	$a \oplus b \oplus c \oplus u$	$a \oplus b \oplus c \oplus d \oplus u$
$r+2$	$a \oplus u \oplus v$	$b \oplus v$	$a \oplus c \oplus u \oplus v$	$b \oplus d \oplus v$
$r+3$	$a \oplus u \oplus v \oplus w$	$a \oplus b \oplus u \oplus w$	$b \oplus c \oplus v \oplus w$	$c \oplus d \oplus w$
$r+4$	$a \oplus u \oplus v \oplus w \oplus x$	$b \oplus v \oplus x$	$c \oplus w \oplus x$	$d \oplus x$

As a result, we have the following useful relations between subkeys:

1. $k_{r+2}[0..3] \oplus k_{r+2}[8..11] = k_r[8..11]$,
2. $k_{r+2}[4..7] \oplus k_{r+2}[12..15] = k_r[12..15]$,
3. $k_{r+2}[4..7] \oplus v = k_r[4..7]$,
4. $k_{r+4}[12..15] \oplus x = k_r[12..15]$,
5. $k_{r+3}[12..15] = k_r[8..11] \oplus k_r[12..15] \oplus w$.

Similar observations on both the keyschedules of AES-192 and AES-256 are made Chapter 7.

3.4 Attack on One-Round AES

We start our analysis with the simplest case, an adversary who seeks to break one full round of AES (a sequence of `AddRoundKey`, `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` operations)

3.4.1 Two Known Plaintexts

We first describe a simple but suboptimal attack. It starts by applying $SR^{-1} \circ MC^{-1}$ to the ciphertext difference, to obtain the output differences of all the S-boxes. Since the input differences of the S-boxes are equal to the plaintext difference in the respective bytes, the adversary can consider each S-box independently, go over the 2^8 possible pairs of inputs whose difference equals the plaintext difference, and find the pairs suggesting

the "correct" output difference. In each S-box, the expected number of suggested pairs is two, and each such pair gives a suggestion of one byte in the subkey k_0. Thus, the adversary gets 2^{16} suggestions for the entire subkey k_0, which can be checked by trial encryption.

This attack, whose time complexity is 2^{16} encryptions, can be further improved using the relation between the subkeys k_0 and k_1. If the adversary checks the S-boxes in bytes 0, 5, 10 and 15 she can use the $2^4 = 16$ suggestions of output values of these S-boxes to get 16 suggestions for the column $k_1[0..3]$, along with bytes 0, 5, 10 and 15 of k_0. Similarly, checking bytes 3, 4, 9, 14 yields 16 suggestions for the column $k_1[4..7]$, along with bytes 3, 4, 9, 14 of k_0. Combining the suggestions, the adversary obtains 256 suggestions for two columns of k_1 and eight bytes of k_0. At this stage, the adversary can use the relation $k_1[4] = k_0[4] + k_1[0]$ which holds by the AES key schedule, as a consistency check. Only a single suggestion is expected to remain. The value of the remaining 8 bytes of k_0 can be obtained similarly by examining the other eight S-boxes. This improvement reduces the time complexity of the attack to 2^{12} S-box applications.

3.4.2 One Known Plaintext

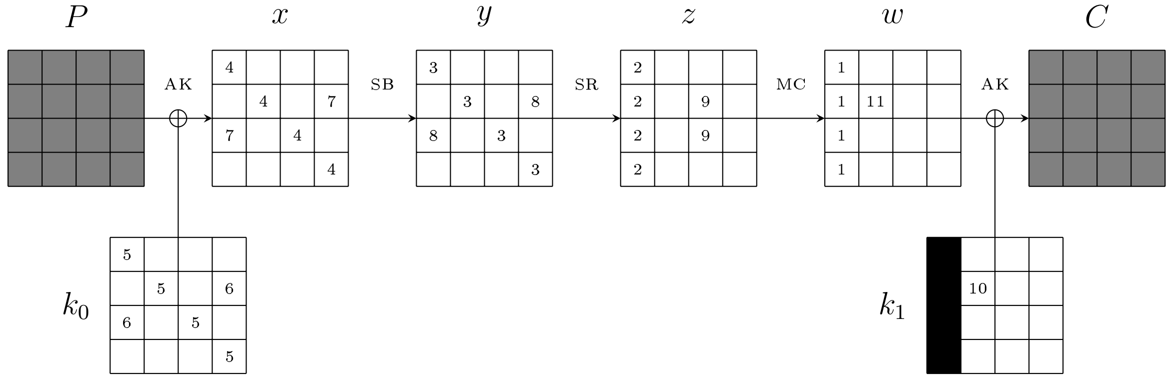
If the data available to the adversary is only a single plaintext, then the attack must use the relation between the two subkeys k_0 and k_1. If the subkeys were independent, then the information available to the adversary would not be sufficient to retrieve the key uniquely. Since any relation between the plaintext and the ciphertext involves the `MixColumns` operation, it seems likely that any such attack should require the guess of a full column, and thus have complexity of at least 2^{32} encryptions.

In this setup, the best attack found manually is a guess-and-determine attack by Dunkelman and Keller [DK10b] that has a running time equivalent to that of 2^{48} encryptions. The tool described in the previous chapter was able to find an attack with time complexity of 2^{32} encryptions, and memory requirement of 2^{16} bytes. Restricted to guess-and-determine solver an attack with time complexity of 2^{40} encryptions and a negligible memory requirement was found, which is still better than Dunkelman and Keller's one.

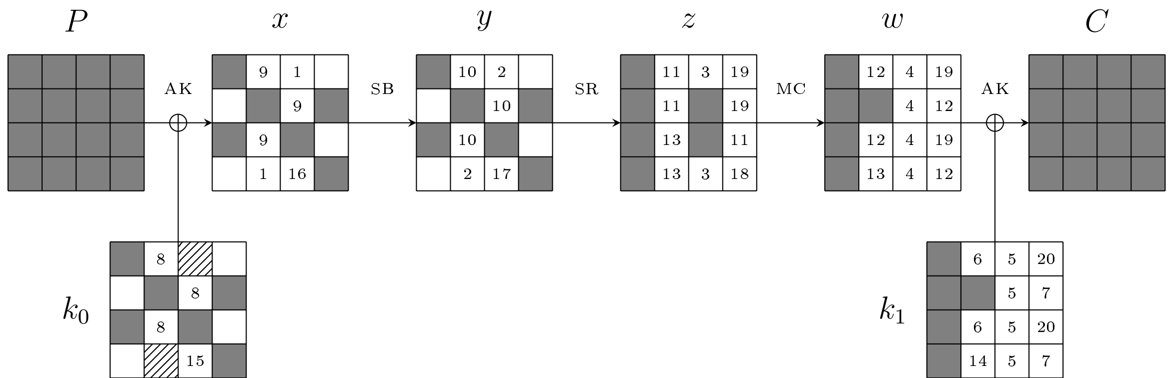
The attack, depicted in Figure 3.3, is based on Property 15 below. In the first phase of the attack, the adversary guesses the column $k_1[0..3]$, and retrieves the value of seven additional subkey bytes. This phase is shown in Figure 3.3(a), where steps 6 and 10 are based on key schedule arguments, and the rest of the steps use the application of known operations on known values. The second phase of the attack, depicted in Figure3.3(b), starts with retrieving the possible values of bytes $k_0[7]$ and $k_0[8]$ using Property 15. Steps 6, 7, 8 and 15 use the key schedule, steps 13 and 19 use Property 13, and the rest of the steps follow the application of AES' operations to known values.

The key step of the attack is the discovery of Property 9.5 by the tool of Chapter 2. We went though the output produced by the tool to write a readable description of the solver's steps.

Property 15. *The knowledge of P, C and the column $k_1[0..3]$ allows one to retrieve bytes $k_0[7]$ and $k_0[8]$ by a table look-up to a precomputed table of size 2^{24}. For each value of $k_1[0..3]$, there are at most 12values of $(k_0[7], k_0[8])$, and a single one on average. The*



(a) First half of the attack



(b) Second half of the attack

Figure 3.3: An attack on one round AES given one known plaintext with time complexity of 2^{32} and memory complexity of 2^{24}. Bytes marked by gray are known (from previous steps of analysis). Bytes marked with tilted lines are retrieved using Property 15.

time complexity required to generate the table is 2^{32} *operations.*

The proof of this property is based on obtaining two non-linear 8-bit relations involving two key bytes (given that other bytes of the key and the state are known). In such a case, we expect on average one solution to these equations (and as our test shows, in reality the maximal number of solutions is 12). Moreover, as there are 2^{32} possible systems, one can precompute the acceptable solutions and store them.

Proof. First, we note that the knowledge of the column $k_1[0..3]$ along with the plaintext and the ciphertext allows us to retrieve one additional byte of k_1 and six bytes of k_0, as shown in Figure 3.3(a). We denote $a = y_0[8]$ and $b = y_0[7]$, and express several other bytes in terms of a, b, and known bytes (from the plaintext, the ciphertext and $k_1[0..3]$). Our goal is to obtain a system of two equations in a and b, and to solve it using a precomputed table. This system is constructed in 5 steps.

1. First, note that the third column of k_1 can be expressed as:

$$\begin{aligned} k_1[8..1] &= C[8..11] + w_0[8..1] \\ &= C[8..11] + MC(z_0[8..11]) \\ &= C[8..11] + MC\left({}^t(a, S(P[13] + k_0[13]), S(P[2] + k_0[2]), b)\right). \quad (3.1) \end{aligned}$$

   In this expression, bytes 2 and 13 of k_0 can be deduced from the 4 guessed bytes in k_1 after the first phase of the attack, as shown in Figure 3.3(a).

2. Let us now turn our attention towards the second column of k_1. Note that $k_1[5]$ can be computed following the procedure shown in Figure 3.3(a), and the remaining three bytes of $k_1[4..7]$ can be expressed as:

$$\begin{cases} k_1[4] &= k_1[8] + S^{-1}(a) + P[8] \\ k_1[6] &= k_1[10] + k_0[10] \\ k_1[7] &= k_1[3] + S^{-1}(b) + P[7] \end{cases} \quad (3.2)$$

   Again, $k_0[10]$ (appearing in the expression of $k_1[6]$) can be derived from the guessed bytes.

3. Next, we will turn our attention away from the key-schedule, to the second column of z_0, and in particular to $z_0[4]$ and $z_0[5]$. We first express it in function of the plaintext. It follows from the definition of the encryption and schedule algorithm that:

$$\begin{cases} z_0[4] &= S(P[4] + k_0[4]) &= S(P[4] + k_1[0] + k_1[4]) \\ z_0[5] &= S(P[9] + k_0[9]) &= S(P[9] + k_1[5] + k_1[9]) \end{cases} \quad (3.3)$$

   It must be noted that all key bytes occurring in these expression can be readily derived from a, b and the four guessed key bytes.

4. On the other hand, the whole column $z_0[4..7]$ can be expressed as a function of the ciphertext:

$$z_0[4..7] = MC^{-1}(w_0[4..7]) = MC^{-1}\left(C[4..7] + k_1[4..7]\right). \quad (3.4)$$

5. Identifying $z_0[4]$ and $z_0[5]$ in (3.3) and (3.4), and exploiting (3.1) and (3.2) results in a system of two linearly independent equations in the unknowns a, b, and the known plaintext, ciphertext, and bytes that can be derived from $k_1[0..3]$. These equations can be written in the form:

$$\begin{cases} \Delta_1 &= f_1(a,b) + S(f_2(a,b) + \Delta_2) \\ \Delta_3 &= f_3(a,b) + S(f_4(a,b) + \Delta_4) \end{cases} \quad (3.5)$$

   where f_1, f_2, f_3 and f_4 are fixed known functions, and Δ_1, Δ_2, Δ_3 and Δ_4 are one-byte parameters depending on the plaintext, the ciphertext, and the guessed subkey bytes (the actual expressions are not given as they are a bit lengthy).

Equations (3.5) allows to perform the following two-phase procedure:

Offline Phase: for each of the 2^{32} possible values of $(a, b, \Delta_2, \Delta_4)$, evaluate the right-hand in (3.5), and find the values of Δ_1 and Δ_3. Store the pair (a, b) in a data-structure (typically an array of linked lists, or a hash table) indexed by $(\Delta_1, \Delta_2, \Delta_3, \Delta_4)$. This phase requires 2^{32} simple operations and 2^{32} 16-bit blocks of storage.

Online Phase: once the values of P, C are known, and for each guess of $k_1[4..7]$, compute the value $(\Delta_1, \Delta_2, \Delta_3, \Delta_4)$, and obtain from the precomputed table the corresponding values of (a, b). Then deduce the subkey bytes $k_0[7]$ and $k_0[8]$ using the equations:

$$\begin{array}{rcccl} k_0[8] & = & P[8] + x_0[8] & = & P[8] + S^{-1}(a) \\ k_0[7] & = & P[7] + x_0[7] & = & P[7] + S^{-1}(b) \end{array} \tag{3.6}$$

Reducing the Memory Complexity: We can reduce the 2^{32} memory required for storing the solutions to only 2^{16}. This is done by fixing the value of two particular linear combinations of the Δ_i's, such that we can deal only with the equations relevant to this specific value in each step of the attack. Those linear combinations are chosen such that from their knowledge it is easy to enumerate the 2^{16} corresponding first columns of k_1 as well as the 2^{16} corresponding values of $(a, b, \Delta_2, \Delta_4)$. The outcome of such an approach is the ability to reduce the number of possible systems that we need to consider in a given time to 2^{16}, which reduces the memory complexity without affecting the time complexity. □

3.5 Attacks on Two-Round AES

In this section we consider attacks on two rounds of AES, denoted by rounds 1 and 2. First we present attacks on two *full* rounds with two known plaintexts. We then study the interesting case of two *chosen* plaintext. In both settings, the tools vastly outperformed human cryptanalysts. We then look at the case of a single known plaintext. We conclude by presenting an improved attack with two known plaintexts that can be applied if the `MixColumns` operation in round 1 is omitted (*i.e.*, if we are facing 1.5 rounds). This attack is used as a procedure in our attack on 6-round AES presented in.

3.5.1 Two Known Plaintexts.

The Manually-Found Attack. We first describe an attack found manually by Dunkelman and Keller. This attack with two known plaintexts, depicted in Figure 3.4, is based on Property 12. As in the one-round attack with two known plaintexts, we observe that the ciphertext difference allows us to retrieve the intermediate difference after the SubBytes operation of round 2. This observation is used in both phases of the attack. We also "swap" the order of the MixColumns and the AddRoundKey operations of the second round. This can be done since both operations are linear, as long as the subkey k_2 is replaced by the equivalent subkey $u_2 = MC^{-1}(k_2)$.

In the first phase of the attack, the adversary guesses bytes 0, 5, 10 and 15 of k_0, which allows her to retrieve the intermediate difference in $x_2[0..3]$ (*i.e.*, just before the

SubBytes operation of round 2). Then, Property 12 can be applied to the four S-boxes in that column, yielding their actual input/output values in both encryptions. This in turn allows us to obtain $k_1[0..3]$ (as the values before the AddRoundKey with k_1 are known). At this stage, the adversary tries to deduce and compute as many additional bytes as she can.

In the second phase of the attack, the adversary guesses two additional subkey bytes ($k_0[7]$ and $k_0[8]$) which are sufficient to retrieve the intermediate difference in $x_1[8..11]$. Then, Property 12 can be applied to the four S-boxes in bytes 8-11 of round 2.

We note that the while the first phase of the attack allows us to obtain several bytes in u_2, the knowledge of these bytes cannot be combined directly with the knowledge of bytes in k_1 and k_0, since u_2 does not satisfy the equations of the key schedule algorithm. Hence, in the second phase of the attack, we obtain bytes in both u_2 and k_2 in parallel, and apply Property 13 to the relation between k_2 and u_2, since they are the input and output of a MixColumns operation.

In Phase 1 of the attack, depicted in the top half of Figure 3.4, step 5 is based on Property 12, steps 9 and 13 exploit the key schedule, and the rest of the steps are performed using encryption/decryption. In the second phase, depicted in the bottom half of the figure, step 5 is based on Property 9.1, step 12 uses Property 13 applied to the relation between k_2 and u_2, steps 9, 10, 11 and 13 exploit the key schedule, and the rest of the steps are performed using encryption/decryption.

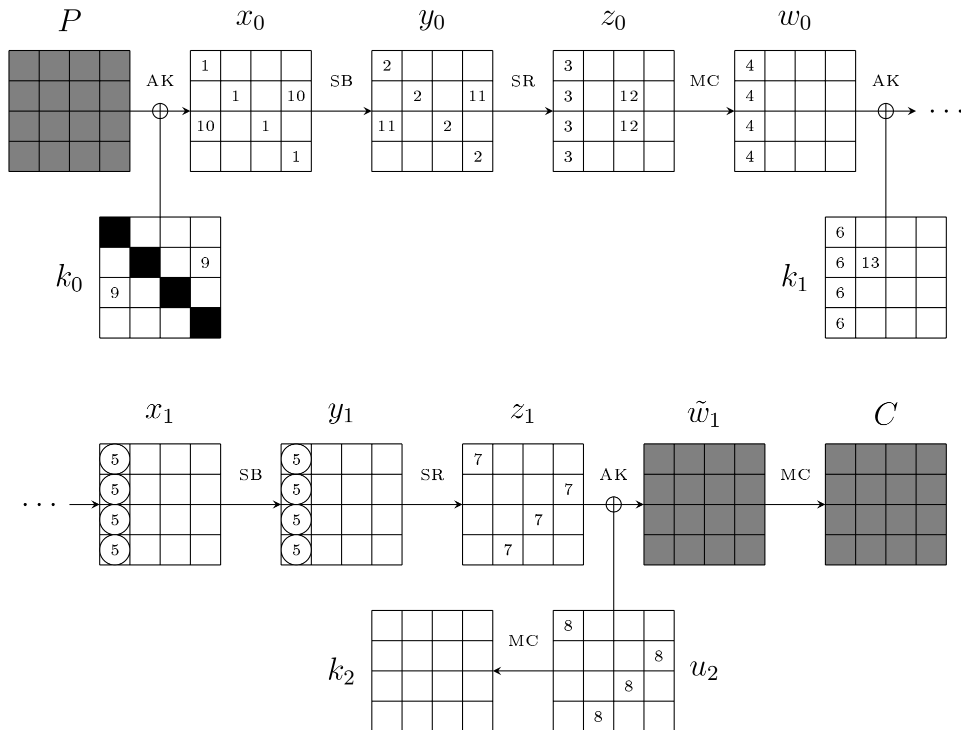
The time complexity of the attack is determined by the fact that 6 subkey bytes are guessed, and for each guess a few simple analysis steps are performed. Hence, the time complexity of the attack is 2^{48}.

The Automatically-Found Attack. Our tool has found an attack with time and memory complexity 2^{32} in this setting, vastly outperforming human cryptanalysts. The attack is a meet-in-the-middle whose main ingredient is the possibility to isolate a set of about 2^{32} candidates for both $k_1[0..3]$ and $k_1[12..15]$ with only 2^{32} operations. These 8 bytes are a sufficient to recover the full key with a complexity of about one encryption.

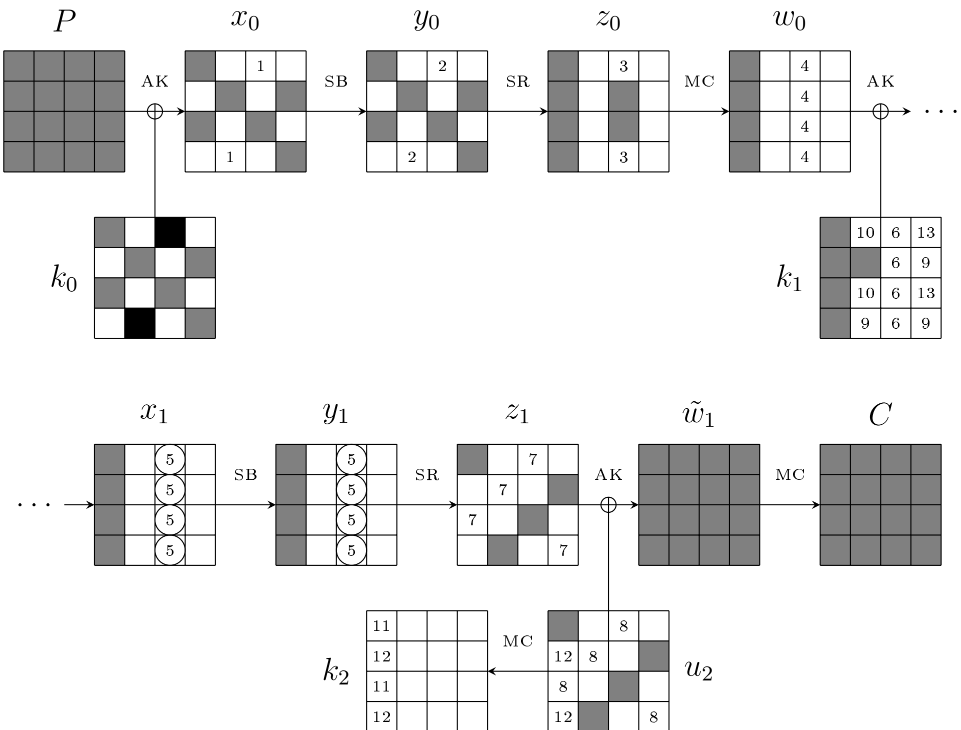
First, we assume that $x_1[12..15]$ is known (for the first message), and we try to derive the value of some other bytes. We can easily obtain the differences in $x_1[12..15]$. Then, by linearity of the `MixColumns` operation, we obtain the differences in $z_0[12..15]$. Using Property 12, we also obtain the values and the differences in byte 1, 6, 11 and 12 of x_0 (and thus of k_0). Note that the values of $w_0[12..15]$ and $k_1[12..15]$ are revealed in the process. Let us denote by A the set of bytes that can be obtained from $x_1[12..15]$.

Similarly, if the value of $x_1[0..3]$ is known, then the values (and differences) in byte 0,2, 5,10, 13 and 15 of x_0 and k_0, as well as $w_0[0..3]$ and $k_1[0..3]$ could be recovered. Let us denote these bytes by B.

Even though the bytes in $A \cup B$ can take 2^{64} values, this can efficiently be reduced to 2^{32}. Indeed, we claim that there exist (at least) 4 *linear* relations between bytes of A



(a) First half of the attack: The difference in the bytes marked is guessed. The bytes marked by 5 are found using the known input and output differences of the Sbox.



(b) Second Half: The difference in the bytes marked in black is guessed. The bytes marked by 5 are found using the known input and output differences of the Sbox. The bytes marked by 12 are found using the relation between the keys k_2 and u_2.

Figure 3.4: The attack with two known plaintexts on two round AES.

and those of B:

$$\begin{array}{rcl} f_1(A) & = & g_1(B) \\ f_2(A) & = & g_2(B) \\ f_3(A) & = & g_3(B) \\ f_4(A) & = & g_4(B) \end{array}$$

Thanks to these relations, a tuple of values from A is associated to a single tuple of values of B on average: for each one of the 2^{32} tuples of values in A, evaluate the f_i's and store the result in a hash table. Then for each one of the of the 2^{32} tuples of values in B, evaluate the g_i's, and loop-up the corresponding value(s) in A.

Two of these linear relations can be obtained very simply: given $k_1[0..3]$ and $k_1[12..15]$, we deduce $k_2[0..3]$. From there, it is also possible to compute bytes 0, 5, 10 and 15 from x_1 by partial decryption. Amongst these, $x_1[15]$ occurs in A while $x_1[0]$ occurs in B. This already gives two linear equations connecting A and B.

Two other constraints can be obtained in a more sophisticated way. First, we notice that given the key bytes in A and B, it is possible to retrieve the full k_2 except byte 4, 8 and 12 by just exploiting the key-schedule and Property 14. Focusing on the last two columns of w_1, we find that 3 bytes are known in each column in w_1 and two bytes are known in each column of z_1. Thanks to Property 13, this gives a linear relation between the known bytes of each column.

3.5.2 A Three Known Plaintext Variant

We note that if the adversary is given three known plaintexts, then a simpler attack can be applied, with the same complexity, namely 2^{32} encryptions. The adversary applies the first phase of the manually-found attack twice (for the pairs (P_1, P_2) and (P_1, P_3)), and uses the values of $k_1[0..3]$ retrieved in that phase for a consistency check. Since for the correct guess of bytes 0, 5, 10 and 15 of k_0, both pairs suggest the same value of the four bytes of k_1, and for an incorrect guess, the two pairs suggest the same value only with probability 2^{32}, this allows us to discard most of the wrong guesses. Then, the adversary performs the second phase of the attack only for the remaining guesses, and thus the time complexity of the attack is dominated by the first step, whose complexity is 2^{32} encryptions.

3.5.3 A Two Chosen Plaintext Variant

If the adversary is given two *chosen* plaintexts, then the time complexity can be reduced. We first describe an attack found manually by Dunkelman and Keller, with a complexity of 2^{28} encryptions. We will next describe an attack found by the improved tool, with complexity 2^8 (!).

The Manually-Found Attack. In order to improve on the known-plaintext scenario, the adversary asks for the encryption of two plaintexts which differ only in four bytes

composing one column. Figure 3.5 shows the difference pattern. In this case, at the end of round 1, there are exactly 127 possible differences in each column. For each such difference, the adversary can apply Property 12 to the four S-boxes of the column, and obtain one suggestion on average for the actual values after the SubBytes operation of round 2. Combining the values obtained from all four columns, the adversary gets about 2^{28} suggestions for the entire state after the SubBytes operation of round 2, and each such suggestion yields a suggestion of the subkey k_2. Thus, the time complexity of the attack is 2^{28} encryptions.

The Automatically-Found Attack. The adversary asks for the encryption of two plaintexts which differ only in four bytes composing one column. The attack relies on Property 16 below, which cleverly uses the linearity in the key-schedule of the AES.

Property 16. *For all $i \geqslant 1$ we have the following equations:*

1. $z_{i-1}[4..7] \oplus z_i[0..3] \oplus z_i[4..7] = MC^{-1}\Big(x_i[4..7] \oplus x_{i+1}[0..3] \oplus x_{i+1}[4..7]\Big)$
2. $z_{i-1}[8..11] \oplus z_i[4..7] \oplus z_i[8..11] = MC^{-1}\Big(x_i[8..11] \oplus x_{i+1}[4..7] \oplus x_{i+1}[8..11]\Big)$
3. $z_{i-1}[12..15] \oplus z_i[8..11] \oplus z_i[12..15] = MC^{-1}\Big(x_i[12..15] \oplus x_{i+1}[8..11] \oplus x_{i+1}[12..15]\Big)$

Proof. Here again the idea is to exploit the interaction between the linearity of `MixColumns` and the linear operations in the key-schedule. We only prove the first equation (the proofs of the other two is quite similar). Expressing y in terms of w gives:

$$z_{i-1}[4..7] = MC^{-1}\left(w_{i-1}[4..7]\right)$$

We can relate w_{i-1} to x_i thanks to the `AddRoundKey` operation:

$$z_{i-1}[4..7] = MC^{-1}\left(k_i[4..7] \oplus x_i[4..7]\right)$$

And there, we can exploit the linearity of the key-schedule:

$$z_{i-1}[4..7] = MC^{-1}\left(k_{i+1}[0..3] \oplus k_{i+1}[4..7] \oplus x_i[4..7]\right)$$

The sub-keys can then be expressed back in terms of w and x:

$$z_{i-1}[4..7] = MC^{-1}\left(w_i[0..3] \oplus x_{i+1}[0..3] \oplus w_i[4..7] \oplus x_{i+1}[4..7] \oplus x_i[4..7]\right)$$

And then, the linearity of `MixColumns` can be exploited as well:

$$z_{i-1}[4..7] = z_i[0..3] \oplus z_i[4..7] \oplus MC^{-1}\left(x_i[4..7] \oplus x_{i+1}[0..3] \oplus x_{i+1}[4..7]\right).$$

□ □

Assume that $x_0[0]$ is known: it is possible to deduce there from the value (and the difference) in $z_0[0]$, and finally the difference in $x_1[0..3]$ (by Property 13). Because the difference in $y_1[0..3]$ can be deduced from the ciphertexts, it follows that the actual values in $x_1[0..3]$ can be deduced thanks to Property 12. This also reveals bytes 0,7,10 and 13 of z_1 (observe Figure 3.5). It follows that if $x_0[0..3]$ were known, then the key could

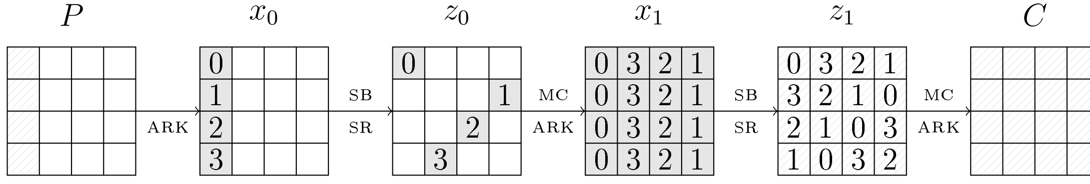


Figure 3.5: Two chosen plaintexts attack on two AES rounds. Gray bytes indicate the presence of a difference, and hatched bytes indicate the presence of a known difference. If byte i is known in x_0, then the actual values of all the bytes with the same number can be found.

easily be deduced. The attack works by constructing a set of possibles values of $x_0[0..3]$ of expected size 256 in which the actual solution is guaranteed to be found. This process has a complexity of the order of 256 encryptions, and therefore dominates the complexity of the attack. A pseudo-code of the attack is shown in Algorithm 8. The attack works in 3 stages, each one using Property 16 in a different way.

Algorithm 8: Pseudo-code of the attack on 2 rounds using 2 chosen plaintexts.

Function `2R-2CP-Attack`(P, C)
- **forall the** $x_0[2] \in \mathbb{F}_{256}$ **do** // Build T_2
  - **compute** $z_0[10]$ and $x_1[8..11]$;
  - **let** $u = x_1[8..11] \oplus C[4..7] \oplus C[8..11]$ **in**;
  - **let** $i = z_0[10] \oplus (0d, 09, 0e, 0b) \cdot u$ **in**;
  - $T_2[i] \leftarrow T_2[i] \cup \{x_0[2]\}$;
- **end**
- **forall the** $x_0[3] \in \mathbb{F}_{256}$ **do** // Build T_3
  - **compute** $z_0[7]$ and $x_1[4..7]$;
  - **let** $u = x_1[4..7] \oplus C[0..3] \oplus C[4..7]$ **in**;
  - **let** $i = z_0[7] \oplus (0b, 0d, 09, 0e) \cdot u$ **in**;
  - $T_3[i] \leftarrow T_3[i] \cup \{x_0[3]\}$;
- **end**
- **forall the** $x_0[1] \in \mathbb{F}_{256}$ **do** // Retrieve the key
  - **Compute** $z_0[13], x_1[12..15], z_1[3], z_1[6], z_1[9], z_1[12]$;
  - **Compute** $z_1[13]$; // Using Property 16
  - **Compute** $x_1[1]$, the difference in $z_0[1]$, and $x_0[0]$;
  - **Compute** $z_0[0], x_1[0..3], z_1[0], z_1[7]$ and $z_1[10]$;
  - **Read** possible value(s) of $x_0[2]$ in $T_2\big[z_1[6] \oplus z_1[10]\big]$;
  - **Read** possible value(s) of $x_0[3]$ in $T_3\big[z_1[3] \oplus z_1[7]\big]$;
  - **Compute** k_2 and check for correctness;
- **end**

end

1. We first show that once $x_0[1]$ is known, then $x_0[0]$ can be determined using Prop-

erty 16, item *iii*). The equation is:

$$z_0[12..15] \oplus z_1[8..11] \oplus z_1[12..15] = MC^{-1}\Big(x_1[12..15] \oplus C[8..11] \oplus C[12..15]\Big),$$

We enumerate the possible values of $x_0[1]$ and compute all the bytes marked "1" in Figure 3.5. At this stage, the right-hand side the equation is fully known. In the left-hand side, $z_0[13]$ and $z_1[9]$ are known, and therefore $z_1[13]$ can be deduced by projecting the (vector) equation on the second component. The actual values and the differences can then be deduced in $x_1[1]$, which reveals the difference in $z_0[0]$ (by Property 13). The actual values in $x_0[0]$ can then be deduced by Property 12. We expect on average one possible value of $x_0[0]$ per value of $x_0[1]$.

2. We then seek to extend this procedure to $x_0[2]$ and $x_0[3]$. To this end, we still use Property 16, equation *ii*):

$$z_1[4..7] \oplus z_1[8..11] = z_0[8..11] \oplus MC^{-1}\Big(x_1[8..11] \oplus C[4..7] \oplus C[8..11]\Big), \qquad (\clubsuit)$$

The third coordinate of the right-hand side can be entirely deduced from $x_0[2]$. We can therefore build a table yielding $x_0[2]$ from the third coordinate of the right-hand side of ($\clubsuit$), as shown in the first loop in Algorithm 8.

We perform the same operations with $x_0[3]$, using Property 16, equation *i*):

$$z_1[0..3] \oplus z_1[4..7] = z_0[4..7] \oplus MC^{-1}\Big(x_1[4..7] \oplus C[0..3] \oplus C[4..7]\Big), \qquad (\heartsuit)$$

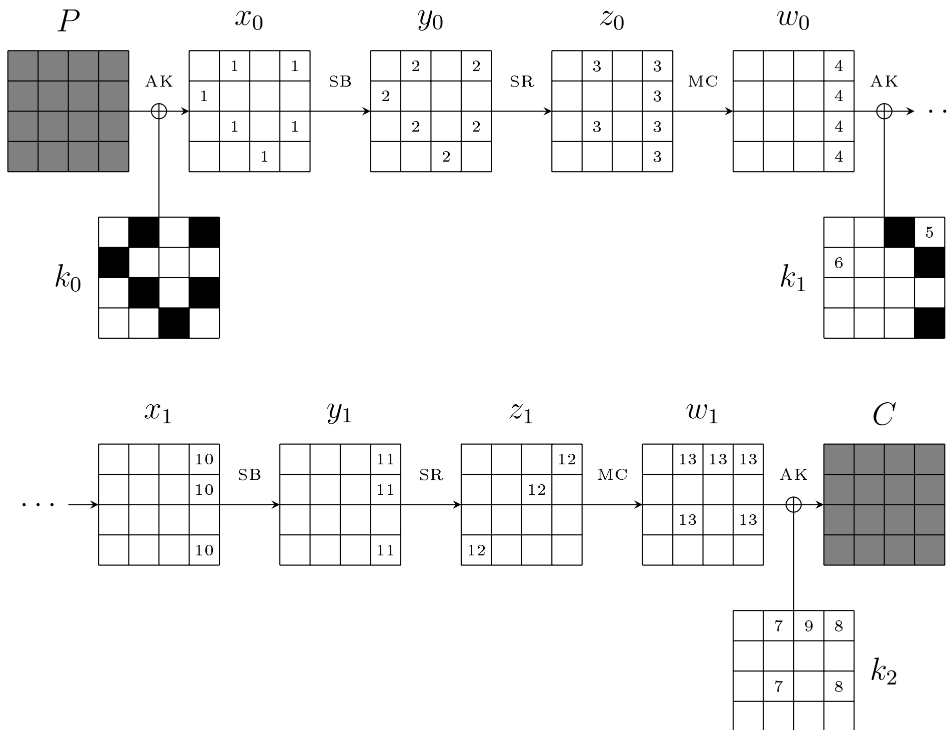
Here, the fourth coordinate of the right-hand side can be entirely deduced from $x_0[3]$. We therefore build a table yielding $x_0[3]$ from the third coordinate of the right-hand side of ($\heartsuit$) (as shown in the second loop in Algorithm 8).

3. Once the two tables T_2 and T_3 have been built, we are ready to derive $x_0[2]$ and $x_0[3]$. For this purpose, we enumerate the values of $x_0[1]$, derive $x_0[0]$ as explained above. The third component of equation ($\clubsuit$) and the fourth component of ($\heartsuit$) can be computed, and thanks to T_2 and T_3 the corresponding values of $x_0[2]$ and $x_0[3]$ can be retrieved in constant time, resulting in an average of 256 suggestion for the first column of x_0. From there, k_2 can be deduced, and the key-schedule can be inverted to retrieve k_0.

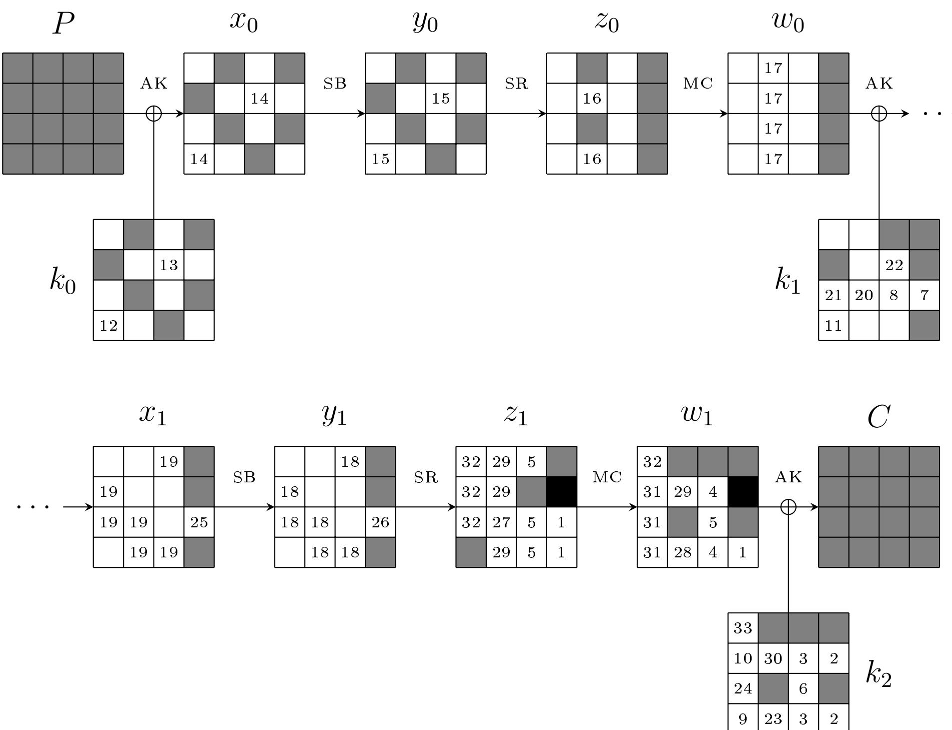
3.5.4 One Known Plaintext

Both Dunkelman and Keller, and our tool of independently found a 1 known-plaintext attack against two full rounds. One possible version of the attack, depicted in Figure 3.6, is based mainly on Property 14 (the "jumps" in the key-schedule) and on many simpler key schedule considerations.

In the first phase of the attack, the adversary guesses nine subkey bytes (marked in black in the upper part of the figure). Step 7 uses Property 14(3), step 8 uses Property 14(2), steps 5, 6 and 9 use the key schedule, and the remaining steps are computed using the AES algorithm.



(a) First half of the attack: the value of the bytes marked in black is guessed. The bytes marked by 7 and 8 are found using Property 14.



(b) The bytes marked is black are guessed. The bytes marked by 9 and 13 are found using Property 14(1)

Figure 3.6: The attack with one known plaintext on two round AES.

In the second phase of the attack, the adversary guesses one state byte (marked in black in the lower part of the figure). Steps 9 and 13 are based on Property 14(1), steps 1, 5, 29 and 32 use Property 13, steps 3, 7, 8, 10-12 and 21-24 use the key schedule, and the remaining steps are performed by applying AES' operations on known values.

The time complexity of the attack is determined by the amount of bytes which are guessed. Namely, as the adversary guesses 10 bytes, the time complexity of the attack is 2^{80} encryptions. Because the automated tool is quite flexible, we could check without any effort that the SQUARE block cipher was a bit less strong than its successor (the AES): in the same setup, we found an attack with 9 guessed bytes, *i.e.*, a time complexity of 2^{72} encryptions.

A Time-Memory Trade-Off. The time complexity can be reduced at the expense of enlarging the memory complexity, using non-linear equations and a precomputed table as in Property 15. This improved attack was also found by our tool. In order to achieve this reduction, the adversary performs the following precomputation: Let bytes 4 and 14 of k_0 be denoted by b and c. It is possible to represent all the bytes found during the attack procedures in terms of b, c, the plaintext, the ciphertext, and the other 8 key bytes which are guessed in the original attack procedure. At the end of the deduction procedure, after a suggestion for the full subkey k_2 (in terms of b and c) is obtained, the adversary decrypts the ciphertext through the last round and obtains a suggestion for bytes 4 and 5 of k_1. These bytes can be used as a consistency check, as they can be retrieved independently by the key schedule algorithm, using the suggestion of k_2. This consistency check supplies two non-linear equations in b and c, and it turns out that the equations are of the following form:

$$\begin{aligned} a_5 &= f_0(b, c, a_0, a_1, a_2, a_3, a_4) \\ a_7 &= f_1(b, c, a_0, a_1, a_2, a_3, a_6) \end{aligned} \tag{3.7}$$

where f_0 and f_1 are fixed known functions, and $a_0, \ldots, a_7$ are one-byte parameters depending on the plaintext, the ciphertext, and the eight additional subkey bytes guessed in the original attack. Since the values of $a_0, \ldots, a_7$ are very cumbersome, we do not present them in this thesis.

Hence, it is possible to compute in advance the values of (b, c) corresponding to each value of $(a_0, \ldots, a_7)$, and store them in a table. In the online phase of the attack, the adversary guesses only 8 subkey bytes (instead of 10), computes the values of $(a_0, \ldots, a_7)$, and uses the table in order to retrieve b and c. The rest of the attack is similar to the original attack.

The time complexity of the resulting attack is reduced to 2^{64}, but on the other hand, the attack requires 2^{64} 16-bit blocks of memory.

The memory requirement can be further reduced to 2^{48} by observing that the knowledge of a_1, a_3 and the six subkey bytes $k_0[6]$, $k_0[11]$, $k_0[12]$, $k_1[8]$, $k_1[13]$, $k_1[15]$ allows one to deduce the value of the two remaining subkey bytes guessed in the modified attack. Using this observation, the attack procedure can be slightly changed as follows: The adversary starts with guessing the values of a_1 and a_3, and prepares the table for the given value of a_1, a_3. In the online phase of the attack, the adversary guesses the six subkey bytes $k_0[6]$, $k_0[11]$, $k_0[12]$, $k_1[8]$, $k_1[13]$, $k_1[15]$, deduces the value of the two

additional required subkey bytes, and performs the original attack. This change reduces the memory complexity to 2^{49} bytes (since the table is constructed according to 6 byte parameters instead of 8), while the time complexity remains unchanged at 2^{64}.

3.5.5 Improved Attack When the Second MixColumns is Omitted

In Section 3.9, we present a differential attack on 6-round AES which uses as a subroutine a 2-round attack on AES. In the attack scenario, the two rounds attacked in the subroutine are the last two rounds of AES, *i.e.*, a full round and a round without the MixColumns operation. In this section we present an improved variant of the attack with two known plaintexts presented above that applies in this scenario. We note that this attack gives another evidence to the claim made in [DK10b] that the omission of the the last MixColumns operation in AES reduces the security of the cipher.

The attack, presented in Figure 3.7, consists of two phases. In the first phase, the first 13 steps are identical to the first 13 steps of the attack on two full rounds presented in Section 3.5.1 above. Steps 14-20 and 25 exploit the key schedule, and the rest of the steps apply AES' operations to known values.

The second phase uses Property 9.3 and simpler key schedule observations. Step 1 uses Property 14(1,2), step 20 uses Property 14(1), step 9 uses Property 13, steps 2-4, 11, 12, 16-19 and 25 use the AES key schedule, and the remaining steps are performed using partial encryption or decryption.

3.6 Attacks on Three-Round AES

In this section we consider attacks on three rounds of AES, denoted by rounds 1-3. First we present a simple attack with two *chosen* plaintexts, then we present a bit more complex meet-in-the-middle attack with 9 known plaintexts, and finally we present a very time-consuming attack with a single known plaintext.

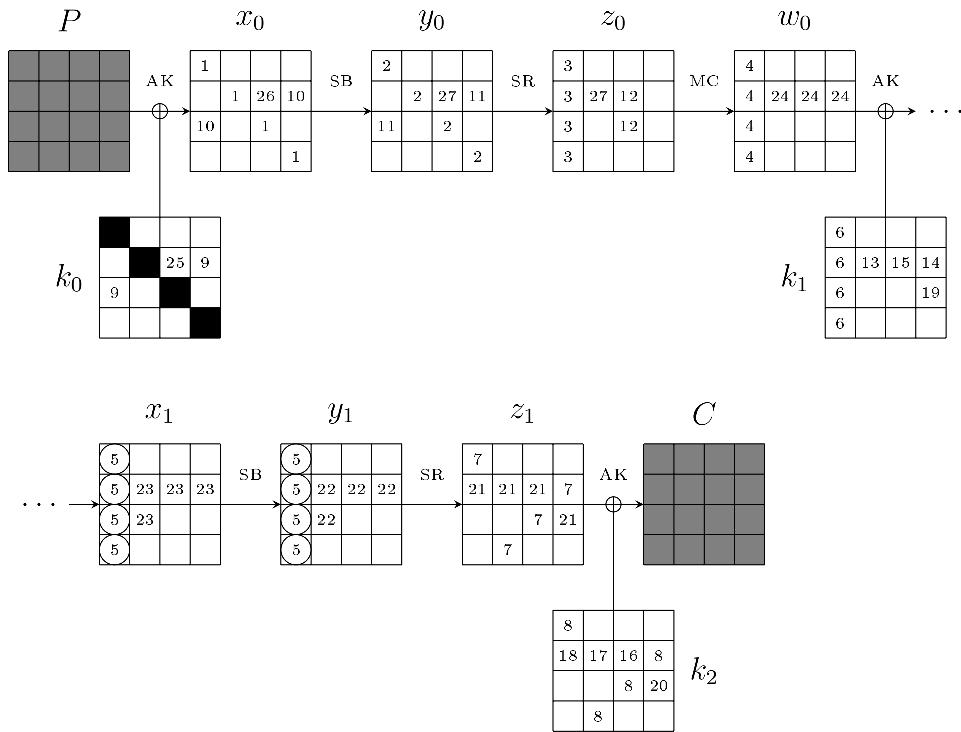
3.6.1 Two Chosen Plaintexts

The 2 rounds/2-chosen plaintext attack of section 3.5.3 can easily be leveraged into a 3-round attack of complexity 2^{16}, thus improving on a manually-found attack with complexity 2^{32} described in [BDD+12].

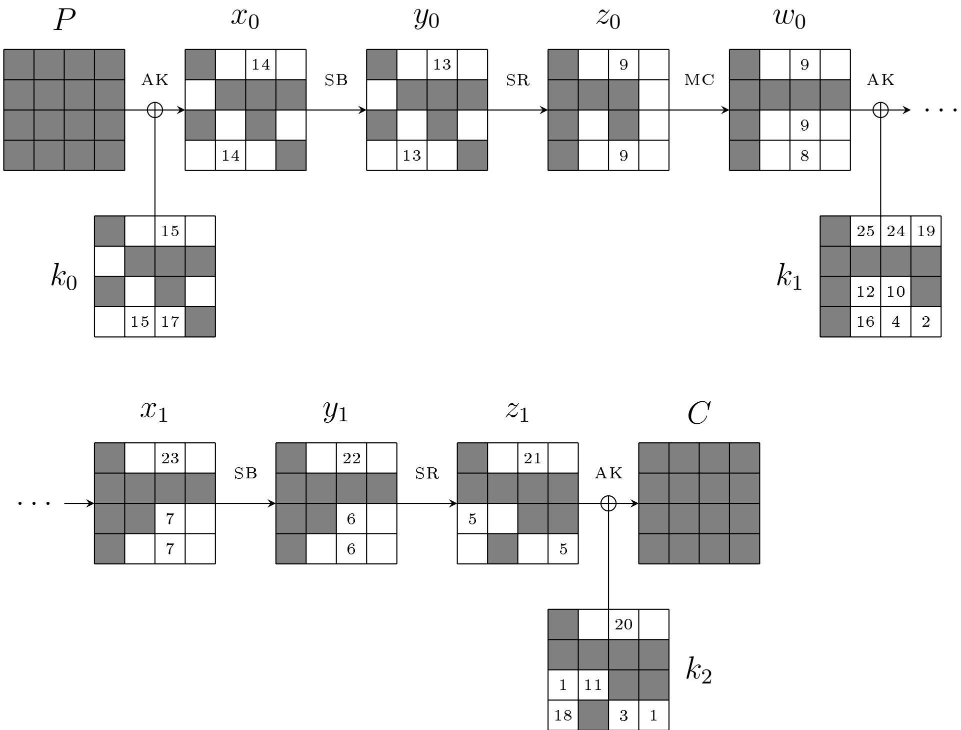
In this improved attack, the adversary asks for the encryption of two plaintexts which differ only in the first byte. By guessing $k_0[0]$, the adversary obtains the differences in $x_1[0..3]$. This is sufficient to apply the attack of section 3.5.3 to rounds 2 and 3. The complexity of the process is therefore 2^{16} encryptions.

3.6.2 Nine Known Plaintexts

An attack with 9 known plaintexts has been found manually by Dunkelman and Keller. It uses a combination between the differential approach and the standard meet-in-the-middle approach. The adversary guesses subkey material in k_0 and k_3, and obtains a



(a) First half of the attack.



(b) Second half.

Figure 3.7: The attack on two rounds of AES without the second `MixColumns` using two known plaintexts. Bytes marked in black are guessed, and bytes marked in gray are known at this phase of the attack.

consistency check on the intermediate difference after the `ShiftRows` operation of round 2.

Concretely, denote the intermediate values in byte 0 after the ShiftRows operation of round 2 by $X_1, X_2, \ldots, X_9$. In the first phase of the attack, the adversary guesses bytes 0, 7, 10 and 13 of the equivalent subkey u_3 and partially decrypts the ciphertexts through the last round (obtaining the actual values in $x_3[0..3]$). Then, using the linearity of the MixColumns operation, the adversary computes the differences $X_1 + X_2, \ldots, X_1 + X_9$, and stores their concatenation (a 64-bit vector) in a hash table. In the second phase of the attack, the adversary guesses bytes 0, 5, 10 and 15 of k_0 and byte $k_1[0]$ and by partial encryption of the plaintexts, obtains the values of $X_1 + X_2, \ldots, X_1 + X_9$, and checks whether their concatenation appears in the hash table. This consistency check is a 64-bit filtering, and thus only $2^{72} \times 2^{-64} = 2^8$ key suggestions are expected to remain. By repeating the procedure with the three other columns, the adversary obtains about 2^{32} suggestions for the full subkey k_0 (along with many other subkey bytes), which can be checked by exhaustive key search. The time complexity of the attack is about 2^{40} encryptions, and the memory requirement is 2^{35} bytes of memory.

3.6.3 One Known Plaintext

The tool found a guess-and-determine attack with a single known plaintext, depicted in Figure 3.8. The attack consists of two phases.

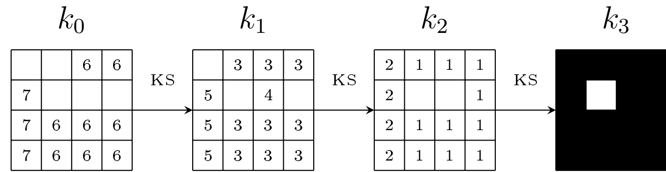
In the first phase (shown in the top part of the figure) the adversary guesses 15 subkey bytes, and uses key schedule considerations deduce numerous additional subkey bytes in the four subkeys k_0, k_1, k_2 and k_3. Step 4 of the deduction uses Property 14(1), and the other steps use the key schedule algorithm directly.

The second phase (shown in the bottom part of the figure) is the meet-of-the-middle part of the attack. Using the known subkey bytes, the adversary partially encrypts the plaintext and decrypts the ciphertext and obtains sufficient information in order to apply Property 13 to the `MixColumns` operations of rounds 2 and 3. Steps 13 and 17 of this part use Property 13, and the other steps use AES' operations and the knowledge obtained in previous steps.

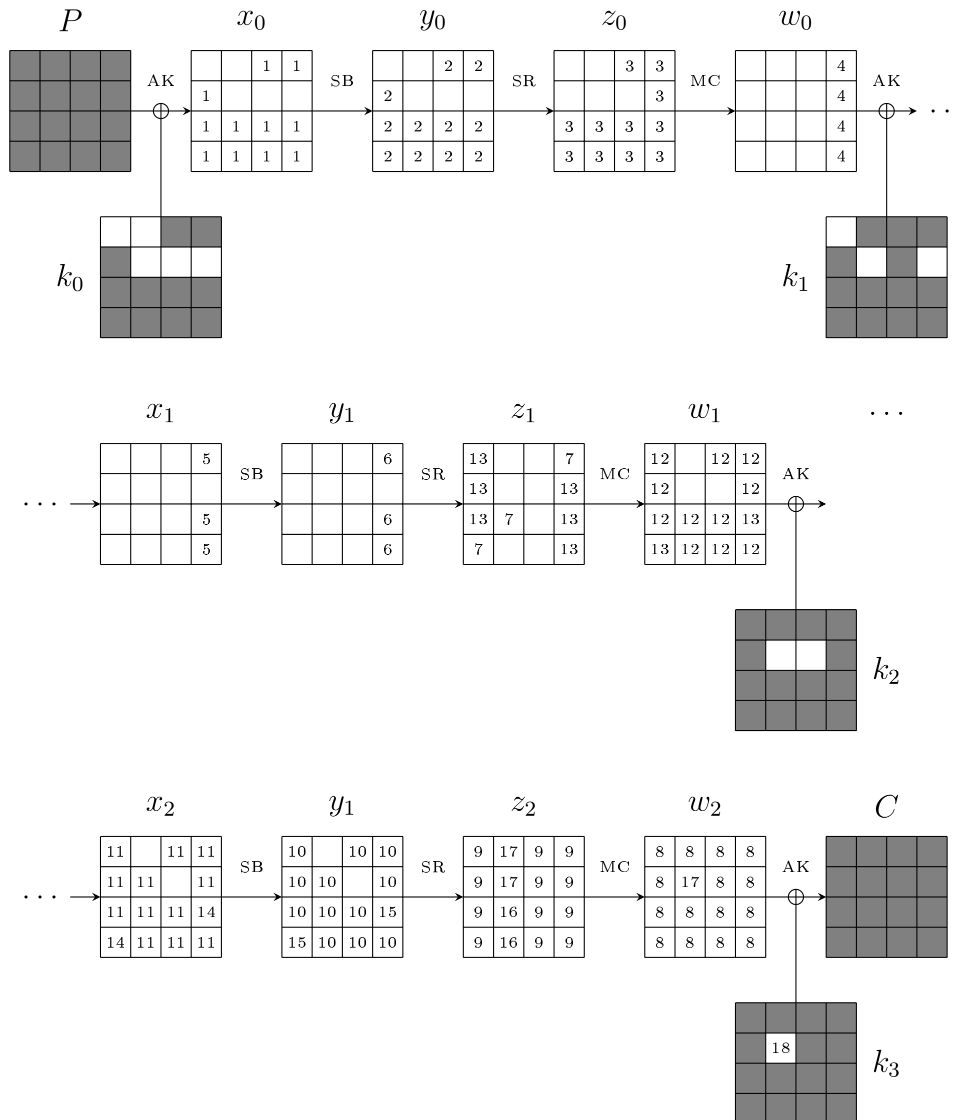
Since the adversary guesses 15 key bytes, the time complexity of the attack is 2^{120} encryptions. As in the single-plaintext attacks on one-round and two-round AES, the adversary can reduce the time complexity at the expense of enlarging the memory requirement, using non-linear equations and a precomputed table. The time complexity of the resulting attack is 2^{96} encryptions, and the memory requirement is 2^{72} bytes. Since the technique is similar to the improvement of the 2-round attack presented in 3.5.4, and the obtained equations are quite cumbersome, we do not present the improvement here.

3.7 Attacks on Four-Round AES

We now consider attacks on 4-round AES and turn our attention to *chosen-plaintexts* attacks. The well-known "square" attack on 4 rounds requires 256 chosen plaintexts and the equivalent of 2^{14} encryptions. Manually-found attacks with 10,5 or 2 chosen plaintexts



(a) he first half of the attack only uses the key-schedule. All bytes but one are guessed in k_3 (marked in black), and the diagram shows in which order other subkey bytes can be deduced using the key relations



(b) Second half of the attack: deduction of the remaining subkey bytes. Known bytes are marked in gray.

Figure 3.8: The attack on three rounds of AES using one known plaintext.

with respective time complexities 2^{40}, 2^{64} and 2^{104} are described in [BDD+12]. The tool described Chapter 2 automatically found a practical attack using four plaintext differing only in one byte, of complexity about 2^{32}.

We note that these attacks can be transformed into known plaintext attacks using the standard birthday-based transformations, but these usually result in a high data complexity.

3.7.1 Ten Chosen Plaintexts

The attack with 10 chosen plaintexts is similar to the 3-round attack with 9 known plaintexts presented 3.6.2. The adversary asks for the encryption of ten plaintexts which differ only in bytes 0,5,10 and 15. Then she guesses subkey material in the subkeys k_0, k_1 and the equivalent subkeys u_3 and u_4, and obtains a consistency check on some intermediate difference after the MixColumns operation of round 2.

Let us denote the intermediate values in byte 0 after the MixColumns operation of round 2 by $X_1, X_2, \ldots, X_{10}$. In the first phase of the attack, the adversary guesses bytes 0, 7,10 and 13 of the equivalent subkey u_4 and byte 0 of the equivalent subkey u_3 and partially decrypts the ciphertexts through the last two rounds obtaining the actual values in the byte $x_3[0]$. (Note that reversing the order of the MixColumns and AddRoundKey operations in the two last rounds allows her to obtain this intermediate value by guessing only 40 subkey bits). Then, using the linearity of the AddRoundKey operation, the adversary computes the differences $X_1 + X_2, \ldots, X_1 + X_{10}$, and stores their concatenation (a 72-bit vector) in a hash table.

In the second phase of the attack, the adversary guesses bytes 0, 5, 10 and 15 of k_0 and the byte $k_1[0]$. By the structure of the chosen plaintexts, this allows her to compute the differences between pairs of intermediate values $w_2[0..3]$ (since the actual values in byte 0 before the MixColumns operation of round 2 are known by partial encryption, and the difference in bytes 1, 2, 3 is zero). Thus, the adversary obtains the values of $X_1 + X_2, \ldots, X_1 + X_{10}$, and checks whether their concatenation appears in the hash table. This consistency check is a 72-bit filtering, and thus only $2^{80} \times 2^{-72} = 2^8$ key suggestions are expected to remain. By repeating the procedure with the three other columns (from the ciphertext side), the adversary obtains about 2^{32} suggestions for the full equivalent subkey u_4 (along with many other subkey bytes), which can be checked by exhaustive key search. The time complexity of the attack is about 2^{40} encryptions, and the memory requirement is about 2^{43} bytes of memory.

3.7.2 Five Chosen Plaintexts

If only five chosen plaintexts are available to the adversary, she can perform a variant of the attack described above, at the expense of enlarging the time and memory complexities. The plaintexts are chosen as before, but more key material is guessed: from the ciphertext side, the adversary guesses bytes 0, 7, 10 and 13 of u_4 and bytes 0, 1, 2 and 3 of u_3, and from the plaintext side, the adversary guesses bytes 0, 5, 10 and 15 of k_0 and bytes 0, 1, 2 and 3 of k_1. This allows her to get a consistency check on the intermediate difference at the end of round 2 in bytes 0, 5, 10 and 15 (instead of only byte 0), and thus, the four

pairs which can be extracted from the data supply a 128-bit filtering and only the correct key suggestion is expected to remain. Finally, the adversary repeats the attack procedure with three other columns from the ciphertext side, and obtains a single suggestion (or a few suggestions) for the full equivalent subkey u_4. The time complexity of the attack is about 2^{64} encryptions, and the memory requirement is about 2^{68} bytes.

3.7.3 Four Chosen Plaintexts.

The four plaintext only differ in byte 0 of the plaintext (but they *must* be pairwise different). We use the notation $x_i^{(j)}$ to denote the j-th message.

In a first phase, we construct 16 hash tables $\mathcal{T}_0, \dots, \mathcal{T}_{15}$, which are subsequently used in the remaining steps of the attack. The table $\mathcal{T}_\ell$ is constructed according to the following steps:

1. First, enumerate all the possible values of $x_0^{(0)}[0]$. Because the differences in x_0 are known, then $x_0^{(i)}[0]$ can be deduced for $i = 1, 2, 3$. This in turn allows to determine the differences in $y_0[0]$, and also in $x_1[0..3]$.
2. Define $c_2 = \lfloor \ell/4 \rfloor$ and $r_1 = \sigma(c_2)$, where σ denotes the permutation (0321).
3. Next, enumerate $x_1^{(0)}[r_1]$. Because the differences in this byte are known, then the values in $x_1^{(i)}[r_1]$ can be deduced for $i = 1, 2, 3$. This allows to find the differences in $y_1[r_1]$, and then in $x_2[4c_2..4c_2 + 3]$.
4. Finally, enumerate the values of $x_2^{(0)}[\ell]$. Again, recover $x_2^{(i)}[\ell]$ for $i = 1, 2, 3$, and thus recover the differences in $y_2[\ell]$.
5. Store the association
$$\left(y_2^{(0)}[\ell] \oplus y_2^{(1)}[\ell], y_2^{(0)}[\ell] \oplus y_2^{(2)}[\ell], y_2^{(0)}[\ell] \oplus y_2^{(3)}[\ell]\right) \mapsto \left(x_0^{(0)}[0], x_1^{(0)}[r_1]\right)$$
in the hash table $\mathcal{T}_\ell$.

The hash tables are now used in the following way: enumerate the values of $x_3^{(0)}[0..3]$, compute the differences in byte 0, 5, 10 and 15 of y_2, and use the differences to look-up in $\mathcal{T}_0, \mathcal{T}_5, \mathcal{T}_{10}$ and $\mathcal{T}_{15}$. Only keep values of $x_3[0..3]$ that suggest the same value of $x_0^{(0)}[0]$ (there should be about 2^8 of them). We implemented the attack, and we could indeed verify in practice that this procedure isolates a set of about $2^{8.5}$ candidates for the first column of x_3. It can then be repeated for the other three columns, and we are left with about $2^{34.5}$ candidates for the full x_3, each one of which suggest a full key (partial encryption reveals w_3, which in turns reveal k_4 and the key-schedule can be inverted back to k_0).

This could be refined a little bit by only considering the quadruplets of columns that suggest the same values of $x_1[0..3]^{(0)}$ (and there should very likely be very few of them). This would avoid testing 2^{32} keys.

3.8 Attack on Five-Round AES

In this section we present two new attacks on five rounds of AES. The first one is an attack using only one known plaintext. Its approximate time complexity is 2^{120} with a

memory requirement around 2^{96} bytes so it is unclear that this “attack” can be implemented faster than exhaustive search. However, it shows a weakness of the AES and, in particular, a weakness in the key-schedule. The second one is an attack requiring 8 chosen plaintexts with a time and memory complexities around 2^{64} and 2^{56} respectively.

3.8.1 One Known Plaintext.

The attack is a direct meet-in-the-middle between two sets of bytes. Knowing the values of all the bytes in both sets allows to retrieve the master key k_0 instantly.

The first set of bytes is represented in white on figure 3.9. It is obtained from 15 well chosen key-bytes (indexed by 1 and circled in the figure). Thanks to key-schedule equations, it is possible to deduce all the white key bytes from the 15 first bytes. The remaining white bytes are found by partial encryption/decryption.

The second set of bytes is pretty small and is represented in black. Like white bytes, it is obtained from bytes indexed by 1 (and circled) which allow to deduce some other key-bytes in a simple way.

Even though black and white bytes can take $2^{8(15+12)}$ values, this can efficiently be narrowed down to 2^{120} by the technique presented Chapter 2. To apply it, we need 12 *independent* and separable equations between some of these bytes. To obtain these equations, we look for bytes which are linear combinations of white and black bytes (assumed known). They are colored in gray.

1. Bytes indexed by A are obtained from k_5 by using key-schedule equations and this leads to three equations because $k_0[0]$, $k_0[11]$ and $k_2[12]$ are constrained. Furthermore, they are linear combinations of white and black bytes because the last column of each subkey is known and only those bytes go through the Sbox.
2. Bytes indexed by B are obtained from partial encryption/decryption (only the AddRoundKey operation).
3. Bytes indexed by C are obtained by application the Property 13. This leads to nine equations since in three cases 5 bytes are known and in two cases 6 bytes are known.

3.9 Attack on Six-Round AES

The design of AES follows the *wide trail* design strategy, which assures that the probability of differentials is extremely low, even for only four rounds of the cipher. For example, it was proved in [PSC+02, PSLL03], that any 4-round differential of AES has probability of at most 2^{-110}. Hence, it is widely believed that no regular differential attack can be mounted on more than 5 rounds of AES. Furthermore, the best currently known differential attack on AES-128 is on only four rounds, and all known attacks on 5 and more rounds use "more sophisticated" techniques like impossible differentials, boomerangs, or Squares.

In this section we show that the low data complexity attack on 2-round AES presented in 3.5.5 can be leveraged to a differential attack on 6-round AES. Although the data complexity of the resulting attack is high, the data complexity of its known plaintext

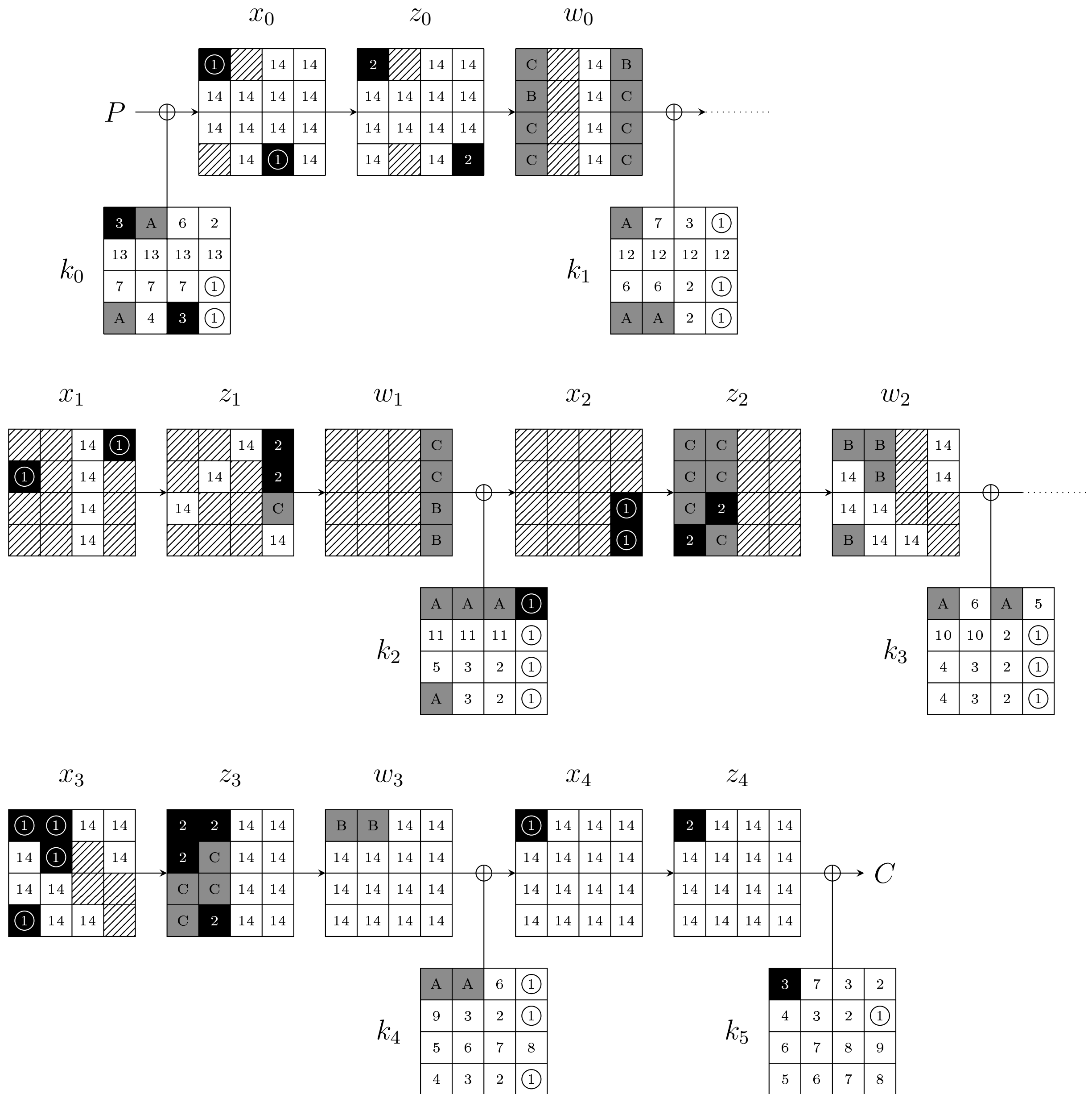


Figure 3.9: One known plaintext attack on 4.5 AES rounds. Black bytes are enumerated and stored in a hash table. White bytes are enumerated. Gray bytes are linear combinations of white and black bytes. Hatched bytes play no role. The number indicates the step of the attack in which the value of each byte is discovered.

variant is still smaller than the data complexity of the best known attack on 6-round AES in the known plaintext model. While our attack certainly does not threaten the security of AES, it shows that its security with respect to conventional differential attacks is lower than expected before.

As in most published attacks on reduced-round variants of AES, we assume that the `MixColumns` operation in the last round is omitted, like in the full AES. We were not able to extend the attack to the case where the last `MixColumns` operation is not omitted. This gives another evidence to the claim made in [DK10b] that the omission of the last round `MixColumns` affects the security of AES.

Our 6-round attack is based on the following 3-round truncated differential: The input difference in all bytes except for byte 0 is zero, and the output difference in all bytes except for bytes 0, 5, 10 and 15 is zero. We depict the differential in Figure 3.10. A pair satisfying the input and output requirements of the differential in rounds 2-4 is called a right pair.

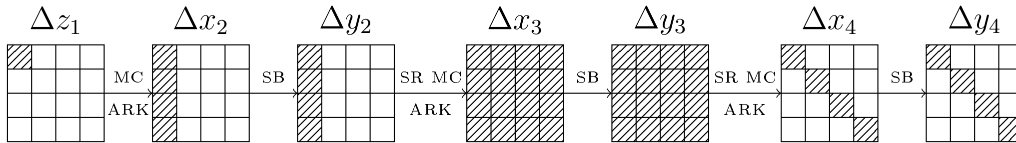


Figure 3.10: The 3-Round Truncated Differential Used in the 6-round Differential Attack.

Consider a right pair (P, P'). By the structure of AES, the intermediate difference at the input of round 3 is zero in all bytes except for 0, 1, 2 and 3. Thus, there are at most 2^{31} possible differences in the input of the `SubBytes` operation of round 4. On the other hand, since the difference at the output of round 4 is zero in all bytes except for 0, 5, 10 and 15, there are only 2^{32} possible differences after the `SubBytes` operation of round 4. Note that by Property 9.1, the input and output differences of a `SubBytes` operation yield a single suggestion (on average) for the actual values. Therefore, if (P, P') is a right pair, then there are only 2^{64} possibilities of the corresponding actual values after the `SubBytes` of round 4 (or equivalently, for the actual values after the `MixColumns` operation of round 4).

This observation allows us to mount the following known plaintext attack:

1. Ask for the encryption of $2^{108.5}$ plaintexts P_i under the unknown key, and denote the corresponding ciphertexts by C_i.
2. Insert (P_i, C_i) into a hash table indexed according to bytes 1-4, 6-9, 11-14 of P_i and bytes 1-6, 8, 9, 11, 12, 14, 15 of C_i, and consider only the colliding pairs in the hash table (which are the only pairs which may be right). The number of remaining pairs is $2^{216} \times 2^{-192} = 2^{24}$.
3. For each of the remaining pairs, assume that it is a right pair, and for each of the 2^{64} possible actual values after the MixColumns operation of round 4, apply the attack presented in 3.5.5 on rounds 5-6. Note that the 2-round attack requires that the difference in the four bytes $x_2[0..3]$ in the attacked variant is non-zero, and this condition is indeed satisfied in our attack (for the state x_6 which corresponds to x_2 in the two-round attack).

Since the time complexity of the 2-round attack presented in 3.5.5 is 2^{24} encryptions, the overall complexity of the attack is $2^{24} \times 2^{64} \times 2^{24} = 2^{112}$ encryptions. The data complexity of the attack is $2^{108.5}$ known plaintexts, which is smaller than the data complexities of the previously known attacks in the known plaintext model (see, e.g. [CKK+01]).

3.10 Implementations

We have implemented and verified attacks (or parts thereof) in practice. This brief section mentions some of the techniques we used and the result we obtained.

Several attacks are meet-in-the-middle that require hash tables containing 2^{32} entries (only in the case of described attacks), each entry being 2 or 4-byte long. The main difficulty in implementing these attacks was memory management (how to represent and store the tables). Careful and "low-level" memory management, *e.g.*, using `mmap`, was necessary for the attack to be somewhat practical. The standard techniques for hash tables (storing buckets as linked lists) incurs an important space overhead in our case, because the pointers are 64-bit wide, and are impractical.

We also observed that the distribution of the number of entries in each bucket roughly follows a Poisson law of expectation 1, so that the maximum number of entries in a bucket can be represented by an 8-bit number. We thus use three arrays to store the hash table:

- An array A_c stores the size of each bucket in 8-bit entries (size = 4Gbyte)
- An array A_h stores the content of all the buckets (size=16Gbyte)
- An array A_i stores the location of each bucket in the previous array (size=16Gbyte)

The last array is useful to access the hash table in $\mathcal{O}(1)$ time, but it needs not be stored, which means that such a hash table can be stored in a 20Gbyte file. We then used a two-pass approach: first count the number of entries with the same key in the table and update A_c. Then computes the entries in A_i. Lastly, perform a second pass and stores the actual data in A_h. This way, the peak memory consumption is 36Gbyte.

2 AES Rounds / 2 Known Plaintext.

The meet-in-the-middle part attack has been implemented manually in C. Using the above techniques, it uses 52Gbyte of RAM, and isolates a set of about 2^{32} candidates for the first and last column of x_1 in about two hours. We checked that the set of candidates actually contains the correct solution, and that the number of candidates was consistent with our estimates.

2 AES Rounds / 2 Chosen Plaintext.

The automated tools generated an implementation of this attack, which allowed us to test it. The automatically-generated C file is 110Kbyte long. On average, there are $2^{8.65}$ candidates for $x_0[0..3]$, which is very close to our hypothesis.

4 AES Rounds / 4 Chosen Plaintext.

We implemented the meet-in-the-middle part of the attack manually in C++. Our implementation uses the above techniques for representing the hash tables, and each one of the 16 tables requires 112Mbyte. The attack therefore runs on a laptop and uses less than 1.8Gbyte of RAM. The total running time of the meet-in-the-middle phase is about 2 hours on a single core (the code is easily parallelized is easy using OpenMP, and actually runs in 14 minutes using eight Xeon E5520 cores at 2.27Ghz).

Comparison with optimal attacks.

As mentioned earlier, attacks presented in this article have been modified in order to make them more understandable. But these changes have made them, in practice, less efficient than original attacks found by the tool. Even unoptimized C codes automatically generated by the tool are faster than manual implementations of described attacks. This is mainly due to two reasons. The first one is the memory requirement: each one of these attacks has an optimal version with an approximate memory complexity of 2^{24} so we can use a simple structure to handle hash tables. Furthermore, optimal attacks use less big tables than described attacks. For instance, the best attack on four rounds with four chosen plaintexts, instead of using 16 hash tables with 2^{24} entries, use only 12 lists: 3 with 2^{24} entries, 1 with 2^{16}, and 8 with 2^{8}. The second reason comes from the fact that two attacks with the same approximate time complexity may have different real time complexity. For instance, the optimal attack on four rounds with four chosen plaintexts assign each byte $2^{33.2}$ times on average when the described attack do it $2^{34.5}$ times.

Chapter 4

Low Data Complexity Attacks on AES-Derivatives

Because our tool is somewhat generic, it is not restricted to the AES as a block cipher, and we used it to find new attacks on the message authentication code Pelican-MAC [DR05c], and to the stream cipher LEX [Bir08a]. The tool found the fastest known attacks on these two constructions, again a gratifying result. This demonstrates in a concrete way that low-data complexity attacks can be leveraged into actual attacks on full versions of some primitives.

4.1 A Forgery Attack Against Pelican-MAC

Pelican-MAC [DR05c] is a Message Authentication Code designed by Daemen and Rijmen in 2005. It is an instance of the more general ALRED construction by the same authors, which is reminiscent of CBC-MAC but aims at greater speed [DR05a]. MACs derived from the ALRED construction enjoy some level of provable security: it is shown that the MAC cannot be broken with less than $2^{n/2}$ queries (*i.e.*, without finding internal state collisions) unless the adversary also breaks the full AES itself. Pelican-MAC works as follows:

1. The internal state (an AES state) is initialized to $x_0 = \mathsf{AES}_K(0)$.
2. The message is split in 16-byte chunks, and each chunk is processed in two steps: it is XORed to the internal state, and 4 keyless AES rounds are applied (the `AddRoundKey` operation is skipped).
3. Finally, the full AES is applied with the key K to the internal state, which is then truncated and returned as the tag.

In this construction, recovering the internal state x_0 is sufficient to perform nearly-universal forgeries: first the adversary asks the MAC of an arbitrary message. Given her knowledge of x_0, she can compute the internal state x_{last} just before the full AES is applied and the tag T is returned. Then, given an arbitrary message M, she computes the internal state x_M after M has been fully processed. Then, she knows that $\mathsf{Pelican\text{-}MAC}_K(M \,\|\, x_M \oplus x_{last}) = T$, without querying the MAC (the extra message block sets the internal state to x_{last}, which is known to result in the tag T).

The best published attacks against Alpha-MAC (another ALRED construction) and Pelican-MAC has been recently found by Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, Xiaoyun Wang [YWJ+09] and aim at recovering the initial secret internal state. For Alpha-MAC, after having found an internal state collision (this requires 2^{65} queries), the internal state is recovered with a guess-and-determine attack that makes about 2^{64} simple operations. For Pelican-MAC, an impossible differential attack recovers the internal state with data and time complexity $2^{85.5}$.

The general idea of our attack on Pelican-MAC is to find a single collision in the internal state, found by injecting message blocks following a fixed truncated differential characteristic. Then, the state recovery problem has been encoded in equations and given to the tool of Chapter 2. It must be noted that an attack with the same global complexity has been independently found time by Dunkelman, Keller and Shamir [DKS11], using impossible differential techniques. The "state-recovery" phase presented here is faster though.

Our Attack.

We now present our attack against Pelican-MAC, with time and data complexity 2^{64}. We pick an arbitrary message block M_1 and query the MAC with 2^{64} random two-block messages $M_1 \parallel M_2$, and store the (message,tag) pair in a table. Then, we query the MAC on $(M_1 \oplus \Delta) \parallel M_2'$, where Δ is zero everywhere except on the first byte, and M_2' is random. When the tags collide, we check whether there is also a collision in the internal state by checking if:

$$\mathsf{MAC}_K(M_1 \parallel M_2 \parallel M_3) = \mathsf{MAC}_K\Big((M_1 \oplus \Delta) \parallel M_2' \parallel M_3\Big)$$

for several random message blocks M_3. If all the resulting tags collide, then we known that an internal collision occurred after the first two blocks with overwhelming probability, and we have:

$$\mathsf{AES}_4(x_0 \oplus M_1) \oplus M_2 = \mathsf{AES}_4(x_0 \oplus M_1 \oplus \Delta) \oplus M_2'$$

In other terms, the input difference Δ goes to the output difference $M_2 \oplus M_2'$ though 4 keyless AES rounds. The most likely differential characteristic is the one shown in Figure 4.1, even though there could be accidental difference cancellations with small probability.

We then write down the state-recovery problem as a system of equations: two unknown states with a known one-byte difference yields two unknown states with a known (full) difference. The tool described in the previous chapter quickly found[1] an attack that runs in time and space about 2^{32}, and which is summarized by Figure 4.1. Property 13 tells

1. it also found an attack with a smaller memory consumption 2^{24}, but the improved attack is much more complicated to describe

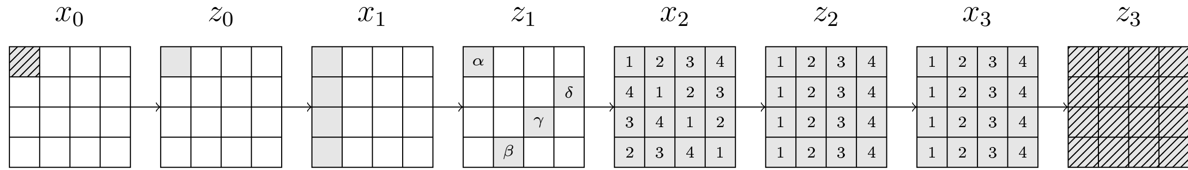


Figure 4.1: Differential path used in the attack against Pelican-MAC. Gray squares denote the presence of a difference. Hatched squares denote a known difference.

us that if α, β, γ and δ denote the differences in z_1, then the differences in x_2 are:

$$\begin{pmatrix} 02\alpha & \beta & \gamma & 03\delta \\ \alpha & \beta & 03\gamma & 02\delta \\ \alpha & 03\beta & 02\gamma & \delta \\ 03\alpha & 02\beta & \gamma & \delta \end{pmatrix}$$

The state-recovery proceeds as follows:

1-a. Guess the values in $x_3[0..3]$ and obtain the differences (thanks to the output difference).

1-b. Partially decrypt to get suggestions for α, β, γ and δ (using Property 13).

1-c. Store bytes 0–3 of x_3 in a hash table $\mathcal{T}_0$ indexed by $(\alpha, \beta, \gamma, \delta)$

2. Repeat the process with the second column of x_3. Store bytes 4–7 of x_3 in a table $\mathcal{T}_1$ indexed by $(\alpha, \beta, \gamma, \delta)$.
3. Repeat the process with the third and fourth column of x_3. Build tables $\mathcal{T}_2$ and $\mathcal{T}_3$
4. Enumerate $(\alpha, \beta, \gamma, \delta)$. Look-up $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2$ and $\mathcal{T}_3$ and retrieve the parts of x_3 corresponding to $(\alpha, \beta, \gamma, \delta)$, if present.
5. if $(\alpha, \beta, \gamma, \delta)$ occurs in the 4 tables, then we get a complete suggestion for x_3. Decrypt 3 rounds and recover x_0. Check if the input difference is right.

Alpha-MAC.

Obviously, we cannot overally improve on the attack of [YWJ+09], since finding the internal state collision dominates the running time of their attack. However, it is noteworthy that the tool found a state-recovery procedure that requires only 2^{32} elementary operations and lists of 2^{16} items, when the first input message difference contains only one active byte. This is much more efficient than its counterpart in [YWJ+09].

4.2 A Key-Recovery Attack Against LEX

LEX is a stream cipher presented by Biryukov as an example of the *leak extraction* methodology of stream cipher design [Bir06]. In this methodology, a block cipher is used in the OFB mode of operation, where after each *round* of the cipher, some part of the

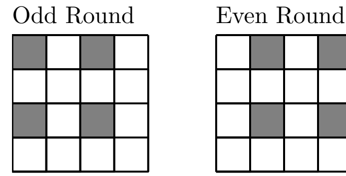


Figure 4.2: State Bytes which Compose the Output in Odd and Even Rounds of LEX. The gray bytes are the leaked bytes.

intermediate encryption value is output as part of the key stream. LEX itself uses the AES as the block cipher.

In the initialization step of LEX, the publicly known IV is encrypted by AES under the secret key K to obtain $S = AES_K(IV)$. Actually, LEX uses a tweaked version of AES where the `AddRoundKey` before the first round is omitted, and the `MixColumns` operation of the last round is present. Then, S is repeatedly encrypted in the OFB mode of operation under K, where during the execution of each encryption, 32 bits of the internal state are leaked in each round. These state bits compose the key stream of LEX. The state bytes used in the key stream are shown in Figure 4.2. After 500 encryptions, another IV is chosen, and the process is repeated. After 2^{32} different IVs, the secret key is replaced. It follows that with a given key LEX can only generate $2^{46.3}$ bytes of keystream.

4.2.1 Prior Art

LEX was submitted to the eSTREAM competition (see [Bir08b]). Due to its high speed (2.5 times faster than the AES in counter mode), fast key initialization phase (a single AES encryption), and expected security (based on the security of AES), LEX was considered a very promising candidate and selected to the third (and final) phase of evaluation. However, it was not selected to the final portfolio of eSTREAM due to an attack with data complexity of $2^{36.3}$ bytes of key stream and time complexity of 2^{112} encryptions found by Dunkelman and Keller a few weeks before the end of the eSTREAM competition [DK08]. These authors subsequently improved their own result, and the best published attack on LEX requires about 2^{40} bytes of keystream and the time equivalent of 2^{100} AES encryptions [DK10a].

Their attack is illustrated by Figure 4.3. The key idea is to find a pair of internal states, potentially obtained with different IVs, and after different numbers of encryptions, that partially collide after 4 rounds. More precisely, the objective is to find a pair of state yielding the same bytes in $x_4[4..7]$ and $x_4[12..15]$. Because this is a collision on 64 bits, the birthday paradox guarantees that 2^{32} distinct internal states are necessary. In fact, the attack is not restricted to “start” at the first round of an AES encryption cycle, but can be applied (with minor variations) to rounds $1, \ldots, 8$. Thus, only $2^{64}/8 = 2^{61}$ pairs of encryptions are necessary for the collision to occur. This number of pairs can be obtained from 2^{31} distinct encryptions, and thus from $2^{32} \cdot 10 \cdot 4 = 2^{36.3}$ keystream bytes.

One of the problems is that the collision needed for the attack cannot be fully detected just by observing the keystream: it can be detected on bytes 4,6,12 and 14, but we have

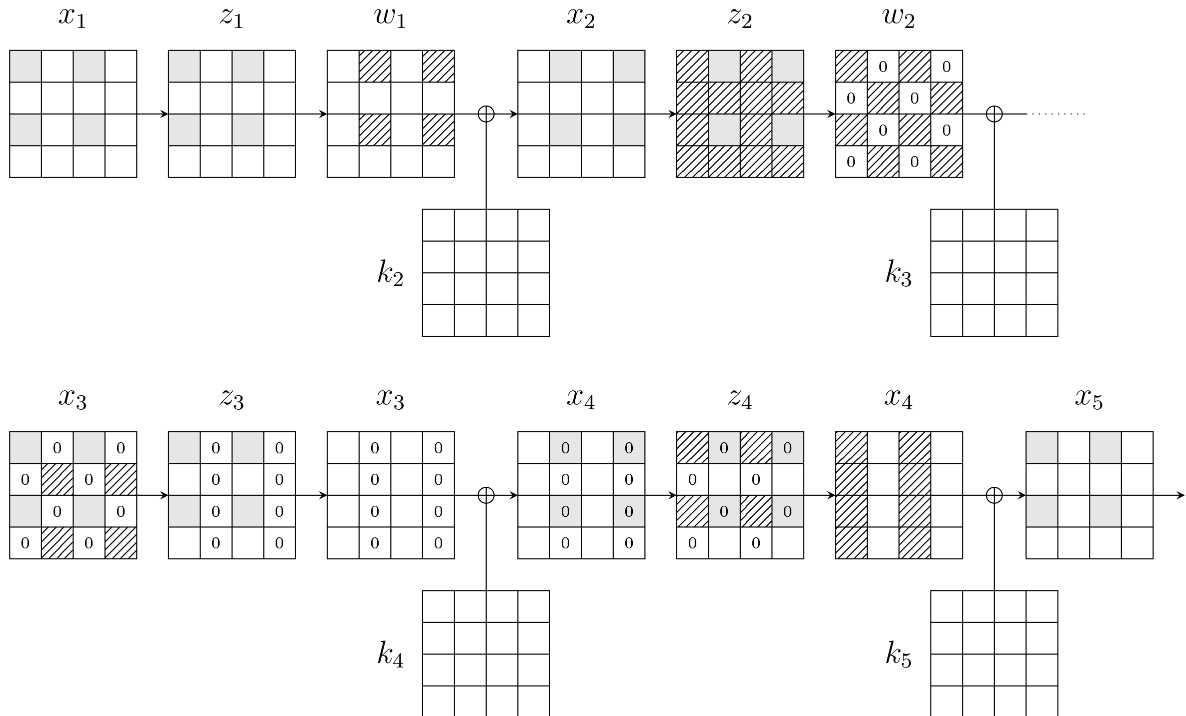


Figure 4.3: Gray squares are leaked to form the key-stream. The differences are null in squares with a 0. The differences in the hatched squares can be deduced from the leaked bytes and the existence of zero differences.

no way of detecting whether bytes 5,7,13 and 15 collide or not. The only solution is to assume that the full collision occurred and to run the next steps of the attack. In case of failure, we know *a posteriori* that the full collision did not occur. Thus, the remaining steps of the attacks have to be carried out on average 2^{32} times in order for a full collision to occur.

In the first attack of Dunkelman and Keller (given in [DK08]), the collision is exploited by a guess-and-determine attack that guesses 10 bytes. Their second attack (given in [DK10a]) uses an improved key-ranking procedure that filters the guesses and discards unlikely candidates.

Revisiting the Existing Attacks.

The key-recovery problem can be encoded as a system of equations and given to the tools. Helped by our tool we found that Dunkelman and Keller first attack was sub-optimal, as the guess-and-determine part of the attack could be dealt with in 2^{64} elementary operations (versus 2^{80} previously). This yields an attack with time complexity about 2^{96} and data complexity $2^{36.3}$, marginally improving on their second attack.

4.2.2 A New Attack

It turns out that the tool can be used to mount a different, more efficient attack. This new attack proceeds in 3 phases. The first phase is similar to the existing attacks. However, instead of looking for a *pair* of states colliding on bytes 4-7 and 12-15 in x_4, we look for *3-way collisions* on these bytes (*i.e.*, a triplet of states all having the same values in these bytes). The advantage of working with 3 messages instead of just two is that Property 12 generalizes nicely to this case: if 4 differences $\alpha, \beta, \gamma, \delta$ are randomly chosen in $\mathbb{F}_{2^8}$, then the probability that $S(x \oplus \alpha) \oplus S(x) = \gamma$ and $S(x \oplus \beta) \oplus S(x) = \delta$ is $2^{-9.5}$. Thus, in most cases, no single value of x satisfies these constraints.

Phase 1: Finding the 3-Collision. Finding the 3-collision requires $2^{128}/8 = 2^{125}$ triplets of encryptions, which can be obtained from $2^{42.5}$ distinct encryptions. This makes $2^{47.8}$ bytes of key-stream, about three times the maximally allowed quantity for a given key. This means that in the normal setting where LEX is restricted to produce $2^{46.3}$ bytes of key stream (80 terabytes), then out attack will only succeed with probability $\approx 1/32$. Indeed, under the normal restrictions, only 500×2^{32} encryptions are allowed, leading to $2^{120.3}$ triplets. Because each triplet leads to a 3-collision with probability 2^{-125}, it follows that the probability that the 3-collision exists is about 1/32. Our attack thus targets on average one key over 32.

The problem of detecting the 3-collision is even more acute than previously, because it can only be partially observed. The strategy is again to repeat the last two phases of the attack on the expected 2^{64} triplets matching on the observable 32 bits. The subsequent steps require about 2^{16} simple operations, yielding a total time complexity of 2^{80}.

Phase 2: Exploiting the 3-Collision. First of all, by exploiting the zero-difference bytes and the known key-stream bytes, it is possible to reconstruct the differences between the 3 concurrent processes in vast portions of the internal state. Figure 4.3 shows the situation.

- The differences in bytes 0, 2, 8 and 10 of w_4 are given by the leakage in x_5. Also, the differences are known to be zero in bytes 1, 3, 9 and 11 of z_4. Thus, thanks to observation 13n the differences can be found in bytes 0-3 and 8-11 of both z_4 and w_4.
- It is also known that the differences are zero in bytes 4-7 and 12-15 of both z_3 and w_3, and these zero differences propagate to x_3 and w_2. Accordingly, using Property 13 in z_2 and w_2 yields the missing differences in x_3, w_2 and z_2.

The second phase of the attack obtains the value of bytes 0-3 and 8-11 in x_2, as well as bytes 5,7,13 and 15 in x_3 and bytes 0,2,8 and 10 in x_4. This requires 2^{16} simple operations, and is illustrated by Figure 4.4. In fact, four independent processes could be run in parallel:

1-a. Guess bytes 7 and 13 of x_3 (these are the dotted squares). This enables to find the actual values in the 3 concurrent states in bytes 8–11 of z_3 and w_3, because the differences in x_3 are known. This also yields the differences in bytes 8-11 of x_4.

1-b. In both x_4 and y_4, the differences are now known in bytes 8 and 10. Only a fraction $2^{-9.5}$ of the differences are consistent in each byte. Thus, we expect to sieve *all*

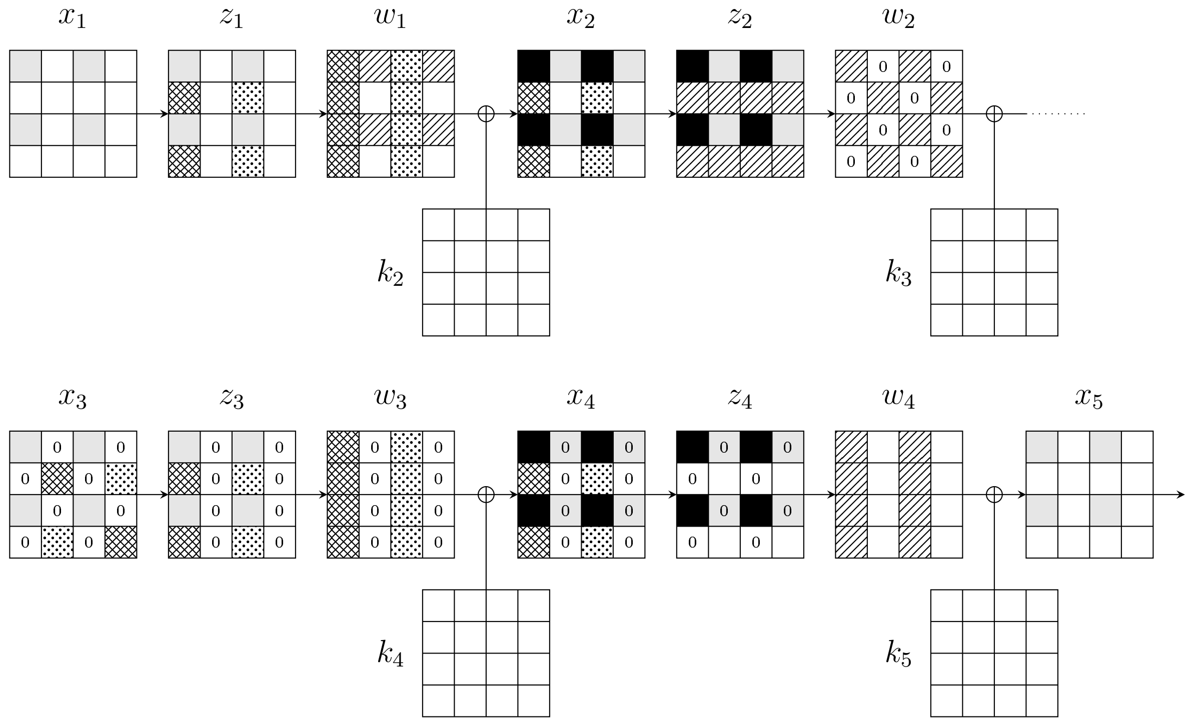


Figure 4.4: Second stage of the attack.

the wrong guesses in the previous step, and to be left with *only the right value.* In addition, the actual values in bytes 8 and 10 of x_4 are revealed.

2-a. Guess bytes 5 and 15 of x_3 (cross-hatched squares). This yields the differences in bytes 0–3 of x_4.

2-b. Using the same sieving technique allows us to filter just the right value for the two guesses, and to get bytes 0 and 2 in $_4$.

3-a. Guess bytes 1 and 3 in x_2 (cross-hatched squares). This yields the corresponding differences in w_1. Then, the differences in bytes 0–3 of w_1 and x_2 can be found thanks to Property 13.

3-b. The differences are known in bytes 0 and 2 in both x_2 and y_2. Therefore, the sieving technique yields the only feasible value for bytes 0–3 of x_2.

4. Guess bytes 9 and 11 in x_2 (dotted squares). Use the same difference propagation and sieving to recover the only value of bytes 8–11 in x_2.

Phase 3: a Guess-and-determine Finish. The third phase of the attack is a standard guess-and-determine procedure that guesses 2 bytes in order to completely recover k_3, and thus the master key. It requires 2^{16} simple operations, and is summarized by Figure 4.5. The actual values are known (from the previous phase) in gray squares. Hatched squares denotes known differences. The bytes are numbered in the order in which they can be computed. Circled bytes numbered 11 are guessed. In fact, some key bytes can be determined from the result of the second phase without guessing anything.

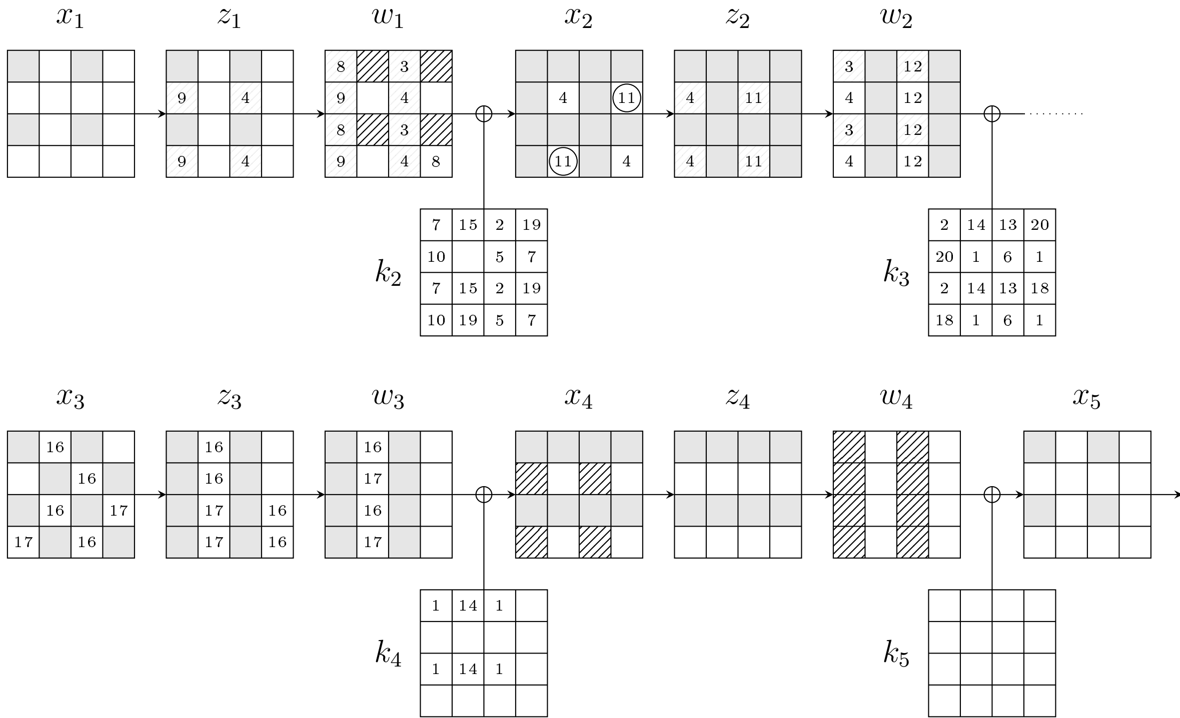


Figure 4.5: Third phase of the attack.

Step 1,5,10,13 and 18 result from the knowledge of both w_i and x_{i+1}. Step 2,6,7,14,15,19 and 20 exploit the key-schedule equations, and bytes obtained in previous steps. Steps 3,8 and 16 are just partial encryptions/decryptions. Step 4,9,12 and 17 use Property 13.

Chapitre 5

Fault Attacks on the AES

Since the early work of Piret and Quisquater [PQ03] on fault attacks against AES at CHES 2003, many works have been devoted to reduce the number of faults and to improve the time complexity of this attack. This attack is very efficient as a single fault is injected on the third round before the end, and then it allows to recover the whole secret key in 2^{32} in time and memory. However, since this attack, it is an open problem to know if provoking a fault at a former round of the cipher allows to recover the key. Indeed, since two rounds of AES achieve a full diffusion and adding protections against fault attack decreases the performance, some countermeasures propose to protect only the three first and last rounds.

In this chapter, we present several news fault attacks on the AES. We first show an improvement of the original attack of Piret and Quisquater reducing its overall complexity by a factor 2^8. Then we give two practical cryptographic attacks on one round earlier for all keysize variants. The first attack requires 5 faults and its complexity is around 2^{40} in time and memory while the second one is an impossible differential attack that requires at least 28 faults (depending on fault model) and recovers the secret key a bit faster.

Excepted the impossible differential attack, those attacks were found helped by the tool described Chapter 2.

5.1 Fault Analysis

Fault Analysis was introduced in 1996 by Boneh *et al.* [BDL97] against RSA-CRT implementations and soon after Biham and Shamir described differential fault attack on the DES block cipher [BS97]. Several techniques are known today to provoke faults during computations such as provoking a spike on the power supply, a glitch on the clock, or using external methods based on laser, Focused Ion Beam, or electromagnetic radiations [HCTW04]. These techniques usually target hardware or software components of smartcards, such as memory, register, data or address bus, assembly commands and so on [AK97]. After a query phase where the adversary collects pairs of correct and faulty ciphertexts, a cryptographic analysis of these data allows to reveal the secret key. The knowledge of a small difference at an inner computational step allows to *reduce* the analysis to a small number of rounds of a block cipher for instance. On the AES block cipher,

many such attacks have been proposed [BS03, DLV03, Gir04, MSS06, PQ03] and the first non trivial and the most efficient attack has been described by Piret and Quisquater in [PQ03].

5.1.1 Related Works

The embedded software and hardware AES implementations are particularly vulnerable to side channel analysis [BK07b, Bog07, SLFP04]. Considering fault analysis, it exists actually three different categories of attacks. The first category is non cryptographic and allows to reduce the number of rounds by provoking a fault on the round counter [AK97, CT05]. In the second category, cryptographic attacks perform fault in the state during a round [BS03, DLV03, Gir04, MSS06, PQ03] and in the third category, the faults are performed during the key schedule [CY03, Gir04, TFY07].

Several fault models have been considered to attack AES implementations. The first one and the less common is the random bit fault [BS03], where a fault allows to switch a specific bit. The more realistic and widespread fault model is the random byte fault model used in the Piret-Quisquater attack [PQ03], where a byte somewhere in the state is modified. These different fault models depend on the technique used to provoke the faults.

Piret and Quisquater described a general Differential Fault Analysis (DFA), against Substitution Permutation Network schemes in [PQ03]. Their attack uses a single random byte fault model injected between the two last MixColumns of AES-128. They exploited only 2 pairs of correct and faulty ciphertexts. Since this article was published in 2003, many works have proposed to reduce the number of faults needed in [Muk09, TM09], or to apply this attack to AES-192 and to AES-256 [Kim10].

There exist two kinds of countermeasures to protect AES implementations against fault attacks. The first category detects fault injection with hardware sensors for instance. However, they are specifically designed for one precise fault injection mean and do not protect against all different fault injection techniques. The second one protects hardware implementation against fault effects. This kind of countermeasures increases the hardware surface requirement as well as the number of operations. As a consequence, there is a tradeoff between the protection and the efficiency and countermeasures essentially only protect from existing fault attacks by taking into account the known state-of-the-art fault analysis. Therefore, the first three and the last three rounds used to be protected [CFGR10]. The same kind of countermeasures has been performed on DES implementation and a rich literature has been devoted to increase the number of attacked rounds as it is done in [Riv09]. Securing AES implementation consists in duplicating rounds, verifying operation with inverse operation for non-linear operations and with complementary property for linear ones, for example. Moreover, another approach computes and associates to each vulnerable intermediate value a cyclic redundancy checksum or, an error detection or correction code, for instance fault detection for AES S-Boxes [KRM08] as it has been proposed at CHES 2008. Our attacks could target any operation between MixColumns at the 6th round and MixColumns at the 7th round. Another countermeasure

consists in preventing from fault attack inside round [SSHA08]. However, it is possible to perform fault injection between rounds.

5.2 Meet-in-the-Middle Fault Analysis on AES

In this section, we remind how the Piret-Quisquater attack works and present the improvements found by the tool of Chapter 2.

5.2.1 Original Attack of Piret-Quisquater

In [PQ03], Piret and Quisquater assume a fault injection on one byte during the state computation between `MixColumns` at round 6 and `MixColumns` at round 7. on AES-128 as it is represented in the Figure 5.1. We assume that the fault is injected on the first column, the extension to other columns being straightforward. This attack allows to recover the last subkey in 2^{32} in time and negligible memory.

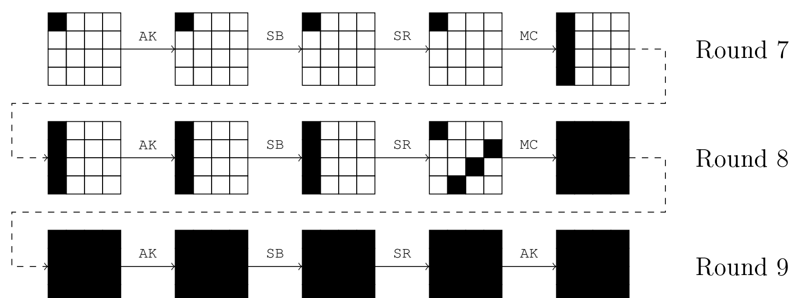


Figure 5.1: Fault injected on one byte between `MixColumns` at the 7th round and `MixColumns` at the 8th round on AES-128. Black bytes are active, white bytes are not.

The idea of the attack consists of partitioning the last subkey in four sets of 4 bytes such that each one can assume only a restricted number of values. As we know the difference in the ciphertexts, we also know the difference right after the last `SubBytes` operation in the state y_9. The state z_8 has exactly four active bytes, one per column. Guessing the difference in one of them allows us to deduce the differences in one column of the state just before the last `SubBytes` operation. As a consequence, we know the differences before and after the `SubBytes` and thus we can deduce its actual value (there is one on average) leading to the knowledge of four bytes of k_{10}.

As a result, this attack allows us to restrict the number of candidates for the last subkey to $(2^8)^4 = 2^{32}$. If the adversary has in his possession the corresponding plaintext then he may perform an exhaustive search on those candidates, otherwise asking for a second pair of correct and faulty ciphertexts should allow him to isolate the right one (with very high probability).

5.2.2 Improvement of Piret-Quisquater Attack

To improve the Piret-Quisquater attack, we essentially use the keyschedule equations between the two last subkeys. Indeed, for each of the 2^{32} candidates for the last subkey,

guessing the difference in the active byte of z_7 leads to the knowledge of 4 bytes of the subkey u_9 in the exact same way we obtained the subkey k_{10}. In the case of AES-128 both the subkeys u_9 and k_{10} are related and only $2^{32} \times 2^8 \times 2^{-32} = 2^8$ candidates for the last subkey should verify those equations.

Our attack is very simple as it is a basic meet-in-the-middle. First the adversary begins by guessing the difference in the active byte of the state z_7. Then he picks two active bytes in state z_8, guesses the differences in them, deduces the values of the corresponding 2 bytes of u_9 and 8 bytes of k_{10} and stores them. Then he builds a similar list from the two others active bytes of z_8. As the equations between the three last columns of u_9 and the full subkey k_{10} are linear, he can perform a meet-in-the-middle on the two lists leading to only $2^{16} \times 2^{16} \times 2^{-24} = 2^8$ candidates. Finally he keeps only the remaining candidates satisfying the equation concerning the byte of the first column of u_9. All in all, using hash tables this attacks can be performed in roughly 2^{24} operations and 10×2^{16} bytes of memory, and was found by the tool described Chapter 2. Note that the memory complexity can be decreased by a factor 5 since it is sufficient to get back the value of the differences in the actives bytes to recover the values of the key bytes.

Extension to AES-192. Our improved attack on AES-128 uses the fact that both the subkeys u_{r-1} and k_r are related essentially by affine equations. For AES-192, those subkeys are still related but now only the bytes of the two (instead of three) first columns of u_{11} are linearly dependent of k_{12}. Thus our attack still applies with the same complexity but the number of candidates remaining is now 2^{24}. To recover the master key, we need at least $24 - (16 + 4) = 4$ key bytes so another pair of correct and faulty ciphertexts is required and the fault must be located on one of the three other columns.

Extension to AES-256. In the case of AES-256 the subkeys u_{13} and k_{14} are independent so our attack restricts the number of possible values of $16 + 4 = 20$ key bytes to 2^{40}, in 2^{40} simple operations which is not an improvement.

5.2.3 Extension to One More Round

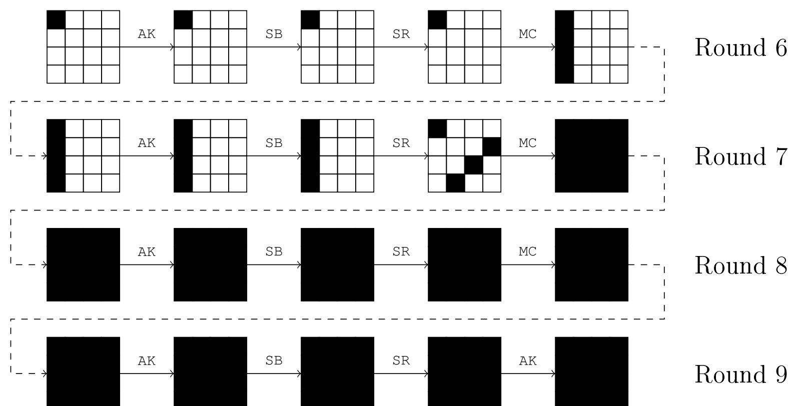


Figure 5.2: Fault injected on one byte between `MixColumns` at the 6th round and `MixColumns` at the 7th round on AES-128. Black bytes are active, white bytes are not.

We now realize a fault injection on one byte between MixColumns at the 6th round and MixColumns at the 7th round on AES-128. The fault is totally diffused at the whole 10th round as the Figure 5.2 shows it. This fault analysis requires 5 pairs of correct and faulty ciphertexts. The complexity of the attack is around 2^{40} in time and memory, and is schematized as follows:

1. Ask for 5 five pairs of correct and faulty ciphertexts.
2. Build the four lists:
   - $L_0 = \{(k_{10}[0], k_{10}[7], k_{10}[10], k_{10}[13], u_9[0])\}$
   - $L_1 = \{(k_{10}[3], k_{10}[6], k_{10}[9], k_{10}[12], u_9[13])\}$
   - $L_2 = \{(k_{10}[2], k_{10}[5], k_{10}[8], k_{10}[15], u_9[10])\}$
   - $L_3 = \{(k_{10}[1], k_{10}[4], k_{10}[11], k_{10}[14], u_9[7])\}$
3. Each list L_i contains 2^{40} elements and each one allows to deduce unique values for $\Delta_j x_8[i]$, $j = 1, \ldots, 5$, so in particular it allows to deduce unique values for the differences in the active byte of the first column of the state z_7 (which is byte 0 in Figure 5.2).
4. Build the list $L_{0,1}$ by taking all the element $(a, b) \in L_0 \times L_1$ such that a and b lead to the same value of the 5-byte sequence $(\Delta_1 z_7[0], \ldots, \Delta_5 z_7[0])$. Only $2^{40} \times 2^{40} \times 2^{-40} = 2^{40}$ candidates should remain.
5. Similarly, build the list $L_{0,1,2}$ and then the list $L_{0,1,2,3}$. Each of them still contains 2^{40} elements.
6. The list $L_{0,1,2,3}$ suggests 2^{40} values for 20 key bytes including the whole subkey k_{10}. By the keyschedule of AES-128, only $2^{40} \times 2^{-32} = 2^8$ may be correct.
7. For each of them decrypt the ciphertexts and keep only the ones leading to a single active byte in state z_7 at expected location. Only the right key is expected to remain since the probability for a wrong key to pass this test is $2^{-8\times15\times5} = 2^{-600}$.

All in all, storing the lists in hash tables indexed by the 5-byte sequence $(\Delta_1 z_7[0], \ldots, \Delta_5 z_7[0])$ allows to perform this attack in 2^{40} encryptions and $3 \times 5 \times 2^{40}$ bytes of memory. We stress that it is not required for the faults to be at the same location neither on the same column.

Reduction of Memory Requirement. The complexity of our attack is considered as practical but in practice it may be difficult to store as much as $3 \times 5 \times 2^{40}$ bytes of data on fast memory as computer RAM. Thus reducing the memory complexity may also speed-up the attack on common computer.

Our idea is to begin by guessing $\Delta_1 z_7[0]$. From there, the four lists $L_0, \ldots, L_3$ now contain only 2^{32} elements and can be built in as much operations. Indeed, from $\Delta_1 z_7[0]$ we obtain $\Delta_1 x_8[0]$ and if we guess the four key bytes $k_{10}[0]$, $k_{10}[7]$, $k_{10}[10]$ and $k_{10}[13]$ then we obtain $\Delta_1 y_8[0]$ allowing us to deduce the actual value of $x_8[0]$ (and $y_8[0]$) and leading to the knowledge of $u_9[0]$. As a consequence, we can reduce the memory complexity to $3 \times 5 \times 2^{32}$ bytes which is already more practical.

If we suppose the adversary have a *sixtuplet* consisting of a correct and five faulty ciphertexts then the memory complexity can be reduced even further. This time we begin by guessing $\Delta_1 z_7[0]$ and $\Delta_2 z_7[0]$ and the goal is to build the four lists from there in 2^{24}

operations. For instance, building the list L_0 by assuming that these values are known can be done as follows:

1. Build the list $L'_0 = \{(k_{10}[0], x_8[0])\}$
2. Each element of L'_0 allows to deduce unique values for:
   - $\Delta_j z_8[0]\ ,\ j = 1, 2$
   - $\Delta_j x_9[0]\ ,\ j = 1, \ldots, 5$
3. Guess $k_{10}[7], k_{10}[10], k_{10}[13]$
   - Deduce $\Delta_j x_9[1, 2, 3]\ ,\ j = 1, \ldots, 5$
   - Look in L'_0 for corresponding values of $k_{10}[0]$ and $x_8[0]$ using $\Delta_j x_9 = MC(\Delta_j z_9)$
   - Deduce $u_9[0]$ and $\Delta_j z_7[0]\ ,\ j = 3, 4, 5$

The lists L_1, L_2 and L_3 can be built in the same way, reducing the memory complexity of the overall attack to $3 \times 15 \times 2^{24}$ bytes.

This improvement makes the attack much more feasible. The implementation provided by the tool takes a little bit less than 13 days on a Core 2 Duo E8500 and 900MB of ram to test all possibilities but it can be improved by optimizing and parallelizing the C code.

Random Byte Fault Model. In case of unknown fault positions, we have to guess the column on which the fault occurs for each of the five pairs of correct and faulty ciphertexts. As a result, the time complexity is increased by a factor $4^5 = 2^{10}$ leading to an overall complexity of 2^{50}. Our attack can still be considered as practical but not enough to run it on a regular computer.

Extension to AES-192 and AES-256. Our attack can be easily applied on both the 192 and 256-bit versions of the AES without requiring more faulty pairs. For $i \in \{0, \ldots, 15\}$, let K_i be the 5 key bytes needed to decrypt the byte i of the state x_8 and L_i be the list of their possible values. In our attack we combined the lists L_0, L_1, L_2 and L_3 according to the value of the 5-byte sequence $(\Delta_1 z_7[0], \ldots, \Delta_5 z_7[0])$ and obtained a list $L_{0,1,2,3}$ containing 2^{40} candidates for the key bytes needed to decrypt the first column of x_8. But now those bytes are not sufficient to decrypt the ciphertexts and finding their right value cannot be done as in our previous attack. Instead, we build the list $L_{4,5,6,7}$ containing 2^{40} candidates for the key bytes needed to decrypt the second column of x_8. Then we perform a meet-in-the-middle between $L_{0,1,2,3}$ and $L_{4,5,6,7}$ as they must suggest the same value for the last subkey. Since $2^{40} \times 2^{40} \times 2^{-128} = 2^{-48}$, only the right value should remain. Once the last subkey is known and thus we may decrypt the ciphertexts by one round and apply the attack of Piret-Quisquater. However, continuing to combine the lists is faster since $L_8, \ldots, L_{15}$ now contain 2^8 elements each.

5.3 Impossible Differential Fault Attack on AES

In this section, we present a more efficient attack since we do not assume where the fault is provoked and the time complexity is reduced to 2^{42} inequality checks. However, this fault attack needs more faulty ciphertexts, less than 28, 41, 355 or 800 depending on the fault model. Our attack is based on the fact that it is impossible to have a zero-difference in state z_8 in the 9th round just before MixColumns operation; as Phan and

Yen mentioned this fact in [PY06] and developed with an example of the fault injected on the subkey k_7 in the key schedule. This fact is illustrated by the Figure 5.3. In this section, two principles are associated, the first one impossible differential, which is first published in [Knu98a, Knu98b], and the second one fault analysis, like [BGN05, PY06]. Our impossible differential fault analysis corresponds to 5-round impossible differential cryptanalysis attack, which is described in [BK00].

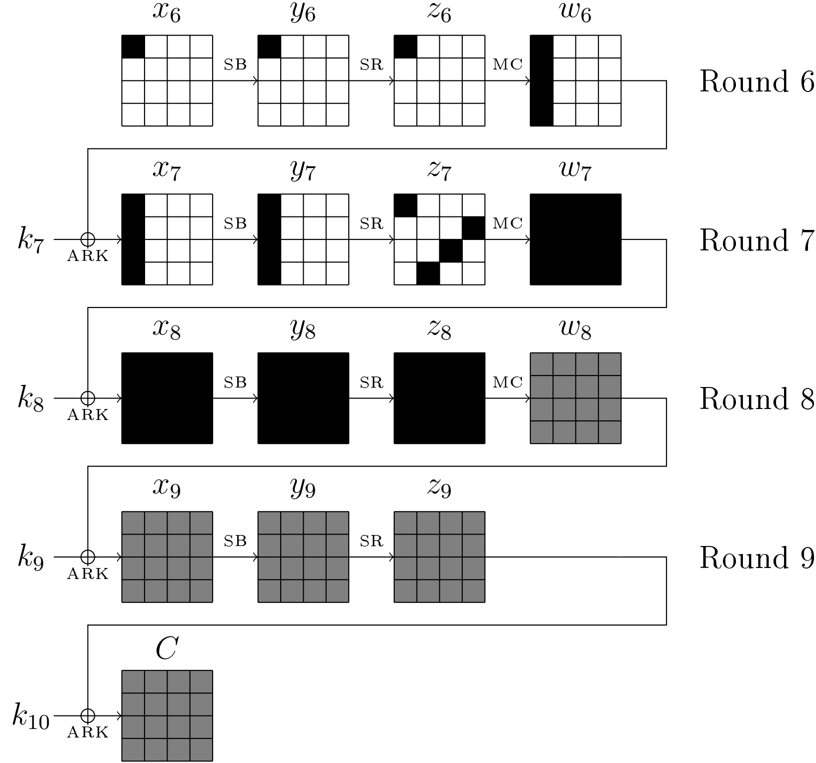


Figure 5.3: Colored bytes are active. Differences in black bytes are non-zero.

The Attack. Due to a well-known property of the differential through the MixColumn operation, all differences between bytes are *not null* at the internal state z_8.

$$\Delta z_8[i] \neq 0 \ \ , \text{ for } i \in \{0, \dots, 15\} \tag{5.1}$$

Moreover, we know that computing the difference in the z_8 can be done by guessing last subkey k_{10}. So we can reduce the number of candidates for the last subkey by keeping only the ones such that the inequalities hold. Furthermore, k_{10} can be partitioned in four sets of four bytes such that each one allows to compute the difference in one column of z_8. Thus we work on each of them separately.

Given m pairs of correct and faulty ciphertexts, each 4-byte candidate satisfies the inequalities with probability $\left(\frac{255}{256}\right)^{4m}$. Hence, to be left with n candidates we need on average m_n pairs where m_n is the smaller integer m such that $2^{32} \times \left(\frac{255}{256}\right)^{4m} \leqslant n$. Furthermore, this requires to check $\sum_{i=0}^{m_n} 4 \times 2^{32} \left(\frac{255}{256}\right)^{4i}$ inequalities which is equal to approximately 2^{40} as long as $n \leqslant 2^{24}$. As a consequence it is unclear that this attack can be implemented faster than the previous one.

Once we are left with n candidates for each of the four subsets of key bytes we can applied the attack described Section 5.2.3 with time and memory complexity reduced by a factor $2^{32}/n$. If we know where the faults are located then taking $n = 2^{24}$ makes the complexity of this part around 2^{32}. Otherwise we have to take $n = 2^{14}$ to obtain the same result. Anyway, the time complexity of this attack is dominated by checking around $4 \times 2^{40} = 2^{42}$ inequalities and the number of pairs of correct and faulty ciphertexts required is either 355 or 800 depending on the model.

Reducing the Data Complexity. An interesting property of reusing incorrect ciphertexts is described here. Let be given two faulty ciphertexts $\tilde{C}^1$ and $\tilde{C}^2$ built from the same plaintext while fault injection targeted on the same byte. Only MixColumns operation generates collision in one byte, whereas the others do not. Furthermore, if two different inputs of MixColumns only vary on one byte, the two outputs of MixColumns do not collide. As a consequence, the pair $(\tilde{C}^1, \tilde{C}^2)$ is a pair of correct and faulty ciphertexts, where the fault injection has been realized on one byte between MixColumns at the 6th round and MixColumns at the 7th.

More generally, if we have a *tuple* consisting of a correct and n faulty ciphertexts such that all their faults are on the same byte then we can generate $n(n+1)/2$ pairs of correct and faulty ciphertexts. As a result, we can reduce the data complexity of our attack from 355 to 28 and from 800 to 41 depending on the fault model. Furthermore, instead of testing all the inequalities we just have to verify that for each byte of z_8 the 27-byte (resp. 40-byte) sequence built by decrypting the difference in this byte from the ciphertexts does not contain the same value twice. So the number of inequalities we have to check becomes:

$$4 \times \sum_{i=0}^{27} 4 \times 2^{32} \left(\frac{255}{256}\right)^{\frac{4i(i-1)}{2}} \approx 4 \times \sum_{i=0}^{40} 4 \times 2^{32} \left(\frac{255}{256}\right)^{\frac{4i(i-1)}{2}} \approx 2^{39.5}.$$

All in all, the time complexity of the attack is still dominated by checking the inequalities but their number is decreased by a factor $2^{3.5}$. As the memory complexity is approximately 2^{32} bytes, we didn't run this attack and thus we are not able to compare it with the meet-in-the-middle attack described Section 5.2.3. However, this attack should be faster on a computer with enough RAM.

5.4 Conclusion

We have presented an attack on the $n-3^{th}$ round of AES and two different ones on the $n-4^{th}$ round. We studied the three AES versions against different fault models, and improved the previous known results. Current state-of-the-art countermeasure consists on protecting the three first rounds and the three last rounds of AES. All operations inside round need to be protected and state between rounds too. In order to defeat our fault analysis, all AES-128 rounds need to be protected against fault attacks. Considering AES-192 and AES-256, at least the last 5 rounds and the first 5 rounds need to be protected against fault analysis.

Chapitre 6

Faster Chosen-Key Distinguishers on Reduced-Round AES

In this chapter, we study another model that has been suggested to study the security of hash functions based on AES components. Knudsen and Rijmen [KR07] have proposed to consider *known-key* attacks since in the hash function domain, the key is usually known and the goal is to find two input messages that satisfy some interesting relations. In some setting, a part of the key can also be chosen (for instance when salt is added to the hash function) and therefore, cryptanalysts have also considered the model where the key is under the control of the adversary. The latter model has been called *chosen-key* model and both models belong to the *open-key* model. The chosen-key model has been popularized by Biryukov et al. in [BKN09], since a distinguisher in this model has been extended to a related-key attack on the full AES−256 version.

Related Work. Knudsen and Rijmen in [KR07] have been the firsts to consider known-key distinguishers on AES and Feistel schemes. The main motivations for this model are the following:

- if there is no distinguisher when the key is known, then there will also be no distinguisher when the key is secret,
- if it is possible to find an efficient distinguisher, finding partial collision on the output of the cipher more efficiently than birthday paradox would predict even though the key is known, then the authors would not recommend the use of such cipher,
- finally, such model where the key is known or chosen can be interesting to study the use of cipher in a compression function for a hash function.

In the same work, they present some results on Feistel schemes and on the AES. Following this work, Minier et al. in [MPP09] extend the results on AES on the Rijndael scheme with larger block-size.

In [BKN09], Biryukov et al. have been the firsts to consider the chosen-key distinguisher for the full 256-bit key AES. They show that in time $q \cdot 2^{67}$, it is possible to construct q-multicollision on Davies-Meyer compression function using AES−256, whereas for an ideal cipher, it would require on average $q \cdot 2^{\frac{q-1}{q+1}128}$ time complexity. In these chosen-key distinguishers, the adversary is allowed to put difference also in the key. Later, Nikolic et al. in [NPSS10], describe known-key and chosen-key distinguishers on Feistel

and Substitution-Permutation Networks (SPN). The notion of chosen-key distinguisher is more general than the model that we use: here, we let the adversary choose the key, but it has to be the same for the input and output relations we are looking for. We do not consider related-keys in this article. Then in [MPRS09], rebound attacks have been used to improve known-key distinguishers on AES by Mendel et al. and in [GP10], Gilbert and Peyrin have used both the **SuperSBox** and the rebound techniques to get a known-key distinguisher on 8-round AES-128. Last year at FSE, Sasaki and Yasuda show in [SY11] an attack on 11 Feistel rounds and collision attacks in hashing mode also using rebound techniques, and more recently, Sasaki et al. studied the known-key scenario for Feistel ciphers like Camellia in [SEHK12].

Our Results. In this chapter, we study 128- and 256-bit reduced versions of AES in the (single) chosen-key model where the attacker is challenged to find a key k and a pair of messages (m, m') such that $m \oplus m' \in E$ and $\texttt{AES}_k(m) \oplus \texttt{AES}_k(m') \in F$, where E and F are two known subspaces. On AES-128, we describe in that model a way to distinguish the 7-round AES in time 2^8 and the 8-round AES in time 2^{24}. In the case of the 7-round distinguisher, our technique improves the 2^{16} time complexity of a regular rebound technique [MRST09] on the SubBytes layer by computing intersections of small lists. The 8-round distinguisher introduces a problem related the **SuperSBox** construction where the key parameter is under the control of the adversary. As for AES-256, the distinguishers are the natural extensions of the ones on AES-128. Our results are reported in Table 6.1 and have been published in [DFJ12]. We have experimentally checked our results and examples are provided in the appendices. While those results do not threaten at all the security of the AES, we believe that our low-time distinguishers may pave the way for new cryptanalytic results on AES-based designs. For now, they can be useful to construct non-trivial inputs for the AES block cipher to be able to check the validity of some theoretical attacks, for instance [DKS10a]. Future works may include investigating the bridge between the open-key model and the secret-key traditional cryptanalysis framework by using efficient constraint satisfaction techniques.

6.1 Chosen-key distinguishers

6.1.1 Limited Birthday Distinguishers

In this section, we precise the distinguishers we are using. Our first goal is to distinguish the AES-128 from an ideal keyed-permutation in the chosen-key model. We will derive distinguishers for AES-256 afterwards. We are interested in the kind of distinguishers where the attacker is asked to find a key and a pair of plaintext whose difference is constrained in a predefined input subspace such that the ciphertext difference lies in an other predefined subspace.

Property 17. *Given two subspaces E_{in} and E_{out}, a key k and a pair of messages (x, y) verify the property on a permutation P if $x + y \in E_{in}$ and $P(x) + P(y) \in E_{out}$.*

This type of distinguisher looks like the limited birthday distinguishers introduced by Gilbert and Peyrin in [GP10] with a very close lower bound proved in [NPSS10],

Table 6.1: Comparison of our results to previous ones on reduced-round distinguishers of the AES-128 in the open-key model. Results from [BK09] are not mentioned since we do not consider related-keys in this chapter.

Target	Model	Rounds	Time	Memory	Ideal	Reference
AES-128	Known-key	7	2^{56}	-	2^{58} ⋆	[KR07]
	Known-key	7	2^{24}	2^{16}	2^{64}	[MPRS09]
	Chosen-key	7	2^{22}	-	2^{64}	[BN09]
	Chosen-key	**7**	$\mathbf{2^{8}}$	$\mathbf{2^{8}}$	$\mathbf{2^{64}}$	**Section 6.1.2**
	Known-key	8	2^{48}	2^{32}	2^{64}	[GP10]
	Chosen-key	8	2^{44}	-	2^{64}	[BN09]
	Chosen-key	**8**	$\mathbf{2^{24}}$	$\mathbf{2^{16}}$	$\mathbf{2^{64}}$	**Section 6.1.3**
AES-256	**Chosen-key**	**7**	$\mathbf{2^{8}}$	$\mathbf{2^{8}}$	$\mathbf{2^{64}}$	**Section 6.2.1**
	Chosen-key	**8**	$\mathbf{2^{8}}$	$\mathbf{2^{8}}$	$\mathbf{2^{64}}$	**Section 6.2.2**
	Chosen-key	**9**	$\mathbf{2^{24}}$	$\mathbf{2^{16}}$	$\mathbf{2^{64}}$	**Section 6.2.3**

⋆ Claimed by the authors as a *very inaccurate estimation of the [ideal] complexity.*

except that we allow the attacker more freedom; namely, in the choice of the key bits. To determine how hard this problem is, we need to compare the real-world case to the ideal scenario. In the latter, the attacker faces a family [2] of pseudo-random permutations $\mathcal{F} : \mathcal{K} \times \mathcal{D} \longrightarrow \mathcal{D}$, and would run a limited birthday distinguisher on a particular random permutation F_k to find a pair of messages that conforms to the subspace restrictions of Property 17. The additional freedom of this setting does not help the attacker to find the actual pair of messages that verifies the required property, because the permutation F_k has to be chosen beforehand. Put it another way, the birthday paradox is as constrained as if the key were known since no difference can be introduced in the key bits.

Therefore, even if we let the key to be chosen by the attacker, the limited birthday distinguisher from [GP10] applies in the same way. For known E_{in} and E_{out}, we denote $n_i = \dim(E_{in})$ and $n_o = \dim(E_{out})$. In terms of truncated differences, n_i (resp. n_o) represents the number of independent active truncated differences in the input (resp. output) of a random permutation $F_k \in \mathcal{F}$ (see Figure 6.1). Both n_i and n_o range in the interval between 1 and n^2, where $n = 4$ in the case of AES-128. Without loss of generality, we assume that $n_i \leqslant n_o$: the attacker thus considers F_k rather than its inverse, as it is easier to collide on $n^2 - n_o$ differences than on $n^2 - n_i$.

The attacker continues by constructing two lists L and L' of 2^{8n_i} plaintexts each by choosing a random value for the $n^2 - n_i$ inactive bytes of the input and considering all the n_i active ones in E_{in}. With a birthday paradox on the two lists L and L', she expects a collision on at most $2n_i$ bytes of the ciphertexts. In the event that $n^2 - n_o \geqslant 2n_i$, then $n^2 - 2n_i$ bytes have not a zero-difference in the ciphertext. Hence, we need to restart the birthday paradox process about $2^{8(n^2-n_o-2n_i)}$ times, which costs $2^{8(n^2-n_o-n_i)}$ in total. Otherwise, if $n^2 - n_o < 2n_i$, then a single birthday paradox with lists of size $2^{8(n^2-n_o)/2}$ is sufficient to get a collision on the $n^2 - n_o$ required bytes in time $2^{8(n^2-n_o)/2}$.

2. where both $\mathcal{K}$ and $\mathcal{D}$ are $\{0,1\}^{128}$ in the case of AES-128.

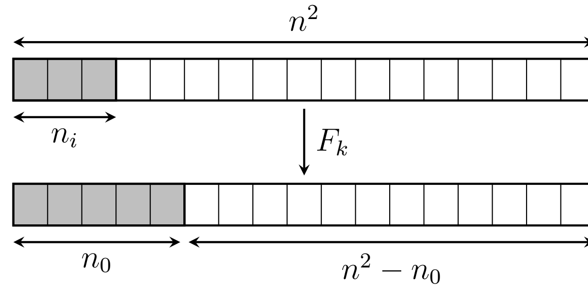


Figure 6.1: Assuming $n_i \leqslant n_o$, the attacker searches for a pair of input to the random permutation F_k differing in n_i known byte positions such that the output differs in n_o known byte positions. A gray cell indicates a byte with a truncated difference.

6.1.2 Distinguisher for 7-round AES-128

We consider the 7-round truncated differential characteristic of Figure 6.2, where the matrices of differences in both the plaintext and the ciphertext lie in matrix subspaces of dimension four. Indeed, the output difference lies in a subspace of dimension four since all the operations after the last SubBytes layer are linear. With respect to the description of the distinguisher (Section 6.1.1), the time complexity to find a pair of messages that conforms to those patterns in a family of pseudo-random permutations is 2^{64} basic operations.

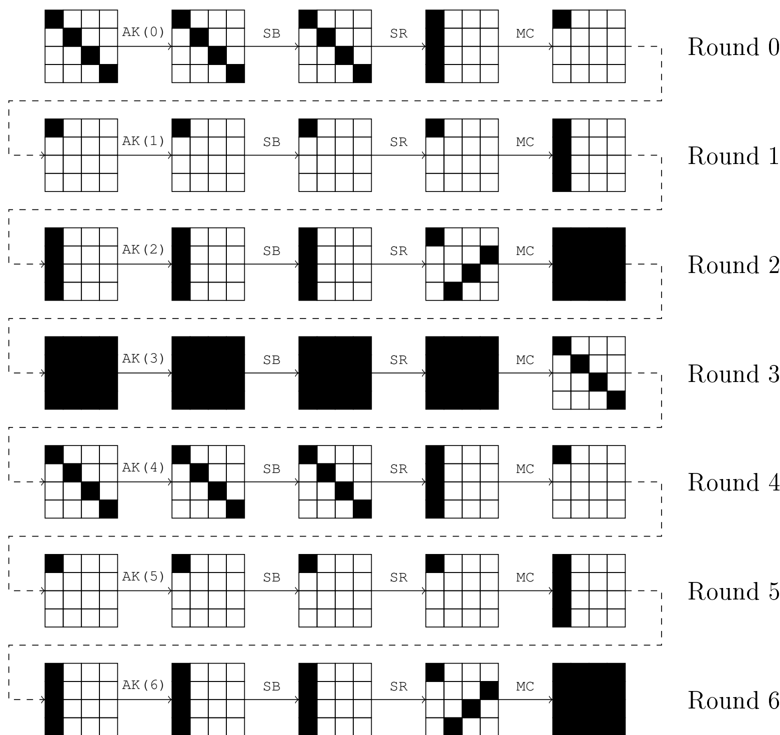


Figure 6.2: The 7-round truncated differential characteristic used to distinguish the AES-128 from a random permutation. Black bytes are active, white bytes are not.

The following of this section describes a way to build a key and a pair of messages that conform to the restrictions in time 2^8 basic operations using a memory complexity of 2^8 bytes. This complexity has to be compared to 2^{16} operations, which is the time complexity expected for a straightforward application of the rebound attack [MRST09] on the `SubBytes` layer of the `AES`. In that case, there are 16 random differential transitions around the `AES` S-Box, which happens to be all compatible[3] with probability 2^{-16}. Repeating with random differences 2^{16} times, we expect to find a pair of internal states that conforms to the randomized differences. In the following, we proceed slightly differently to reach a solution in time 2^8.

In terms of freedom degrees, we begin by estimating the number of solutions that we expect to verify the truncated differential characteristic. There are 16 bytes in the first message, 4 more independent ones in the second message and 16 others in the key: that makes 36 freedom degrees at the input. On a random input, the probability that the truncated differential characteristic being followed depends on the amount of freedom degrees that we loose in probabilistic transitions within the `MixColumns` transitions: -3 in round 0 to pass one $4 \to 1$ truncated transition, -12 in round 3 to pass four $4 \to 1$ transitions and -3 again in round 4 for the last $4 \to 1$ transition. In total, we thus expect

$$2^{8\times(16+4+16)}\, 2^{-8\times(3+12+3)} = 2^{8\times 18}$$

triplets (m, m', k) composed by a pair (m, m') of messages and a key k to conform to the truncated differential characteristic of Figure 6.2. Hence, we have 18 freedom degrees left to find such a triplet.

First, we observe that whenever we find such a solution for the middle rounds (round 1 to round 4), we are ensured that all the rounds will be covered as in the whole truncated differential characteristic due to an outward propagation occurring with probability 1. Hence, our strategy focuses on those rounds. The context is similar to the rebound scenario, where we first solve the inbound phase and then propagate it into the outbound phase.

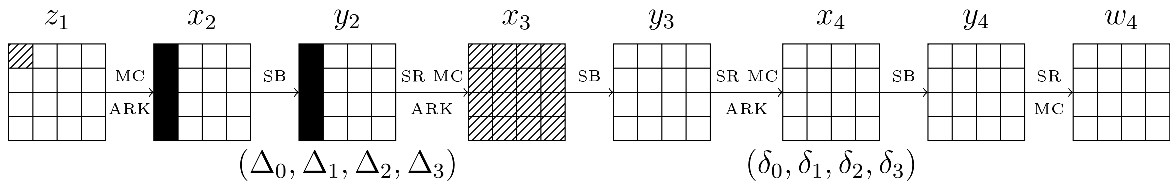


Figure 6.3: The 7-round distinguishing attack focuses of the middle rounds. Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences.

To reduce the number of valid solutions, we begin by fixing some bytes (Figure 8.2) to a random value: Δz_1 and $x_2[0..3]$. Therefore, we can deduce the values and differences in the

3. By compatible, we mean that we can find at least a pair of values that conforms to the differential transition. In the case of the `AES` S-Box, for a random differential transition $\delta \to \delta'$, this is known to be possible with probability close to 1/2.

first column of x_2 and y_2, as well as the difference Δx_3 by linearity. Let $[\Delta_0, \Delta_1, \Delta_2, \Delta_3]^{\mathrm{T}}$ be the column-vector of deduced differences in Δy_2 and $\mathrm{diag}(\delta_0, \delta_1, \delta_2, \delta_3)$ the differences in the diagonal of Δx_4. Linearly, we can express the differences around the `SubBytes` layer of round 3 (see Figure 6.4). As a consequence, from the differential properties of

Δx_3

$2\Delta_0$	Δ_3	Δ_2	$3\Delta_1$
Δ_0	Δ_3	$3\Delta_2$	$2\Delta_1$
Δ_0	$3\Delta_3$	$2\Delta_2$	Δ_1
$3\Delta_0$	$2\Delta_3$	Δ_2	Δ_1

$\xrightarrow{\text{SB}}$

Δy_3

$14\delta_0$	$11\delta_1$	$13\delta_2$	$9\delta_3$
$13\delta_3$	$9\delta_0$	$14\delta_1$	$11\delta_2$
$14\delta_2$	$11\delta_3$	$14\delta_0$	$9\delta_1$
$13\delta_1$	$9\delta_2$	$14\delta_3$	$11\delta_0$

Figure 6.4: Differences around the `SubBytes` layer of round 3: each Δ_j is fixed, whereas the δ_i are yet to be determined.

the AES S-Box, for $i, j \in \{0, \ldots, 3\}$, Δ_j suggests 2^7 different values for δ_i: we store them in the list $L_{i,j}$.

$$L_{i,j} = \Big\{ \delta_i \Big/ \Delta_j \to \delta_i \text{ is possible} \Big\}. \tag{6.1}$$

Once done, we build the list L_i, for $i \in \{0, \ldots, 3\}$:

$$L_i = \bigcap_{j=0}^{3} L_{i,j} = \Big\{ \delta_i \Big/ \forall j \in \{0, \ldots, 3\},\ \Delta_j \to \delta_i \text{ is possible} \Big\}. \tag{6.2}$$

Each $L_{i,j}$ being of size 2^7, we expect each L_i to contain 2^4 elements.

We continue by setting $\Delta x_4[0]$ to random value in L_0 and $x_4[0]$ to a random value, which allow to determine the value and difference in $y_4[0]$. Since the difference Δy_4 can only take 2^8 values due to the `MixColumns` transition of round 4, we also deduce Δw_4 and the remaining differences in Δy_4. The knowledge of Δy_4 suggests 2^7 possible values for δ_i. As before, we store them in lists called T_i, and we select a value for δ_i in $L_i \cap T_i$ (Figure 8.3). We expect each intersection to contain about 2^3 elements. More rigorously, if we assume that the lists $L_{i,j}$ and T_i are uniformly distributed, then the probability that L_0, $L_1 \cap T_1$, $L_2 \cap T_2$ and $L_3 \cap T_3$ are not empty is higher than 99.96% (see proof in Appendix 6.B). Finally, we compute the values in x_3 and in the diagonal of x_4.

We now need to find a key that matches the previous solving in the internal states: we build a partial pair of internal states that conforms to the middle rounds, but that sets 8 bytes on constraints in the key. Namely, if we denote k_i the subkey introduced in round i and $u_i = \mathsf{MC}^{-1}(k_i)$, then both u_3 and k_4 have four known bytes (see Figure 6.6). We start by fixing all the bytes marked by `1` in u_3 to random values: this allows to compute the values of all `2`'s in the two last columns of k_3. By the column-wise operations of AES key schedule, we can get the values of all bytes marked by `3`. As for the `4`'s, we get them since there are four known bytes among the eight in the first columns of u_3 and k_3. Again, the

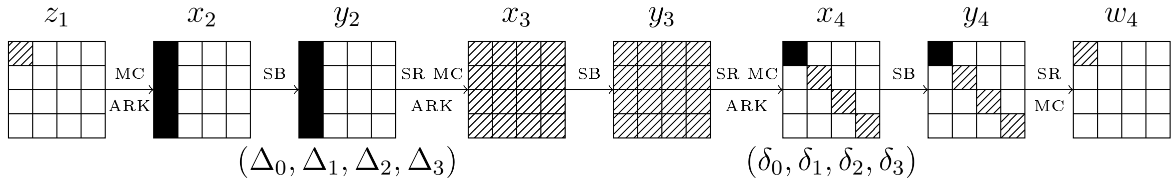


Figure 6.5: The 7-round distinguishing attack focuses of the middle rounds. Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences.

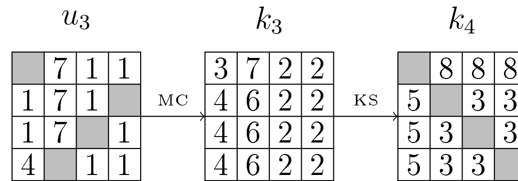


Figure 6.6: Generating a compatible key: gray bytes are known, and numbers indicate the order in which we guess or determine the bytes.

key schedule gives the `5`'s and `6`'s, and the `MixColumns` the `7`'s. Finally, we determine values for all the byte tagged by `8` from the key schedule equations. By inverting the key schedule, we are thus able to compute the master key k.

All in all, we start by getting a partial pair of internal states that conforms to the middle rounds, continue by deriving a valid key that matches the partial known bytes and determine the rest of the middle internal states to get the pair on input messages. The bottleneck of the time and memory complexity occurs when handling the lists of size at most 2^8 elements to compute intersections. Note that those intersections can be done in roughly 2^8 operations by representing lists by 256-bit numbers and then perform a logical AND.

In the end, we build a pair of messages (m, m') and a key k that conforms to the truncated differential characteristic of Figure 6.2 in time 2^8 basic operations, where it costs 2^{64} in the generic scenario. We note that among the 18 freedom degrees left for the attack, we uses only 10 by setting 10 bytes to random values, such that we expect $2^{8\times8} = 2^{64}$ solutions in total. All those solutions could be generated in time 2^{64} by iterating over all the possibilities of the bytes marked by `1` in Figure 6.6.

We implemented the described algorithm to verify that it indeed works, and we found for instance the triplet (m, m', k) reported in Appendix 6.A.

6.1.3 Distinguisher for 8-round `AES-128`

We consider the 8-round truncated differential characteristic of Figure 6.7, where the matrices of differences in both the plaintext and the ciphertext lie in the same matrix subspaces of dimension four as before. Indeed, the output difference lies in a subspace of dimension four since all the operations after the last `SubBytes` layer are linear. Again,

the distinguisher previously described (Section 6.1.1) claims that the time complexity to find a pair of messages that conforms to those patterns in a family of pseudo-random permutations runs in time 2^{64} operations.

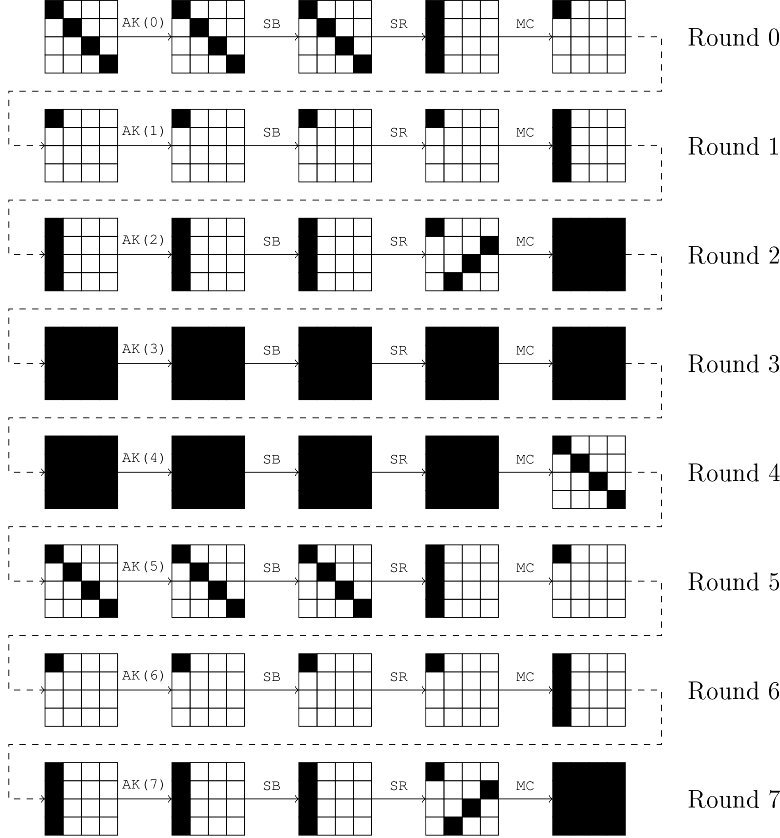


Figure 6.7: The 8-round truncated differential characteristic used to distinguish the AES-128. Black bytes are active, white bytes are not.

The following of this section describes a way to build a key and a pair of messages that conform to the restrictions in time and memory complexity 2^{24}. We note that it is possible to optimize the memory requirement to 2^{16}. As in the previous section, there are 36 freedom degrees at the input, which shrink to 18 after the consideration of the truncated differential characteristic. Therefore, we also expect $2^{8\times 18}$ solutions in the end.

First of all, we observe that finding 2^{24} triplets (m, m', k) composed by a key and a pair of internal states that conform to the rounds 2 to 5 is sufficient since the propagation in the outward rounds is done with probability 2^{-24} due to the MixColumns transition of round 1. The following analysis consequently focuses of those four middle rounds.

We now describe an instance of a problem that we use as a building block in our algorithm, which is related to the keyed **SuperSBox** construction.

Problem 1. *Let a and b two bytes. Given a 32-bit input and output differences* Δ_{in} *and* Δ_{out} *of a* $\mathbf{SuperSBox}_k$ *for an unknown 32-bit k, find all the pairs of* AES*-columns* (c, c') *and keys k such that:*

i. $c + c' = \Delta_{in}$,

ii. $\mathbf{SuperSBox}_k(c) + \mathbf{SuperSBox}_k(c') = \Delta_{out}$,

iii. $\mathbf{SuperSBox}_k(c) = [a, b, \star, \star]^{\mathrm{T}}$.

Considering the key k known and the case where there is no restriction on the output bytes (*iii*), we would expect this problem to have one solution on average. Finding it would naively require 2^{32} computations by iterating over the 2^{32} possible inputs and checking whether the output has the correct Δ_{out} known difference. The additional constraints on the two output bytes reduce the success of finding a pair (c, c') of input to 2^{-16}, but if we allow the four bytes in the key k to be chosen, then we expect 2^{16} solutions to this problem.

To find all of them in 2^{16} simple operations, we proceed as follows (Figure 6.8): the two output bytes a and b being known, we can deduce the values of the two associated bytes before the last `SubBytes`, $\tilde{a}$ and $\tilde{b}$ respectively. We can also deduce the difference in those bytes since the output difference is known. Then, we guess the two unset differences at the input of the last `SubBytes`: the differences then propagate completely inside the **SuperSBox**. At both `SubBytes` layers, by the differential properties of the `AES` S-Box, we expect to find one value on average for each of the six unset transitions. Consequently, the input and output of the `AddRoundKey` operation are known, which determines the four bytes of k. In the end, we find the 2^{16} solutions of Problem 1 in time 2^{16} operations.

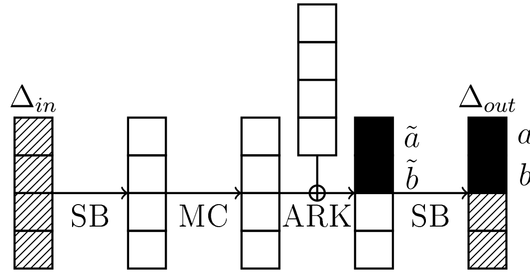


Figure 6.8: Black bytes have known values and differences, hatched bytes have known differences and white bytes have unknown values and/or differences.

To apply this strategy to the 8-round truncated differential characteristic of Figure 6.7, we start by randomizing the difference Δy_2, the difference Δw_5 and the values in the first column of w_5. Due to the linear operations involved, we deduce $\Delta x_3 = \Delta w_2$ from Δy_2 and Δy_4 from Δw_4. To use the previous algorithm, we randomize the values of the two first columns of w_4 (situtation in Figure 6.9). Doing so, the four columns of y_4 are constrained on two bytes each and have fixed differences. Consequently, the four **SuperSBoxes** between x_3 and y_4 keyed by the four corresponding columns of k_4 conforms to the requirements [4] of Problem 1. In time and memory complexity 2^{16}, for $i \in \{0, 1, 2, 3\}$, we store the 2^{16} solutions for the ith **SuperSBox** associated to the ith column of x_4 in the list L_i.

4. The positions of the known output bytes differ, but the strategy applies in the same way.

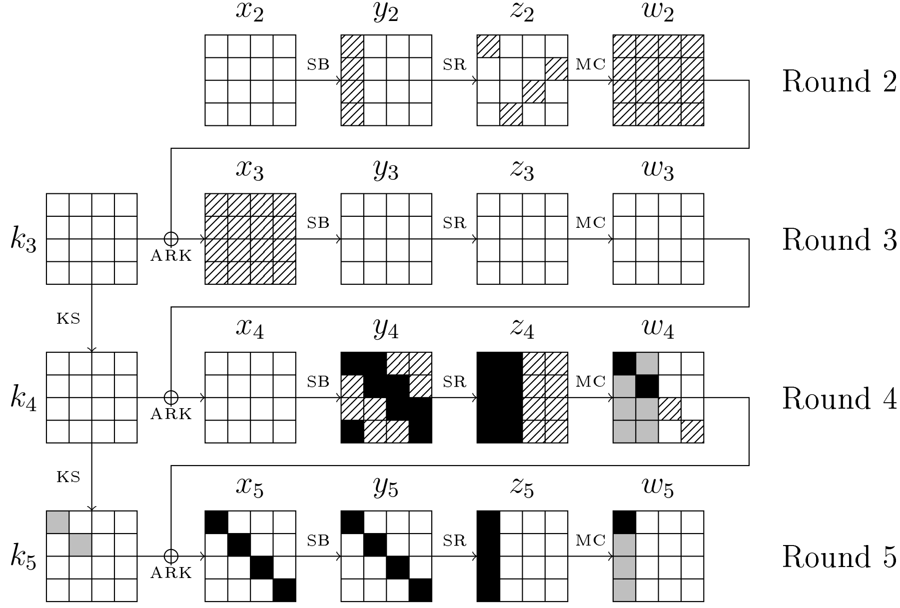


Figure 6.9: Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences.

We continue by observing that the randomization of the bytes in w_4 actually sets the value of two diagonal bytes in k_5, $k_5[0]$ and $k_5[5]$, which imposes constraints of the elements in the lists L_i. We start by considering the 2^{16} elements of L_3, and for each of them, we learn the values $x_4[12..15]$ and $k_4[12..15]$. Due to the column-wise operations in the key schedule, we also deduce the value of $k_4[0]$. Filtering the elements of L_0 which share that value of $k_4[0]$, we are left with 2^8 elements for bytes $x_4[0..3]$ and $k_4[0..3]$. At this point, we constructed $2^{16+8} = 2^{24}$ solutions in time 2^{24} that we store in a list $L_{0,3}$.

As $k_5[5]$ has been previously determined, we can deduce $k_4[5] = k_5[5] + k_5[1]$ from the AES key schedule for each of the entry of $L_{0,3}$. Again, this adds an 8-bit constraint on the elements of L_1: we expect 2^8 of them to match the condition on $k_4[5]$. In total, we could construct a list $L_{0,1,3}$ of size $2^{24+8} = 2^{32}$, whose elements would be the columns 0, 1 and 3 of x_4 and k_4, but as soon as we get 2^{24} elements in that list, we stop and discard the remaining possibilities.

Finally, to ensure the correctness of the choice in the remaining column 2, we need to consider the MixColumns operation in round 4 and the subkey k_5. Indeed, as soon as we choose an element in both $L_{0,1,3}$ and L_2, x_4, k_4 and k_5 become fully determined, but we need to ensure that the values $x_5[10]$ and $x_5[15]$ equal to the known ones. In particular, for $x_5[10]$, we have:

$$
\begin{aligned}
k_4[10] + k_5[6] &= k_5[10] && (6.3)\\
&= w_4[10] + x_5[10] && (6.4)\\
&= z_4[8] + z_4[9] + 2z_4[10] + 3z_4[11] + x_5[10], && (6.5)
\end{aligned}
$$

and for $x_5[15]$:

$$\begin{aligned} k_4[11] + k_5[7] + k_4[15] &= k_5[11] + k_4[15] && (6.6)\\ &= k_5[15] && (6.7)\\ &= w_4[15] + x_5[15] && (6.8)\\ &= 3z_4[12] + z_4[13] + z_4[14] + 2z_4[15] + x_5[15], && (6.9) \end{aligned}$$

where (6.3), (6.6) and (6.7) come from the key schedule, (6.4) and (6.8) from the AddRoundKey and (6.5) and (6.9) use the equation from the MixColumns. Hence, for each element of $L_{0,1,3}$, we can compute:

$$\mathsf{S}(x_4[8]) + k_4[10] := x_5[10] + k_5[6] + \mathsf{S}(x_4[13]) + 2\,\mathsf{S}(x_4[2]) + 3\,\mathsf{S}(x_4[7]), \quad (6.10)$$

$$k_4[11] + 2\,\mathsf{S}(x_4[11]) := k_5[7] + k_4[15] + 3\,\mathsf{S}(x_4[12]) + \mathsf{S}(x_4[1]) + \mathsf{S}(x_4[6]) + x_5[15] \quad (6.11)$$

and lookup in L_2 to find $2^{16}\,2^{-8\times 2} = 1$ element that match those two byte conditions. We create the list L by adding the found element from L_2 to each entry of $L_{0,1,3}$.

All in all, in time and memory complexity 2^{24}, we build L of size 2^{24} and we now exhaust its elements to find one that passes the 2^{-24} probability of the $4 \to 1$ backward transition in the MixColumns of round 1. Indeed, an $a \to b$ transition in the MixColumns layer cancels $4 - b$ output bytes, so that it would happen with probability $2^{-8(4-b)}$ for a random input a. Consequently, we expect to find a pair (m, m') of messages and a key k that conforms to the 8-round truncated differential characteristic of Figure 6.7 in time 2^{24} when it requires 2^{64} computations in the ideal case.

Among the 18 available freedom degrees available to mount the attack, we uses 17 of them, which means that we expect to have 2^8 solutions. We could have them in time 2^{32}, but since we discarded 2^8 elements in the algorithm described, we get only 1 in time 2^{24}. We note that it is possible to gain a factor 2^8 in the memory requirements of our attack since we can implement the algorithm without storing the lists L_0, $L_{0,3}$ and $L_{0,1,3}$, by using hash tables for L_1, L_2 and L_3.

We also implemented the described algorithm to verify that it indeed works, and we found for instance the triplet (m, m', k) reported in Appendix 6.A.

6.2 Extention to AES-256

The two distinguishers described in the previous section can be easily extended in distinguishers on the AES-256. The main idea is to use the 16 additional freedom degrees in the key to extend the truncated differential characteristics by introducing a new fully active round in the middle.

6.2.1 Distinguisher for 7-round AES-256

The first step of the attack described in the 7-round distinguisher on AES-128 (Section 6.1.2) still applies in the case of AES-256 since it does not involve the key schedule.

Then, we can generate a compatible key easily since there are only two subkeys involved: we can just choose bytes of k_3 and k_4 as we want, except the imposed ones, and deduce the master key afterwards. This yields to a distinguisher with time and memory complexities around 2^8.

6.2.2 Distinguisher for 8-round AES-256

We use a similar approach as the 7-round distinguisher on AES-128 of Section 6.1.2, but the truncated differential characteristic has one more fully active round in the middle [5].

We begin by choosing values for Δz_1 and $x_2[0..3]$. This allows to deduce Δx_2, Δy_2, and Δx_3. Then, we also set random values for Δw_5 and for the diagonal of x_5 to obtain both Δx_5 and Δy_4. Now, we find a value for Δx_4, which is compatible with Δx_3 and Δy_4. Indeed, we can not take an arbitrary value for Δx_4 because the probability that it fits is very close to 2^{-32}. However, we can find a correct value with the following steps:

1. Store the 2^7 possible values for $\Delta x_4[0]$ in a list L_0.
2. In a similar way, make lists L_1 with $\Delta x_4[1]$, L_2 with $\Delta x_4[2]$ and L_3 with $\Delta x_4[3]$.
3. Choose a value for $(x_3[0], x_3[5], x_3[10], x_3[15])$ and compute $\Delta x_4[0..3]$.
4. If $\Delta x_4[0..3]$ is not in $L_0 \times L_1 \times L_2 \times L_3$, then go back to step 3.

On average, we go back to the step 3 only $(2^{8-7})^4 = 2^4$ times since lists are of size 2^7. In the same way, we can obtain values for the other columns of x_4.

At this point, we computed actual values in all those internal states, and we need to generate a compatible key. Finding one can be done using the procedure described in Figure 6.10. Bytes tagged by 1 are chosen at random, odd steps use the key schedule equations and even steps the properties of MixColumns.

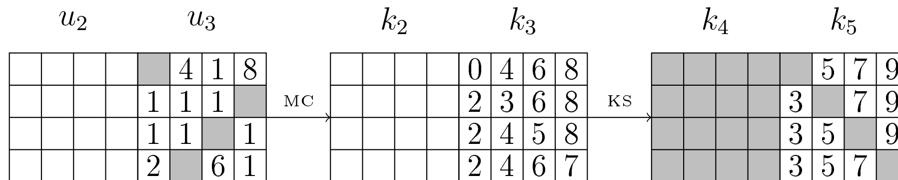


Figure 6.10: Generating a compatible key: gray bytes are known, and numbers indicate the order in which we guess or determine the bytes.

6.2.3 Distinguisher for 9-round AES-256

We begin as in Section 6.1.3 by choosing the difference Δy_2, the difference Δw_6 and the values in the first column of w_6. Then, we deduce $\Delta w_2 = \Delta x_3$ from Δy_2 and Δy_5 from Δw_5. In addition, we set x_3 to a random value, which allows to determine Δx_4. In order to apply the result from Problem 1 again, we set the values in two first columns of w_5 to random values.

5. In that case, the truncated differential characteristic is thus the one from Figure 6.7.

As before, for $i \in \{0,1,2,3\}$, we store in the list L_i the 2^{16} possible values of the i-th column of x_5 and the i-th column of k_5. Unlike previously, we also obtain values of the i-th column of $\mathsf{SR}(k_4)$, but the scenario of the attack still applies. We start by observing that bytes of L_0 allow to compute $k_4[1]$ and $k_4[13]$, which are bytes of L_3. Thus, we can merge L_0 and L_3 in a list $L_{0,3}$ containing 2^{16} elements. Then, we construct the list $L_{0,2,3}$ containing 2^{24} elements of $L_{0,3} \times L_2$. Finally, from bytes of $L_{0,2,3}$, we can compute:

$$3z_5[11] := k_4[2] + \mathsf{S}(k_5[15]) + k_4[6] + k_4[10] + z_5[8] + z_5[9] + 2z_5[10] + x_6[10], \tag{6.12}$$

$$z_5[14] + k_4[3] := \mathsf{S}(k_5[12]) + k_4[7] + k_4[11] + k_4[15] + 3z_5[12] + z_5[13] + 2z_5[15] + x_6[15]. \tag{6.13}$$

As a consequence, we expect only one element of L_1 to satisfy those two byte conditions and so, we obtain 2^{24} solutions for the middle rounds. All in all, this yields to a distinguisher with a time complexity around 2^{24} and a memory requirement around 2^{16} using the same trick given in Section 6.1.3.

6.3 Conclusion

In this chapter, we studied the Advanced Encryption Standard and shown how to find a pair of messages and a key that satisfy some property a lot more efficiently than a generic attack based on the birthday paradox for both `AES-128` and `AES-256`. Our new results improve the previous claimed ones by reaching very practical complexities, and give new insights of the open-key model for block ciphers, and hash functions based on block ciphers.

On `AES-128`, we shown efficient distinguishers for versions reduced to seven and eight rounds, and verified in practice that they indeed work by implementing the actual attacks. We described precisely the algorithms to get the valid inputs, and by applying the same strategy, we deduced similar results for `AES-256`. Namely, we get efficient distinguishers on versions reduced to seven, eight and nine rounds.

6.A. Experimental verification

Table 6.2: Example of a pair of messages (m, m') that conforms to the 7-rounds truncated differential characteristic for AES-128 of Section 6.1.2. The master key found by the attack is: 93CA1344 10A7EBDF B659C8AF ECC59699. The lines in this array contains the values of two internal states before entering the corresponding round, as well as their difference.

Round	m	m'	$m \oplus m'$
Init.	E5FC5DFE 79A851F7 7EB9E366 51C3D9C5	F8FC5DFE 79C951F7 7EB96566 51C3D96E	1D000000 00610000 00008600 000000AB
0	76364EBA 690FBA28 C8E02BC9 BD064F5C	6B364EBA 696EBA28 C8E0ADC9 BD064FF7	1D000000 00610000 00008600 000000AB
1	65CC94D1 85BE1AD3 F3D75BF1 ACCBB8BD	8DCC94D1 85BE1AD3 F3D75BF1 ACCBB8BD	E8000000 00000000 00000000 00000000
2	E93319CD 88F41390 10623230 F66BFBAD	C92309FD 88F41390 10623230 F66BFBAD	20101030 00000000 00000000 00000000
3	89C79074 E09E6F44 F1DBAB2F F984FCC4	1404532A 09774F8D 24BF1AFA CD551921	9DC3C35E E9E920C9 D564B1D5 34D1E5E5
4	867A12E6 BF19139C 1C848362 400030D3	047A12E6 BF5B139C 1C847C62 400030D7	82000000 00420000 0000FF00 00000004
5	84606BEA 0E22D904 3BF29061 9F454807	4B606BEA 0E22D904 3BF29061 9F454807	CF000000 00000000 00000000 00000000
6	FF867544 274436AF 75ECC287 A6BF72F6	3C6A996B 274436AF 75ECC287 A6BF72F6	C3ECEC2F 00000000 00000000 00000000
End	C49E4CB3 0C944043 D5ED6D3B 247E3843	2563B1AF 68F0EC8B A6788B48 EEF27E05	E1FDFD1C 6464ACC8 7395E673 CA8C4646

Table 6.3: Example of a pair of messages (m, m') that conforms to the 8-round truncated differential characteristic for AES-128 of Section 6.1.3. The master key found by the attack is: 98C45623 6CA00686 301E836D 614DFAB0. The lines in this array contains the values of two internal states before entering the corresponding round, as well as their difference.

Round	m	m'	$m \oplus m'$
Init.	9588B342 D43D04D4 AB298AE1 E43687DB	0B88B342 D46904D4 AB29D0E1 E4368728	9E000000 00540000 00005A00 000000F3
0	0D4CE561 B89D0252 9B37098C 857B7D6B	934CE561 B8C90252 9B37538C 857B7D98	9E000000 00540000 00005A00 000000F3
1	53FEBB0F 6BFF8E5E B471A8E3 1A2232A3	0EFEBB0F 6BFF8E5E B471A8E3 1A2232A3	5D000000 00000000 00000000 00000000
2	E9F44380 991A8ECB F7B18344 2C936CEB	65B2054A 991A8ECB F7B18344 2C936CEB	8C4646CA 00000000 00000000 00000000
3	2977F65C 3883EDEF 615D3C9E 5CE5384B	8F24A5A9 2398C0D9 10CEDEEF DFEEB0C3	A65353F5 1B1B2D36 7193E271 830B8888
4	BB1DB144 2BE947C3 5FCD89DF DF1CA0EB	82188658 42FFCAAE B337F0CA 09AB1513	3905371C 69168D6D ECFA7915 D6B7B5F8
5	C3E1961D 02A9713E 770A20D4 5470FA8F	8DE1961D 029B713E 770A3AD4 5470FA27	4E000000 00320000 00001A00 000000A8
6	D79D534C 33CC3861 76635DCD 548870C9	EB9D534C 33CC3861 76635DCD 548870C9	3C000000 00000000 00000000 00000000
7	D7F645C6 89358035 09847940 D831EFDE	0211A2F4 89358035 09847940 D831EFDE	D5E7E732 00000000 00000000 00000000
End	16E58308 DFD78F11 A8B05B9D C0A0363E	E49CFA83 D4DC9207 FC4CF3C9 9B3BF6FE	F279798B 0B0B1D16 54FCA854 5B9BC0C0

6.B. Probability of success

We are interested in the probability that the intersection of four or five subsets of $\{1, \ldots, 255\}$ each of size 128 being empty.

To evaluate it, let $\mathcal{P}$ denote the set of subsets $X \subset \{1, \ldots, 255\}$ such that $|X| = 128$. We also define:

$$T(n, k) := |\{(X_1, \ldots, X_n) \in \mathcal{P}^n \mid |X_1 \cap \ldots \cap X_n| = k\}| \qquad \text{for } n \geqslant 1, k \geqslant 0.$$

In others words, $T(n,k)/|\mathcal{P}^n|$ is the probability that the intersection of n elements from $\mathcal{P}$ has a size equal to k. We observe that $T(n,k)$ satisfies the following recurrence relation:

$$\begin{cases} T(1,k) &= |\mathcal{P}| \text{ if } k = 128, 0 \text{ otherwise} \\ T(n+1,k) &= \sum_{l=k}^{128} T(n,l)\binom{l}{k}\binom{255-l}{128-k} \qquad \text{for } n \geqslant 1, k \geqslant 0. \end{cases}$$

Indeed, if we fix a set $X \subset \{1,\ldots,255\}$, then a set $Y \in \mathcal{P}$ such that $|X \cap Y| = k$ is obtained by choosing k elements in X and $128-k$ elements in X^c.

Using Maple, we found that the probability of failure of the distinguisher described in Section 6.1.2 is:

$$\frac{T(4,0)}{|\mathcal{P}|^4} \times \left(\frac{T(5,0)}{|\mathcal{P}|^5}\right)^3 \approx 0.04\%.$$

Chapitre 7

Exhausting Demirci-Selcuk Meet-in-the-Middle Attacks against Reduced-Round AES

In this chapter we continue the study of meet-in-the-middle attacks on the AES but when the adversary has access to more data than in Chapter 3. More precisely, we describe a generalization of the attack shown by Demirci and Selçuk [DS08] at FSE 2008 which was an improvement of the Gilbert and Minier attack [GM00] using meet-in-the-middle technique instead of collision ideas. Their results at that time use a very small data complexity 2^{32} but require high precomputation and memory in 2^{216}. They need a hash table parametrized by 24 byte values and the attack only works for the 256-bit and 192-bit versions thanks to a time/memory trade-off which significantly increases the data and time complexity.

Here, we consider another direction to improve on Demirci and Selçuk attack using only meet-in-the-middle techniques. We generalize their attack and we automatize the search of the best ones. To perform this search, we use the tool described Chapter 2, but only on the keyschedule equations instead of the system of equations describing the whole AES. These equations are sparse in the number of Sbox and consequently, the complexity of the search is very low. In particular, we have been able to reach a better overall complexity than Demirci and Selçuk, without requiring more data than 2^{32} chosen plaintexts.

Those results have been published at FSE 2013 and some improvements presented in this chapter are used in the next one to reach the best attacks on 7, 8 and 9 rounds for the 128, 192 and 256-bit versions of the AES respectively.

7.1 Attack of Demirci and Selçuk and Improvements

In this section, we remind Demirci and Selçuk attack together with its improvements which are the main results used in our attack. They are based on the study of the encryption of structured sets of 256 plaintexts in which one active byte takes each one of the 256 possible values exactly once, and each one of the other 15 bytes is a (possibly different)

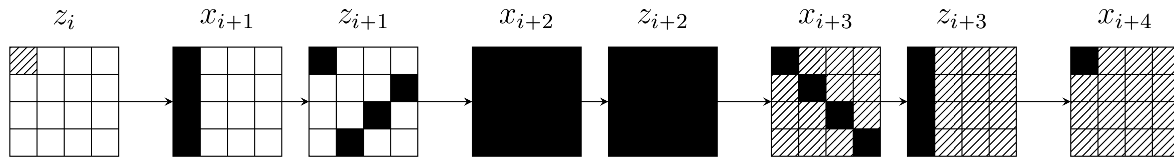


Figure 7.1: 4 AES-rounds. The 25 black bytes are the parameters of Property 18. Hatched bytes play no role. The differences are null in white squares

constant. Such a structure is called a δ-set. We refer the reader to [DS08] and [DKS10a] for details.

7.1.1 The Demirci and Selçuk Attack

At FSE 2008, Demirci and Selçuk described the following 4-round property for AES.

Property 18. *Consider the encryption of a δ-set through four full* AES *rounds. For each of the 16 bytes of the state, the ordered sequence of 256 values of that byte in the corresponding ciphertexts is fully determined by just 25 byte parameters. Consequently, for any fixed byte position, there are at most $(2^8)^{25} = 2^{200}$ possible sequences when we consider all the possible choices of keys and δ-sets (out of the $(2^8)^{256} = 2^{2048}$ theoretically possible 256-byte sequences).*

The 25 parameters are intermediate state bytes for any message of the δ-set and their positions depend on the active byte of the δ-set and on which byte we want to build values. As depicted on Figure 7.1, if there are both at position 0 then the 25 parameters are the first column of x_{i+1}, the full state x_{i+2}, the first column of z_{i+3} and $x_{i+4}[0]$. Indeed, if those bytes are known for one of the messages, we can compute the value of $x_{i+4}[0]$ for each message of the δ-set as follows:

1. Knowing the 256 differences in the full state z_i we can compute the 256 differences in the full state x_{i+1} because $\Delta x_{j+1} = \mathrm{MC}.\Delta z_j$ for any round number j, where MC is the matrix used in the MixColumns operation.
2. Knowing the value of the first column of x_{i+1} for one message we can now compute the value of this column for all messages.
3. Then we apply the Sbox on those bytes and get the value of $z_{i+1}[0]$, $z_{i+1}[7]$, $z_{i+1}[10]$ and $z_{i+1}[13]$ for each message of the δ-set.
4. The differences are null in all the other bytes of z_{i+1} so we know the 256 differences in the full state z_{i+1}.
5. In the same way we obtain the 256 differences in the full state z_{i+2} and then in the first column of z_{i+3} to finally compute the 256 values of $x_{i+4}[0]$

They first use this property to mount a basic meet-in-the-middle attack on 7 rounds AES-256 depicted on Figure 7.2 and its procedure is roughly as follows:

- **Preprocessing phase:** Compute all the 2^{200} possible sequences according to Property 18, and store them in a hash table.

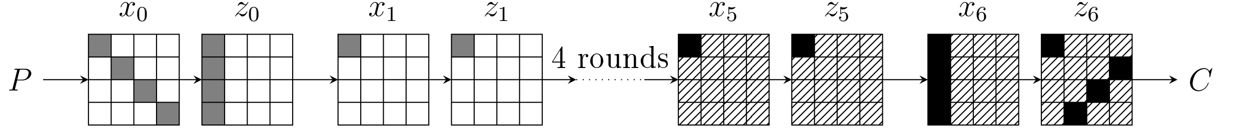


Figure 7.2: Online phase of Demirci and Selçuk attack. $\mathcal{B}_{on}$ is composed by gray and black bytes. Gray bytes are used to identify a δ-set and to order it. Black bytes are used to build the sequence from ciphertexts. Hatched bytes play no role. The differences are null in white squares.

- **Online phase:**
  1. Ask for a structure of 2^{32} chosen plaintexts such that the main *diagonal* takes the 2^{32} possible values and the remaining bytes are constant.
  2. Choose one plaintext and guess the first column of its intermediate state z_0 and byte $z_1[0]$.
  3. For each of the 255 non-zero values of Δz_1 compute the corresponding difference in the plaintext using the guessed bytes.
  4. Order the obtained δ-set according to the value of the state byte $z_1[0]$.
  5. Guess the first column of x_6 and the byte $x_5[0]$ for one of the message and deduce those state bytes for the 256 ciphertexts.
  6. Build the sequence and check whether it exists in the hash table. If not, discard the guess.

Note that the parameters of both the online and offline phases are state bytes which we shall refer in the sequel as respectively $\mathcal{B}_{on}$ and $\mathcal{B}_{off}$. The complexity of the attack depends directly on how many values can assume those state bytes and how fast can we enumerate them. Indeed, bytes of $\mathcal{B}_{off}$ (resp. $\mathcal{B}_{on} \cup P \cup C$) are related by the AES equations and thus lead to the knowledge of some linear combinations of the (sub)keys bytes. Then it may exist some relations derived from the key-schedule between them, allowing to reduce the number of assumed values. In the sequel, we will denote by $\mathcal{K}_{off}$ (resp. $\mathcal{K}_{on}$) the vector space generated from these linear combinations. For instance, in the case of the described attack and if the last MixColumns is omitted:

- $\{k_{-1}[0], k_{-1}[5], k_{-1}[10], k_{-1}[15], k_0[0], u_5[0], k_6[0], k_6[7], k_6[10], k_6[13]\}$ is a basis of $\mathcal{K}_{on}$,
- $\{u_1[0], u_2[0], u_2[7], u_2[10], u_2[13], k_3[0], k_3[5], k_3[10], k_3[15], k_4[0]\}$ is a basis of $\mathcal{K}_{off}$.

All in all, this attack has a data complexity of 2^{32} chosen plaintexts, a time complexity of $2^{80} \times 2^8$ partial encryptions/decryptions, and a memory requirement of 2^{200} 256-byte sequences. The memory complexity of this attack is too high to apply it on the 128 and 192-bit versions. But its time complexity is low enough to mount an attack from it on 8 rounds AES-256. This is done by fully guessing the last subkey, decrypting the last round and applying the 7-round attack, which increases the time complexity by a factor 2^{128}.

7.1.2 Previous Improvements of the Original Attack

We summarize the main improvements to the original attack of Demirci and Selçuk.

Difference Instead of Value. Demirci and Selçuk showed that the number of parameters can be reduced to 24 in Property 18 by considering the sequence of the differences instead of values because in that case $x_{i+4}[0]$ is not needed.

Data/Time/Memory Trade-Off. They also showed that one can do a classical trade-off by storing in the hash table only a fraction of the possible sequences. Then the attacker has to repeat the online phase many times to compensate the probability of failure if the sequence is not present in the table which will increase the data and time complexities. In other word, if the attack has a complexity (D, T, M) (D for the data, T for the time complexity of the online phase and M for the memory) then it is possible to modify it to reach a complexity equal to $(D \times N, T \times N, M/N)$ for any positive N such that $D \times N$ is smaller than the size of the codebook. This trade-off allows to adapt the attack on 7 rounds of AES-256 to attack the 192-bit version.

Data Recycling. The structure of 2^{32} plaintexts used in the attack contains 2^{24} δ-sets. Thus the data may be reused 2^{24} times in the Data/Time/Memory Trade-Off.

Time/Memory Trade-Off. Kara observed that considering the sequence of the differences instead of values allows to remove $x_5[0]$ from $\mathcal{B}_{off}$ (as Demirci and Selçuk did) or from $\mathcal{B}_{on}$.

Multiset. A multiset is an unordered set in which elements can occur many times. Dunkelman *et al.* introduce them to replace the functional concept used in the attack and propose to store in the hash table unordered sequences of 256 bytes instead of ordered sequences. Moreover, they claim that a multiset still contains enough information to make the attack possible. Indeed they showed that given two random functions $f, g : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_{2^8}$, the multisets $[f(0), \ldots, f(255)]$ and $[g(0), \ldots, g(255)]$ are equal with a probability smaller than $2^{-467,6}$. Combined to the fact that the Sbox is a bijection, the main gain is to remove $z_1[0]$ from $\mathcal{B}_{on}$ since it was used only to ordered the δ-set, and thus the time complexity is decreased by a factor 2^8. Finally, we note that a multiset contains about 512 bits of information and its representation can be easily compressed into 512 bits of space while an ordered sequence needs $256 \times 8 = 2048$ bits.

Differential Enumeration. In [DKS10a], Dunkelman *et al.* introduce a more sophisticated trade-off which reduces the memory without increasing the time complexity. The main idea is to add restrictions on the parameters used to build the table such that those restrictions can be checked (at least partially) during the online phase. More precisely, they impose that sequences stored come from a δ-set containing a message m which belongs to a pair (m, m') that follows a well-chosen differential path. Then the attacker first focus on finding such pair before to identify a δ-set and build the sequence. The next chapter is dedicated to this technique.

7.2 Generalization of the Demirci and Selçuk Attack

The basic attack of Demirci and Selçuk requires a huge memory and a relatively small time complexity. The classical data/time/memory trade-off allows to *balance* these complexities by increasing the data complexity and randomizing the attack. In this section we present new improvements to reduce the data complexity increase which leads to almost 2^{16} variants of the Demirci and Selçuk attack and we explain how to find the best ones between them.

7.2.1 New Improvements of the Original Attack

In this section we summarized our new improvements that allow us to reduce the increase of the data complexity and, sometimes, to keep the deterministic nature of the original attack.

Difference Instead of Value. The sequences stored in the table have the form $[f(0) + f(0), \dots, f(0) + f(255)]$ where f is a function that maps the value of $z_i[0]$ to the value of $x_{i+4}[0] + k_{i+3}[0]$. But, as shown Section 7.1.1, the procedure used to build the table produces functions that map the value of $\Delta z_i[0]$ to the value of $\Delta x_{i+4}[0]$ and then the only effect of mapping the value of $z_i[0]$ is to set the value of the subkey byte $u_i[0]$ (*i.e.*, $u_i[0] \in \mathcal{K}_{off}$). In another hand, if we store in the table sequences of the form $[f(0), \dots, f(255)]$ where f is a function that maps the value of $\Delta z_i[0]$ to the value of $\Delta x_{i+4}[0]$, then each δ-set can be ordered in 256 ways, saving data in the classical data/time/memory trade-off described Section 7.1.2. Furthermore, in the case of a δ-set encryption, each byte of the first columns of x_{i+1} assumes the 256 values. As a consequence, setting one of those bytes to 0 when building the hash table can be compensated by trying the 256 orders of a δ-set without randomizing the attack.

Multiset. Note that, given a sequence of 256 bytes $b_0, \dots, b_{255}$, $b_i = b_j$ implies that the multisets $[b_i{+}b_0, \dots, b_i{+}b_{255}]$ and $[b_j{+}b_0, \dots, b_j{+}b_{255}]$ are equal too. But Dunkelman et al. shown that given a random function $f : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_{2^8}$, the multiset $[f(0) + f(1), \dots, f(0) + f(255)]$ contains on average 162 different values out of 256. Thus we conclude that a δ-set can be reused $162 \approx 2^{7,34}$ times on average. This remark holds on for the multisets stored in the hash table during the precompution phase and so the memory requirement must be corrected by a factor $2^{-0,66}$.

Time/Memory Trade-Off. To improve the attack of Demirci and Selçuk our idea is to store in the sequences the 256 differences in a linear combination of bytes of x_5 instead of the 256 differences in a byte of x_5. As the matrix used in `MixColumns` operation is MDS (Property 13), minimal equations involving Δz_i and Δx_{i+1} contains exactly 5 variables such that k are on a column c of Δz_i and $5 - k$ are on the column c of Δx_{i+1}, with $1 \leqslant k \leqslant 4$ for any round number i. We emphasize that Demirci and Selçuk only consider cases $k = 1$ and $k = 4$. The size of the set $\mathcal{B}_{on}$ (resp. $\mathcal{B}_{off}$) is determined by k and it decreases (resp. increases) when k increases. Thus we can trade time by memory and vice-versa without affecting the data complexity. Furthermore, contrary to the other data/time/memory trade-offs, the attack need not to be randomized. Attacks taking advantage of this trade-off are described Section 7.3.3 and 7.3.5.

New Data/Time/Memory Trade-Off. The idea of the previous trade-off can be applied to the δ-set. Instead of considering sets of 256 plaintexts such that one byte assumes the 256 values and the others are constant, we consider set of 256 plaintexts such that exactly 5 bytes of z_i and x_{i+1} are active. We still call such a set a δ-set. The consequences on the attack are the same as the previous trade-off but it now affects the size of the structure needed and bytes of z_i must be guessed in the online phase despite the use of unordered sequences. An attack taking advantage of this trade-off is described Section 7.3.4.

7.2.2 Finding the Best Attack

Once the round-reduced AES is split into three parts, the new improvements allow to mount $(4 \times \binom{8}{5})^2 \approx 2^{15.6}$ different attacks but there are only $(4 \times (\binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4}))^2 \approx 2^{11.8}$ possible sets $\mathcal{B}_{on}$ (resp. $\mathcal{B}_{off}$) to study. To exhaust all of them and find the best attacks we decide to automatize the search. Thus for each set we need to answer to the two following questions:

- How many values can assume those state bytes?
- How fast can we enumerate them?

A priori, this is not an easy task because S-boxes are involved in the keyschedule. To perform it we used the tool presented Chapter 2, and more precisely in the setting described Section 2.3.1.0.

Note that the tool can be applied directly to the set $\mathcal{B}_{off}$ (resp. $\mathcal{B}_{on}$) and the system of equations describing the AES but it is faster to apply it on a basis of $\mathcal{K}_{off}$ (resp. $\mathcal{K}_{on}$) and the keyschedule equations since its complexity is exponential in the number of S-box involved.

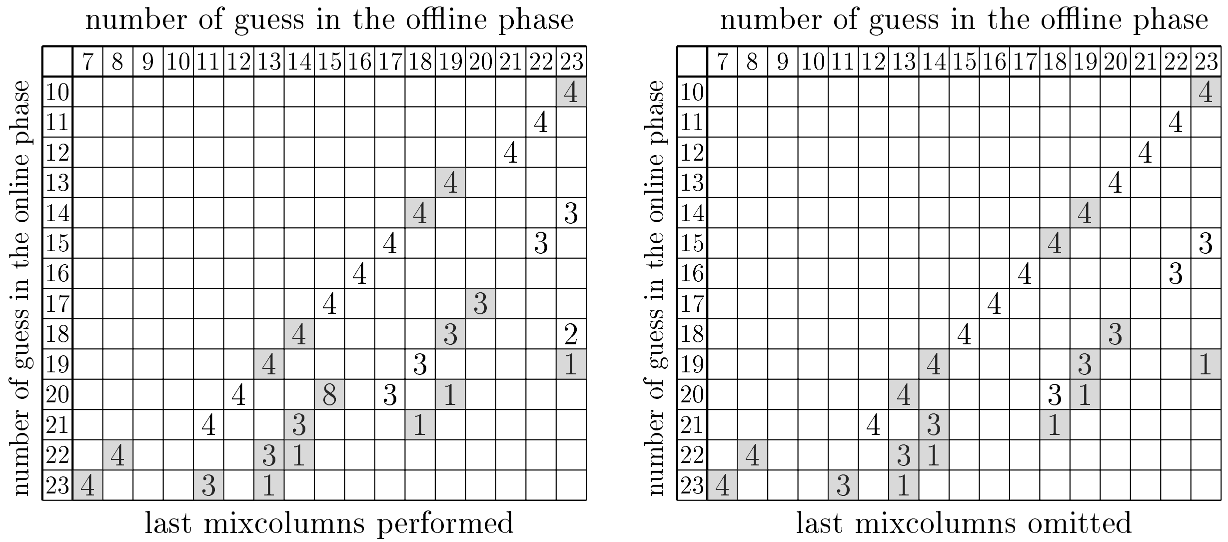
Finally we were able to perform an exhaustive search over all the parameters for all round-reduced versions of AES for the three key lengths in less than an hour on a personal computer.

7.3 Results

In this section we present the results obtained by exhausting the variants of the attack of Demirci and Selçuk. We give an overview of the complexities reached and describe three new attacks requiring at most 2^{32} chosen plaintexts and minimizing the maximum between the time complexity (counted in AES encryption) and the memory complexity (counted in 128-bit block).

7.3.1 Overview of the Results

Our best results on 7 and 8 rounds are summarized on Figures 7.3, 7.4 and 7.5. They give the ($\log_{256}$ of) data complexity reached as a function of the number of guess to perform in the online phase and in the offline phase. A gray cell means that the corresponding attack is deterministic while the other attacks are obtained by applying the classical data/time/memory trade-off.



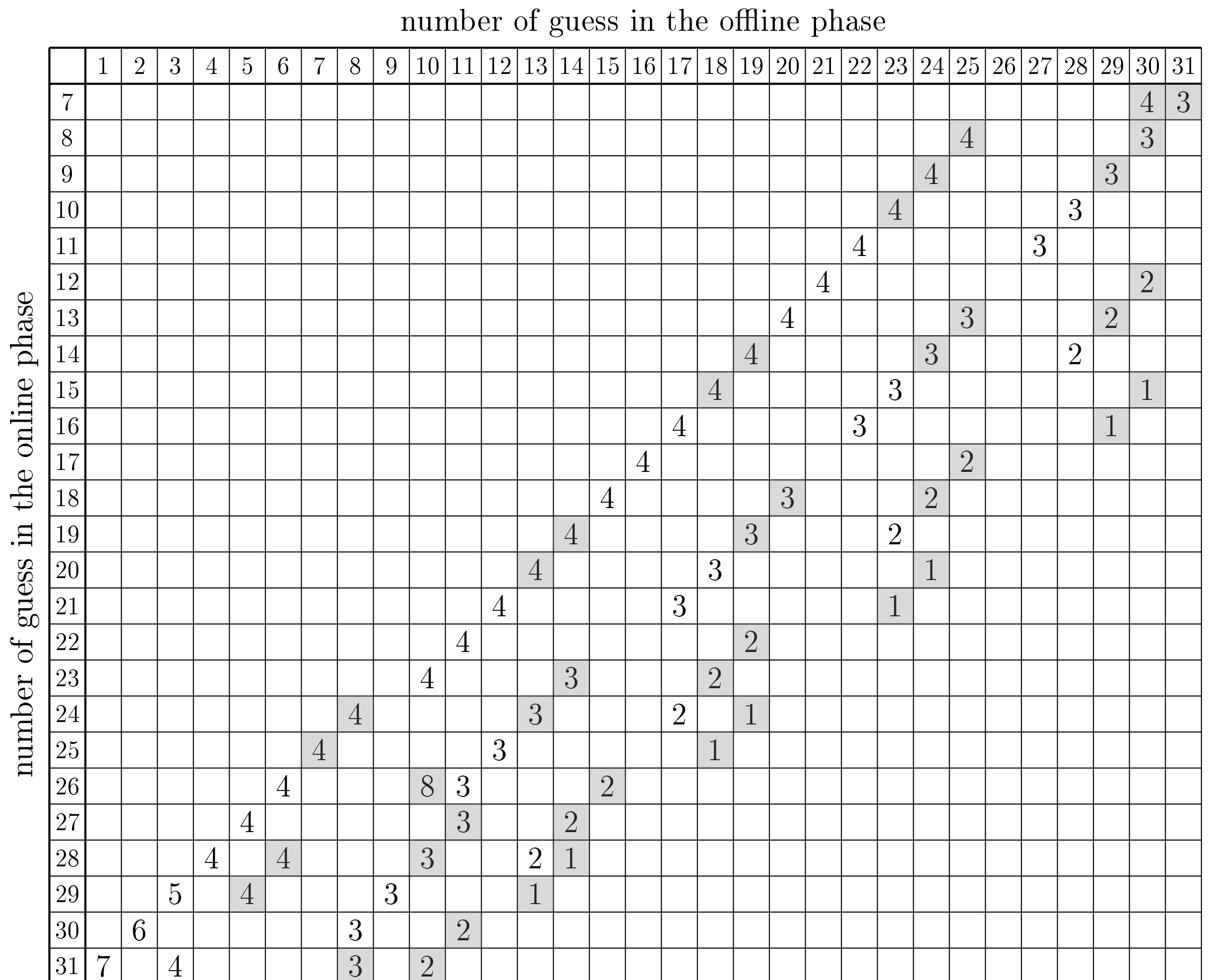
number of guess in the offline phase (columns) / number of guess in the online phase (rows) — last mixcolumns performed

	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10																	4
11																4	
12															4		
13													4				
14												4					3
15											4					3	
16										4							
17									4					3			
18								4					3				2
19							4					3					1
20						4			8		3		1				
21					4			3				1					
22		4					3	1									
23	4				3		1										

number of guess in the offline phase (columns) / number of guess in the online phase (rows) — last mixcolumns omitted

	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10																	4
11																4	
12															4		
13														4			
14													4				
15												4					3
16											4					3	
17										4							
18									4					3			
19								4					3				1
20							4					3	1				
21						4		3				1					
22		4					3	1									
23	4				3		1										

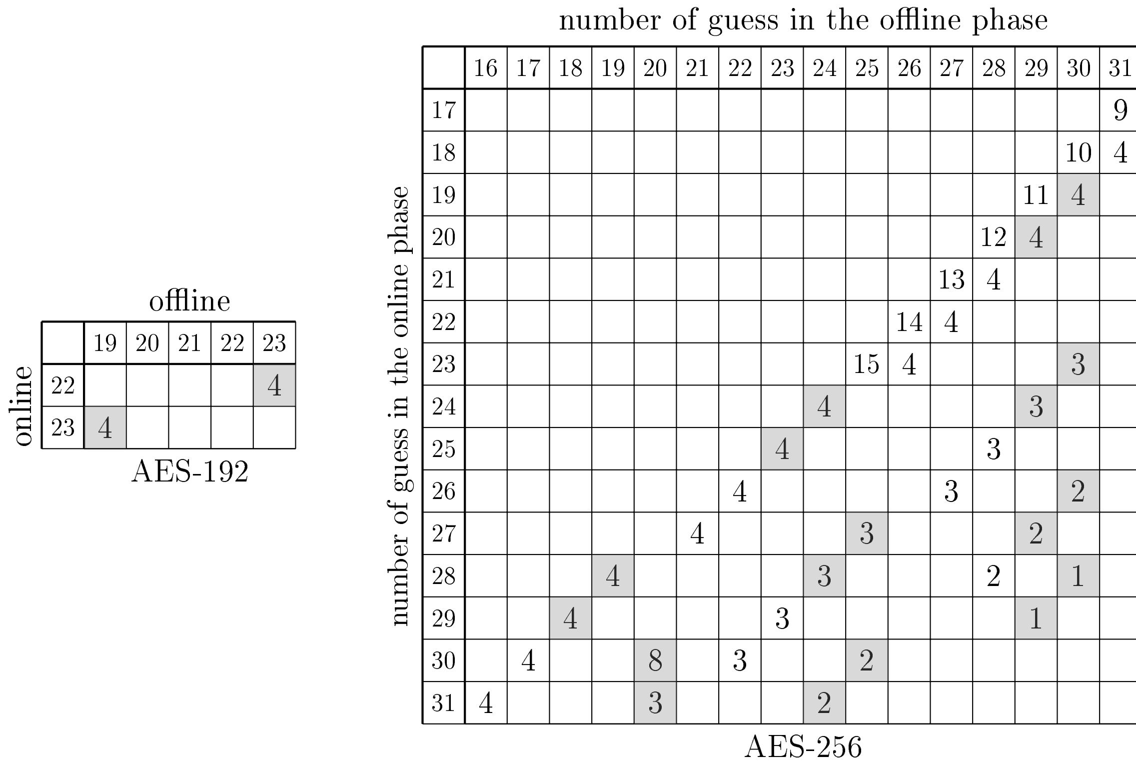
Figure 7.3: Best variants on 7 rounds AES-192.



number of guess in the offline phase (columns) / number of guess in the online phase (rows)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
7																														4	3
8																									4					3	
9																								4					3		
10																							4					3			
11																						4					3				
12																					4									2	
13																				4					3				2		
14																			4					3				2			
15																		4					3							1	
16																	4					3							1		
17																4									2						
18															4					3				2							
19														4					3				2								
20													4					3						1							
21												4					3						1								
22											4								2												
23										4				3				2													
24								4					3				2		1												
25							4					3						1													
26						4				8	3				2																
27					4						3			2																	
28				4		4				3			2	1																	
29			5		4				3				1																		
30		6						3			2																				
31	7		4					3		2																					

Figure 7.4: Best variants on 7 rounds AES-256.



AES-192 (online × offline)

online \ offline	19	20	21	22	23
22					4
23	4				

AES-256 (number of guess in the online phase × number of guess in the offline phase)

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
17																9
18															10	4
19														11	4	
20													12	4		
21												13	4			
22											14	4				
23										15	4				3	
24									4					3		
25								4					3			
26							4					3			2	
27						4				3				2		
28				4					3				2		1	
29			4					3						1		
30		4			8		3			2						
31	4				3				2							

Figure 7.5: Best variants on 8 rounds.

We observe that almost all the best attacks work with only 2^{32} chosen-plaintexts. For comparison, to reach balanced complexities on seven rounds from the original attack by using the classical data/time/memory trade-off, the amount of data needed will be approximately 2^{71} chosen plaintexts. Furthermore, we have been able to increase by one the number of rounds attacked with 2^{32} chosen-plaintexts for the three key length but with time and memory complexities very close to the natural bound of the exhaustive search. We also obtained competitive results in the very low data complexity league with, for instance, attacks on 8 rounds of AES-256 requiring only 2^8 chosen plaintexts.

7.3.2 Observation on the Keyschedules

Our attacks exploit the fact that most of the operations in the keyschedule algorithm are linear. The relation used are described in Property 19, extending the observations made Chapter 3 about the 128-bit version.

Property 19 (the key-schedule properties)**.** *Consider a sequence of consecutive subkeys* $k_r, k_{r+1}, \ldots$ *. We have the following useful relations between the equivalent subkeys* $u_r, u_{r+1}, \ldots$ *:*

- ***AES-128*** *:*
  - *1)* $u_r[4..7] = u_{r+1}[0..3] + u_{r+1}[4..7]$
  - *2)* $u_r[8..11] = u_{r+1}[4..7] + u_{r+1}[8..11]$
  - *3)* $u_r[12..15] = u_{r+1}[8..11] + u_{r+1}[12..15]$

- ***AES-192*** *:*
  1) $u_r[4..7] = u_{r+1}[8..11] + u_{r+1}[12..15]$
  2) $u_r[12..15] = u_{r+2}[0..3] + u_{r+2}[4..7]$
  3) *if r is even:*
     i) $u_r[0..3] = u_{r+1}[4..7] + u_{r+1}[8..11]$
     ii) *the knowledge of* $u_{r+1}[12..15]$ *and* $u_{r+2}[0..3]$ *allows to compute* $u_r[8..11]$
  4) *if r is odd:*
     i) $u_r[8..11] = u_{r+1}[12..15] + u_{r+2}[0..3]$
     ii) *the knowledge of* $u_{r+1}[4..7]$ *and* $u_{r+1}[8..11]$ *allows to compute* $u_r[0..3]$
- ***AES-256*** *:*
  1) $u_r[4..7] = u_{r+2}[0..3] + u_{r+2}[4..7]$,
  2) $u_r[8..11] = u_{r+2}[4..7] + u_{r+2}[8..11]$,
  3) $u_r[12..15] = u_{r+2}[8..11] + u_{r+2}[12..15]$.

7.3.3 Attack on Six-Round AES-128 with 2^8 chosen-plaintexts

If the data available is limited to 2^8 chosen-plaintexts, the best attack found is based on the attack depicted on Figure 7.6 and the meet-in-the-middle is performed on the equation

$$03.\Delta z_3[8] + \Delta z_3[9] = 07.\Delta x_4[8] + 07.\Delta x_4[9] + 02.\Delta x_4[11].$$

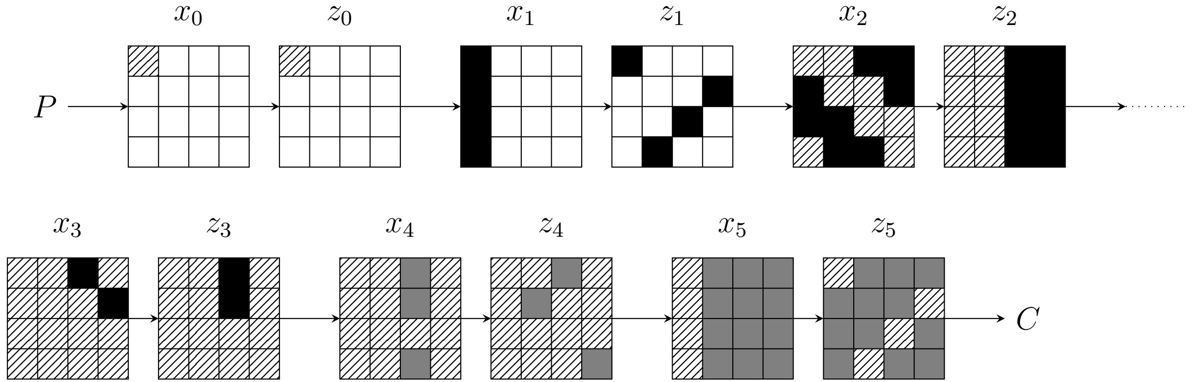


Figure 7.6: Attack on 6 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

The bytes of $\mathcal{B}_{off}$ are the first column of x_1, the two last columns of z_2, and bytes 8 and 9 of z_3. They can assume $2^{8\times 14}$ different values and so the memory requirement is $2^{112-0,66} = 2^{111,34}$ multisets on average according to the remark made in Section 7.2.1.

As the S-box is a bijection and as we consider a δ-set in which only one byte is active, we do not need to guess $x_0[0]$ in order to identify the corresponding set of 256 plaintexts to build the multiset. As a consequence, the bytes of $\mathcal{B}_{on}$ are the entire state

x_5 except the first column, and the third column of x_4 except byte 10. Thanks to the keyschedule equations, they can take only $2^{8\times 12}$ values instead of $2^{8\times 15}$ since we have the three equations $u_4[5] = u_5[1] + u_5[5]$, $u_4[8] = u_5[4] + u_5[8]$ and $u_4[15] = u_5[11] + u_5[15]$.

All in all this leads to the following attack where $e_{in} = 03.z_3[8] + z_3[9]$ and $e_{out} = 07.x_4[8] + 07.x_4[9] + 02.x_4[11]$:

- **Preprocessing phase:**
  1. Set $\Delta_i z_0[0]$ to i for $0 \leqslant i \leqslant 255$. Then $\Delta_i z_0$ is known since the other differences are null.
  2. Guess $x_1[0..3]$ (for one of the 256 messages) and use $\Delta_i z_0$ to compute $\Delta_i z_1[0]$, $\Delta_i z_1[7]$, $\Delta_i z_1[10]$ and $\Delta_i z_1[13]$. Then $\Delta_i z_1$ is known since the other differences are null.
  3. Guess bytes 1, 2, 6, 7, 8, 11, 12 and 13 of x_2. Use them with $\Delta_i z_1$ to compute $\Delta_i z_2[8..15]$.
  4. Guess $x_3[8]$ then compute $\Delta_i z_3[8]$ using $\Delta_i z_2[8..11]$.
  5. Guess $x_3[13]$ then compute $\Delta_i z_3[9]$ using $\Delta_i z_2[12..15]$.
  6. Compute the multiset $[\Delta_0 e_{in}, \ldots, \Delta_{255} e_{in}]$ and store it in a hash table (if it was not already in it).
- **Online phase:**
  1. Ask for a structure of 256 plaintexts such that byte 0 assume the 256 possible values and others bytes are constant.
  2. Choose one of them to be the one from which difference will be computed.
  3. Guess bytes 1, 2, 4, 5, 8, 11, 14 and 15 of u_5. Compute $u_4[5]$ and $u_4[8]$ and then partially decrypt the ciphertexts to obtain $\Delta_i x_4[8]$ and $\Delta_i x_4[9]$ for $0 \leqslant i \leqslant 255$.
  4. Guess bytes 3, 6 and 9 of u_5, and continue to partially decrypt the ciphertexts.
  5. Guess byte 12 of u_5. Compute $u_4[15]$ and then partially decrypt the ciphertexts to obtain $\Delta_i x_4[11]$.
  6. Build the multiset $[\Delta_0 e_{out}, \ldots, \Delta_{255} e_{out}]$ and check whether the multiset exists in the hash table. If not, discard the key guess.

Finally, the time complexity is equivalent to $2 \times 2^{-6} \times 2^8 \times 2^{96} = 2^{99}$ encryptions and the memory requirement is $2^{113{,}34}$ AES-blocks. The probability for a wrong guess to succeed is approximatively $2^{111{,}34} \times 2^{-467{,}6} = 2^{-356{,}26}$ and, as we try 2^{96} key guess, we expect that only the right value remains after the last step.

Trade-Off. Since the memory is higher than the time complexity, the data/time/memory trade-off presented Section 7.1.2 is possible. This leads to an attack using 2^8 chosen plaintexts (as the data is reused $2^{7{,}17}$ times), with a time complexity equivalent to $2^{106{,}17}$ encryptions and requiring $2^{106{,}17}$ 128-bit blocks.

Key Recovery. This attack retrieves the right value of u_5 except on bytes 0, 7, 10 and 13 and so can easily be turned into a key-recovery attack. The attacker guesses the four missing bytes of u_5 to retrieve the master key and try it. This step has a negligible complexity compared to the previous one.

7.3.4 Attack on Seven-Round AES-256 with 2^{16} chosen-plaintexts

The best attack on seven rounds AES−256 with 2^{16} chosen-plaintexts is depicted on Figure 7.7.

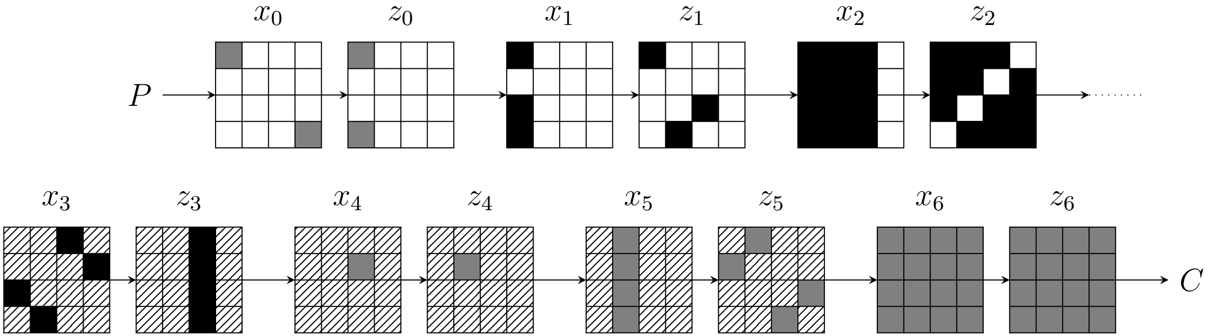


Figure 7.7: Attack on 7 AES rounds (key length : 256 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

The bytes of $\mathcal{B}_{off}$ are bytes 0,2 and 3 of x_1, the three first columns of x_2 and the third column of z_3. The bytes of $\mathcal{B}_{on}$ are bytes 0 and 15 of x_0, the entire state x_6, the second column of x_5 and byte 9 of x_4. The number of values assumed by the bytes of $\mathcal{B}_{on}$ is reduced by a factor 2^8 using the equation $u_4[5] = u_6[1] + u_6[5]$. The time complexity is equivalent to 2^{178} encryptions and the memory is $2^{153,34}$ AES-blocks.

Key Recovery. This attack can easily be turned into a key-recovery attack without increasing the complexity since only 12 key bytes are sufficient to recover the master key.

7.3.5 Attack on Seven-Round AES-192 with 2^{32} chosen-plaintexts

The best attack on seven rounds AES−192 with 2^{32} chosen-plaintexts is depicted on Figure 7.8.

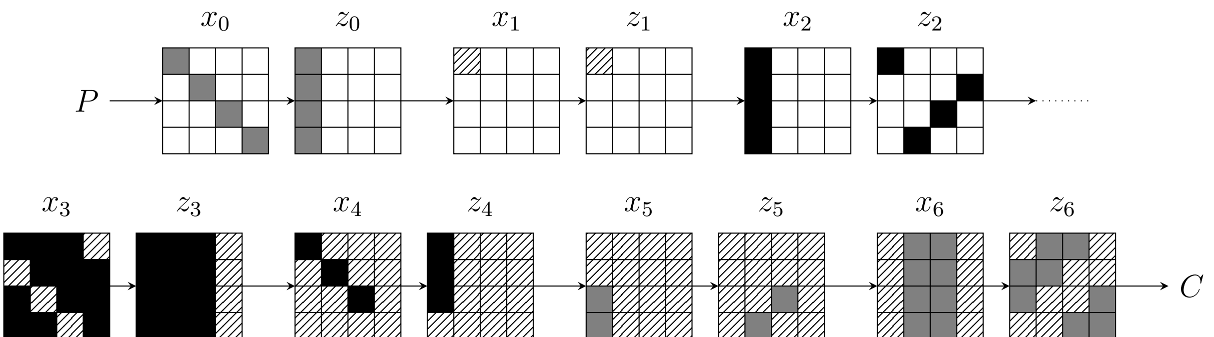


Figure 7.8: Attack on 7 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

The bytes of $\mathcal{B}_{off}$ are the first column of x_2, the three first columns of z_3, and bytes 0, 1 and 2 of z_4. The bytes of $\mathcal{B}_{on}$ are the first column of z_0, the second and third columns of x_6 and bytes 2 and 3 of x_5. Thanks to Property 19, we can reduce the number of possible

values assumed by them by a factor 2^8 since $u_5[7] = u_6[11] + u_6[15]$. The time complexity is equivalent to 2^{106} encryptions and the memory requirement is $2^{153,34}$ AES-blocks.

Trade-Off. Applying the classical data/time/memory trade-off leads to an attack using 2^{32} chosen plaintexts, with a time complexity equivalent to $2^{129,67}$ encryptions and a memory requirement of $2^{129,67}$ AES-blocks. Note that the data complexity remains 2^{32} because the structure may be divided into 2^{24} δ-sets and each of them may be reused $2^{7,34}$ times on average.

Key Recovery. This attack can easily be turned into a key-recovery attack without increasing the complexity since only 15 key bytes are sufficient to recover the master key.

7.4 An SPN-dedicated Tool

The attacks described in this chapter are somewhat generic and can be applied on AES-like block ciphers. More precisely, we can attack iterative block ciphers such that each round takes as input a state and a subkey, both represented by a vector of $\mathbf{F}_q^n$, and performs the three following operations successively on the state:

1. `SubBytes`: a bijective non-linear Sbox is applied on each of the n components of the state vector.
2. `MixColumns`: the state vector is multiplied by an $n \times n$ invertible matrix M_r with coefficients in the field $\mathbb{F}_q$.
3. `AddRoundKey`: the subkey vector is added to the state vector.

An additional `AddRoundKey` operation (using a whitening key) may be applied before the first round. Using the tool described Chapter 2 requires a keyschedule composed of AES-like equations.

Such a construction is very common for recent block ciphers as for instance SQUARE, AES, and LED [GPPR11]. In most of them, the same Sbox and the same sparse matrix are used each round to reach good performance but the generic attack presented works without such assumptions.

All in all, we built a tool taking as input such a block cipher and gives the best meet-in-the-middle attacks on it. The only difficult task is to find the minimal equations of the system $x = M_r \cdot y$ and for now we can only handle matrix composed of MDS submatrix.

7.5 Application to Low Data Complexity Attacks

The attack scheme can also be applied to find low data complexity attacks on AES-like block cipher. The tool described Chapter 2 is strictly more general than this one but exhausting all the attacks can be done in practical time and we were able to find results on more rounds than previously.

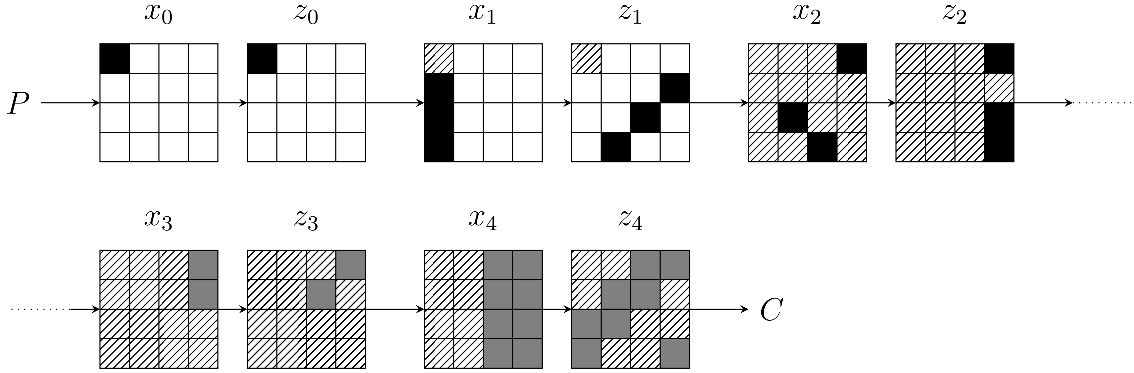


Figure 7.9: First part of the attack on 5 AES rounds. Black bytes are enumerated and stored in a hash table. Gray bytes are enumerated. Hatched bytes play no role. The differences are null in white squares

7.5.1 Attack on Five-Round AES-128 with Eight Chosen Plaintexts

The attack with eight chosen plaintexts is relatively easy and relies on three meet-in-the-middle on equations derived from the mixcolumn operation. The first meet-in-the-middle is depicted Figure 7.9.

The adversary starts by asking for a set of eight (different) chosen plaintexts which have a difference only on one byte. In the sequel we assume this is the byte 0, but similar attacks exist for each other position. Then he builds three hash tables (the indexes will be given later):

- Table 1: Guess $x_0[0]$, $x_1[1..3]$, $x_2[6]$, $x_2[11]$ and $x_2[12]$ for one message. Due to the structure of the AES and of considered plaintexts, we can deduce those bytes for all the eight messages by using the knowledge of the plaintexts. Store them in a hash table. It contains exactly $2^{7\times 8} = 2^{56}$ entries.
- Table 2: Guess the first column of x_4 and $x_3[15]$ for one message. As in the construction of the first table, we can deduce those bytes for all the eight messages but by using the knowledge of the ciphertexts. Store them in a hash table. It contains exactly $2^{5\times 8} = 2^{40}$ entries.
- Table 3: Guess the second column of x_4 and $x_3[14]$ for one message. As in the construction of the second table, we deduce those bytes for all the eight messages and store them in a hash table containing exactly $2^{5\times 8} = 2^{40}$ entries.

Now the attack is pretty simple:

1. Guess the two last columns of x_4 for one message.
2. As $u_4[9] = u_5[5] + u_5[9]$ and $u_4[12] = u_5[8] + u_5[12]$, we can deduce $x_3[12]$ and $x_3[13]$.
3. Deduce those bytes for the eight messages.
4. As the mixcolumn matrix is MDS, we have the following equation:

$$03\Delta S(x_2[6]) + \Delta S(x_2[11]) + 07\Delta S(x_2[12]) = 0e\Delta x_3[12] + 0b\Delta x_3[13].$$

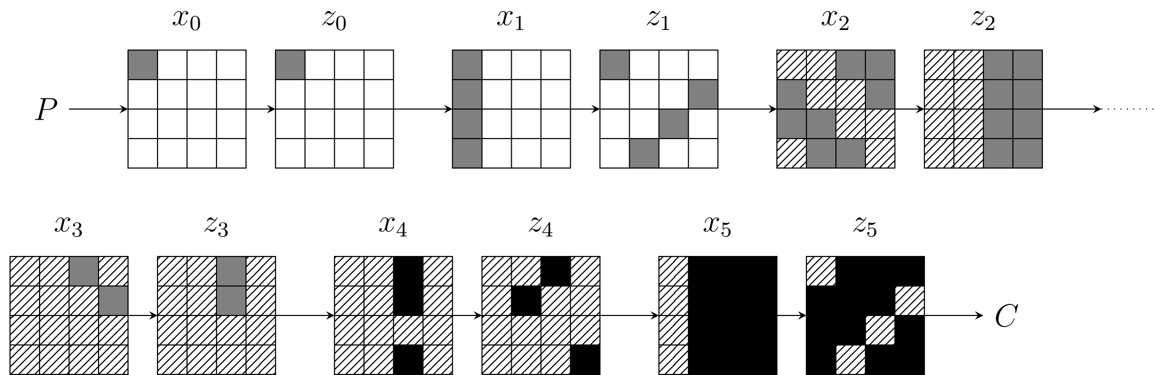


Figure 7.10: First part of the attack on 6 AES rounds. Black bytes are enumerated and stored in a hash table. Gray bytes are enumerated. Hatched bytes play no role. The differences are null in white squares

5. This leads to seven independent equations between the bytes of Table 1 and the actual known bytes. As a consequence we expect on average one entry of this hash table to match.
6. Then we get back the seven differences in $S(x_2[6])$ and $S(x_2[11])$.
7. We can now perform a check on the two other hash tables using these equations:

$$07\Delta x_3[14] = 07\Delta x_3[12] + \Delta x_3[13] + 02\Delta S(x_2[6]) + 03\Delta S(x_2[11]),$$

$$07\Delta x_3[15] = 07\Delta x_3[12] + 03\Delta x_3[13] + \Delta S(x_2[6]) + 02\Delta S(x_2[11]).$$

8. Finally we get back the value of missing bytes in x_4 allowing us to compute the master key and check its correctness.

Note that the time complexity is 2^{64} if and only if the last mixcolumn operation is not omitted. Unless, the keyschedule equations used to deduce $x_3[12]$ and $x_3[13]$ do not hold and the time complexity is 2^{80}.

7.5.2 Attack on Six-Round AES-128 with Thirteen Chosen Plaintexts

This attack is similar at the attack on five rounds and relies on two meet-in-the-middle on equations derived from the mixcolumn operation. The first meet-in-the-middle is depicted Figure 7.10.

The adversary starts by asking for a set of thirteen chosen plaintexts which have a difference only on one byte. In the sequel we assume this is the byte 0, but similar attacks exist for each other position. Then he builds two hash tables (the indexes will be given later):

- Table 1: Guess the three last columns of x_5 for one message. Since $u_5[5] = u_6[1] + u_6[5]$, $u_5[8] = u_6[4] + u_6[8]$ and $u_5[15] = u_6[11] + u_6[15]$, we deduce $x_4[8]$, $x_4[9]$ and $x_4[11]$ respectively. Due to the structure of the AES, we can deduce those bytes for

all the thirteen messages by using the knowledge of the ciphertexts. Store them in a hash table. It contains exactly $2^{12\times 8} = 2^{96}$ entries.

- Table 2: Guess the first column of x_5 and $x_4[10]$ for one message. As in the construction of the first table, we deduce those bytes for all messages and store them in a hash table containing exactly $2^{5\times 8} = 2^{40}$ entries.

Now the attack is pretty simple:

1. Guess $x_0[0]$, $x_1[0..3]$, $z_2[8..15]$, $x_3[8]$ and $x_3[13]$ for one message.
2. Deduce those bytes for the thirteen messages.
3. As the mixcolumn matrix is MDS, we have the following equation:

$$03\Delta S(x_3[8]) + \Delta S(x_3[13]) = 07\Delta x_4[8] + 07\Delta x_4[9] + 02\Delta x_4[11].$$

4. This leads to twelve independent equations between the bytes of Table 1 and the actual known bytes. As a consequence we expect on average one entry of this hash table to match.
5. Then we get back the twelve differences in $x_4[8]$ and $x_4[9]$, and the value of the three last columns of x_5 for one message.
6. We can now perform a check on the second hash table using this equation:

$$02\Delta x_4[10] = \Delta S(x_3[8]) + \Delta S(x_3[13]) + 03\Delta x_4[8] + \Delta x_4[9].$$

7. Finally we get back the value of missing bytes in x_5 allowing us to compute the master key and check its correctness.

Note that the memory complexity is 2^{96} if and only if the last mixcolumn operation is not omitted. Unless, the keyschedule equations used to deduce $x_4[8]$, $x_4[9]$ and $x_4[11]$ do not hold and the memory complexity is 2^{120}, also increasing the data complexity to sixteen chosen plaintexts.

7.A Multiset Representation

As there are about $\binom{2^8+2^8-1}{2^8} \approx 2^{506.17}$ multisets of 256 elements from $\mathbb{F}_{256}$, we are able to represent them on 512 bits. Here is one way of doing it for a given multiset M. In the sequel, we consider that $M = \{x_1^{n_1}, \ldots, x_m^{n_m}\}$, with $\sum_{i=1}^{m} n_i = 256$, that we may represent by

$$\underbrace{x_1\, x_1\, x_1\, x_1}_{n_1} \Big| \underbrace{x_2\, x_2\, x_2}_{n_2} \Big| \ldots \Big| \underbrace{x_m\, x_m\, x_m\, x_m\, x_m}_{n_m}, \tag{7.1}$$

where the distinct elements are the m elements x_i, which appears each with multiplicity n_i. In M, the order of the elements is undetermined.

Consider the set $S = \{x_1, \ldots, x_m\}$ deduced from M by deleting any repetition of element in M. As there are at most 256 elements in S, we can encode whether $e \in \mathbb{F}_{256}$

belongs to S in a 256-bit number s by a 1-bit flag at the position e seen as an index in $[0, \ldots, 255]$ in s. Then, to express the repetition of element, we sort M using the natural order in the integers and consider the sequence of multiplicity of each distinct element: if $x_1 < \cdots < x_m$, then we consider the sequence $n_1, \ldots, n_m$. We use a second 256-bit number t to store the sequence of $(\sum_{j=1}^{i} n_j)_i$ seen as indexes in t, which actually encodes the positions of the vertical separators in the multiset representation (7.1). The 512-bit element (s, t) then represents the multiset M.

7.B More Optimal Attacks

In this section we describe some attacks minimizing the maximum between the time complexity (counted in AES encryption) and the memory complexity (counted in 128-bit block). For each of them we assume that the last MixColumns is performed.

7.B.1 Attack on Six-Round AES-192 with 2^8 Chosen Plaintexts

The best attack found on six rounds of AES-192 is similar to the attack on the 128-bit version but we mount it on an other column because of the key-schedule. It is depicted on Figure 7.11, its time complexity is 2^{106} encryptions and the memory requirement $2^{113,34}$ AES-blocks. Indeed, we can reduce the number of values taken by bytes of $\mathcal{B}_{on}$ by using the two following equations derived from the key-schedule: $u_4[0] = u_5[4] + u_5[8]$ and $u_4[7] = u_5[11] + u_5[15]$. Thus they can assume only $2^{8\times 13}$ values.

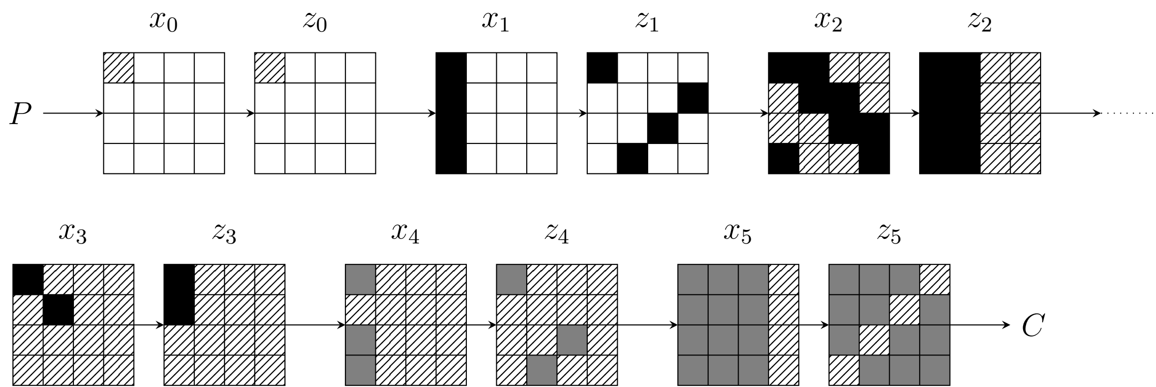


Figure 7.11: Attack on 6 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

Finally, applying the tradeoff leads to an attack using 2^8 chosen plaintexts, with a time complexity equivalent to $2^{109,67}$ encryptions and a memory requirement of $2^{109,67}$ AES-blocks. It can be easily turned into a key-recovery attack without increasing the complexity since only 11 key bytes are needed to recover the master key.

7.B.2 Attack on Six-Round AES-256 with 2^8 Chosen Plaintexts

The six rounds attack on AES−128 described section 7.3.3 can be applied on the AES−256. But in that case u_4 and u_5 are independent so the time complexity is increased by a factor of $2^{8\times3}$. This leads to an attack using 2^8 chosen plaintexts, with a time complexity equivalent to 2^{122} encryptions and a memory requirement of $2^{113{,}34}$ AES-blocks.

Key Recovery. The attacker can guess the 17 missing bytes of u_4 and u_5 to retrieve the master key but this increases the time complexity to 2^{136}. Instead, we can apply the six rounds attack on AES−192 described section 7.5.2. As u_4 and u_5 are independent, the time complexity is increased by a factor of $2^{8\times2}$ but in an other hand $u_5[4..11]$ are already known so the time complexity is decreased by a factor of $2^{8\times8}$. Then the attacker can perform an exhaustive search in order to retrieve the master key. All in all, this leads to a key recovery attack using 2^8 chosen plaintexts, with a time complexity equivalent to 2^{122} encryptions and a memory requirement of $2^{114{,}34}$ AES-blocks.

7.B.3 Attack on Seven-Round AES-192 with 2^8 Chosen Plaintexts

The best attack found using 2^8 chosen plaintexts is depicted on Figure 7.12 and has a time complexity equivalent to 2^{163} encryptions and a memory requirement of $2^{153{,}34}$ AES-blocks. Indeed, the bytes of $\mathcal{B}_{on}$ assume only $2^{8\times20}$ values instead of $2^{8\times26}$ because $u_5[0..3]$ can be computed from $u_6[4..11]$, and we have $u_5[4..5] = u_6[8..9] + u_6[12..13]$, $u_4[7] = u_5[11] + u_5[15]$ and $u_4[10] = u_5[14] + u_6[2]$.

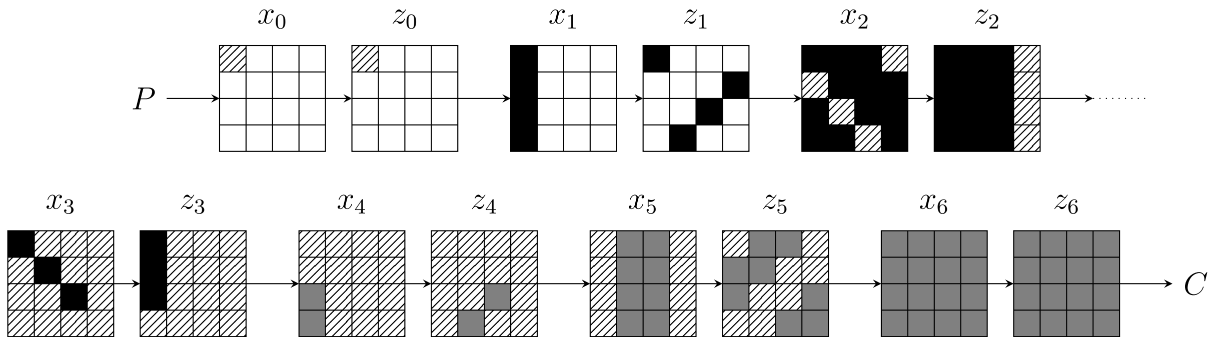


Figure 7.12: Attack on 7 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

Finally, this attack can be easily turned into a key-recovery attack without increasing the complexity since only 4 key bytes are needed to recover the master key.

7.B.4 Attack on Seven-Round AES-256 with 2^8 Chosen Plaintexts

The best attack found with 2^8 chosen plaintexts is based on the attack depicted on Figure 7.13 which has a time complexity equivalent to 2^{163} encryptions and a memory requirement of $2^{193{,}34}$ AES-blocks. Indeed, since u_5 and u_6 are independent we may expect to reduce the number of possible values of $\mathcal{B}_{on}$ by a factor 2^8 only, and this is done by

using the equation $u_4[13] = u_6[9] + u_6[13]$. It is depicted on and is based on the equation

$$\alpha_1 \Delta z_3[0] + \alpha_2 \Delta z_3[1] + \alpha_3 \Delta z_3[2] + \alpha_4 \Delta z_3[3] = \beta_1 \Delta x_4[1].$$

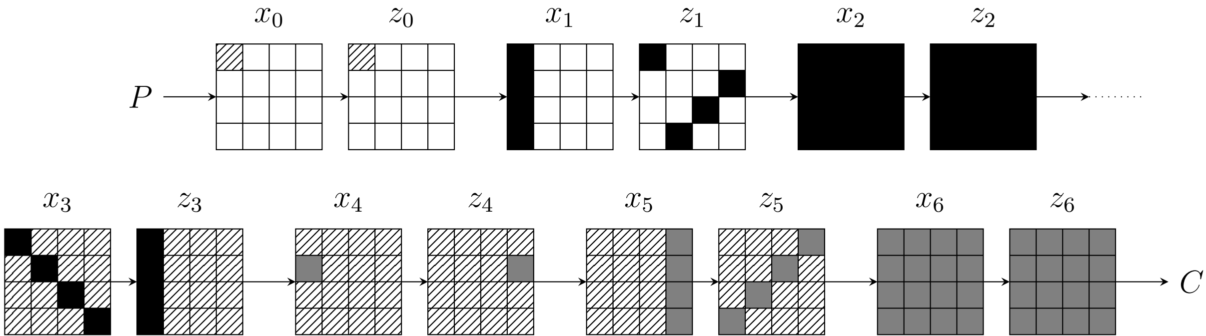


Figure 7.13: Attack on 7 AES rounds (key length : 256 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

Finally, applying the trade-off leads to an attack using 2^8 chosen plaintexts, with a time complexity equivalent to $2^{170,34}$ encryptions and a memory requirement of 2^{186} AES-blocks. If we allow more data it is possible to reach a complexity of $2^{15,84}$ chosen plaintexts, $2^{178,17}$ encryptions and $2^{178,17}$ AES-blocks, to compare to the attack with 2^{16} chosen plaintexts described Section 7.3.4. It can be easily turned into a key-recovery attack without increasing the complexity since only 12 key bytes are needed to recover the master key.

7.B.5 Attack on Eight-Round AES-256 with 2^8 Chosen Plaintexts

The best attack found with 2^8 chosen plaintexts is based on the attack depicted on Figure 7.14 which has a time complexity equivalent to 2^{227} encryptions and a memory requirement of $2^{241,34}$ AES-blocks. Indeed, since u_5 and u_6 are independent we may expect to reduce the number of possible values of $\mathcal{B}_{on}$ by a factor $2^{3\times 8}$ only, and this is reached by using the equations $u_4[7] = u_6[3] + u_6[7]$, $u_4[10] = u_6[6] + u_6[10]$ and $u_4[13] = u_6[9] + u_6[13]$.

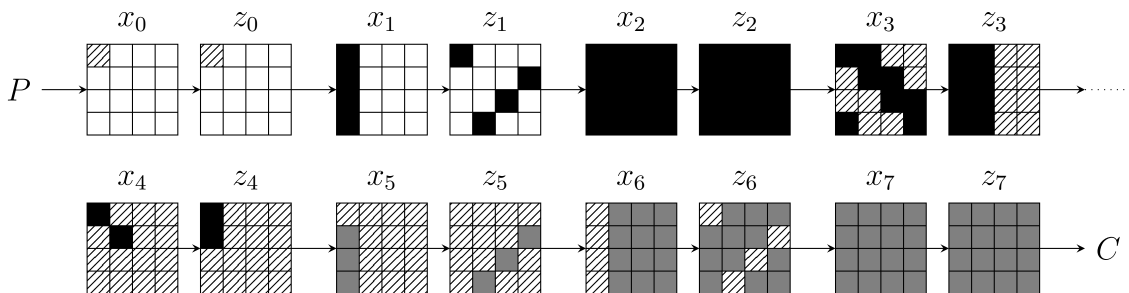


Figure 7.14: Attack on 8 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

Finally, applying the trade-off leads to an attack using 2^8 chosen plaintexts, with a time complexity equivalent to $2^{234,17}$ encryptions and a memory requirement of $2^{234,17}$ AES-blocks. It can be easily turned into a key-recovery attack without increasing the complexity since only 4 key bytes are needed to recover the master key.

7.B.6 Attack on Seven-Round AES-128 with 2^{32} Chosen Plaintexts

The best attack we found is based on the attack depicted on Figure 7.15. The bytes of $\mathcal{B}_{off}$ are the first column of x_2, the entire state z_3 excepted the second column, and bytes 8, 9 and 10 of z_4. They can take $2^{8\times 19}$ different values and so the memory requirement is $4 \times 2^{152-0,66} = 2^{153,34}$ AES-blocks on average.

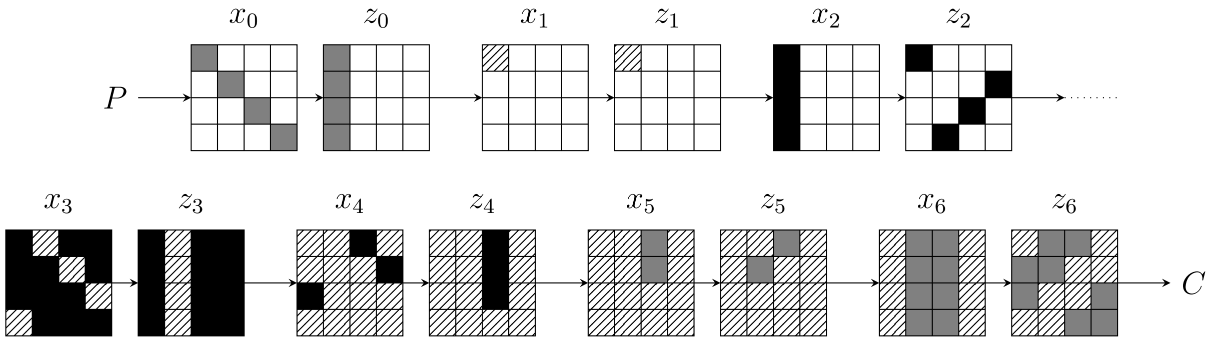


Figure 7.15: Attack on 7 AES rounds (key length : 128 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

The bytes of $\mathcal{B}_{on}$ are the first column of z_0, the second and third columns of x_6, and bytes 8 and 9 of x_5. Thanks to the key-schedule equations, they can assume only $2^{8\times 8}$ values instead of $2^{8\times 10}$. Indeed we have the two equations $u_5[5] = u_6[1] + u_6[5]$ and $u_5[8] = u_6[4] + u_6[8]$.

All in all this leads to the following attack:

- **Preprocessing phase:**
  1. Let consider a δ-set encryption and set $\Delta_i z_1[0]$ to i for $0 \leqslant i \leqslant 255$. Then $\Delta_i z_1$ is known since the other differences are null.
  2. Guess $x_2[0..3]$ and use $\Delta_i z_1$ to compute $\Delta_i z_2[0]$, $\Delta_i z_2[7]$, $\Delta_i z_2[10]$ and $\Delta_i z_2[13]$. Then $\Delta_i z_2$ is known since the other differences are null.
  3. Guess the state z_3 excepted the second column. Use them with $\Delta_i z_2$ to compute $\Delta_i z_3[0..3]$ and $\Delta_i z_3[8..15]$.
  4. Guess $x_4[2]$ then compute $\Delta_i z_4[10]$ using $\Delta_i z_3[0..3]$.
  5. Guess $x_4[8]$ then compute $\Delta_i z_4[8]$ using $\Delta_i z_3[8..11]$.
  6. Guess $x_4[13]$ then compute $\Delta_i z_4[9]$ using $\Delta_i z_3[12..15]$.
  7. Compute the multiset and store it in a hash table (if it was not already in it).
- **Online phase:**
  1. Ask for a structure of 2^{32} chosen plaintexts such that bytes 0, 5, 10 and 15 can take the 2^{32} possible values and the remaining bytes are constant.

2. Guess bytes 0, 5, 10 and 15 of k_{-1}.
3. Choose a δ-set.
4. Guess $u_5[5]$ and bytes 1, 4, 11 and 14 of u_6. Then partially decrypt the ciphertexts to obtain $\Delta_i x_5[9]$ for $0 \leqslant i \leqslant 255$.
5. Guess bytes 2, 8 and 15 of u_6. Compute $u_6[5]$ and $u_5[8]$ and then partially decrypt the ciphertexts to obtain $\Delta_i x_5[8]$
6. Build the multiset and check whether the multiset exists in the hash table. If not, discard the key guess.

Finally, the time complexity is equivalent to $2^{32} \times 3 \times 2^{-6} \times 2^8 \times 2^{64} \approx 2^{99{,}6}$ encryptions and the memory requirement is $2^{153{,}34}$ AES-blocks. The probability for a wrong guess to succeed is approximatively $2^{151{,}34} \times 2^{-467{,}6} = 2^{-316{,}26}$ and, as we try 2^{96} key guess, we expect that only the right value remains after the last step.

Trade-Off. We note that the memory is higher than the time complexity and so the classical data/time/memory trade-off can be applied. This leads to an attack using 2^{32} chosen plaintexts, with a time complexity equivalent to $2^{126{,}47}$ encryptions and requiring $2^{126{,}47}$ 128-bit blocks.

Key Recovery. This attack retrieves the right value of eight bytes of u_6 and so can be easily turned into a key-recovery attack. The attacker guesses the missing eight bytes of u_6 to retrieve the master key and try it. This step has a negligible complexity compared to the previous one. However this attack is obviously slower than an exhaustive search and its interest is only theoretical.

7.B.7 Attack on Seven-Round AES-256 with 2^{32} Chosen Plaintexts

The seven rounds attack on AES−128 described section 7.5.2 can be applied on the AES−256. But in that case u_5 and u_6 are independent so the time complexity is increased by a factor of $2^{8\times 2}$. Thus, this leads to an attack using 2^{32} chosen plaintexts, with a time complexity equivalent to $2^{133{,}67}$ encryptions and a memory requirement of $2^{133{,}67}$ AES-blocks once the trade-off applied.

Key Recovery. The attacker can guess the 22 missing bytes of u_5 and u_6 to retrieve the master key but this increases the time complexity to 2^{172}. Instead, as we know the right value for the four bytes needed to identify a δ-set, two bytes of u_5 and columns 1 and 2 of u_6, we can apply variants of the Demirci and Selçuk attack with negligible complexity compared to $2^{133{,}67}$ in order to get more key material before to perform an exhaustive search on the remaining key bytes.

7.B.8 Attack on Eight-Round AES-192 with 2^{32} Chosen Plaintexts

The best attack found is based on the seven rounds attack described section 7.5.2 extended by one round at the beginning. Thus the memory requirement remains the same as $2^{193{,}34}$ AES-blocks. In another hand, we need to guess four more key bytes in the online phase and the keyschedule equations used have changed.

The bytes of $\mathcal{B}_{on}$ are the entire state x_7, the last column of x_6 and byte 1 of x_5. However, the knowledge of u_7 allows to compute $u_6[0..7]$, $u_5[12..15]$ and $k_{-1}[12..15]$, thanks to Property 19 and the *key bridging technique*. Thus the online phase can be performed as follows:

1. Ask for a structure of 2^{32} chosen plaintexts such that bytes 0, 5, 10 and 15 assume the 2^{32} possible values and the rest of the bytes are constant.
2. Guess the subkey u_7 and compute $k_{-1}[15]$.
3. Guess bytes 0, 5 and 10 of k_{-1} and then choose a δ-set.
4. Compute $u_6[3]$ and $u_6[6]$ and then partially decrypt the ciphertexts.
5. Guess $u_6[9]$ and $u_6[12]$ and then partially decrypt the ciphertexts.
6. Compute $u_5[13]$ and then partially decrypt the ciphertexts to obtain $\Delta_i x_5[9]$ for $0 \leqslant i \leqslant 255$.
7. Build the corresponding multiset and check whether it exists in the hash table. If not, discard the key guess.

All in all, the time complexity is equivalent to $2 \times 2^{8\times 25} \times 2^{-8\times 4} \times 2^{-6} \times 2^{8} = 2^{171}$ encryptions and the memory requirement is $2^{193,34}$ AES-blocks. Then we apply the data/time/memory trade-off to reach a complexity of 2^{32} chosen plaintexts, $2^{182,17}$ encryptions and $2^{182,17}$ AES-blocks. Finally, this attack can easily be turned into a key-recovery attack without increasing the complexity since only 8 key bytes are sufficient to recover the master key.

7.B.9 Attack on Eight-Round AES-256 with 2^{32} Chosen Plaintexts

The best attack found is the same as the 192-bit version. The only difference is that there is only one equation between the key bytes guessed during the online phase. Indeed, the only relation we have is $u_5[13] = u_7[9] + u_7[13]$. Thus the time complexity of this attack is 2^{195} encryptions and its memory requirement about $2^{193,34}$ AES-blocks. Finally, this attack can be easily turned into a key-recovery attack without increasing the complexity since only 12 key bytes are sufficient to recover the master key.

7.B.10 Attack on Nine-Round AES-256 with 2^{32} Chosen Plaintexts

We have been able to mount an attack on nine rounds depicted on Figure 7.16. Since u_7 and u_8 are independent we may expect to reduce the number of possible values of $\mathcal{B}_{on}$ by a factor of $2^{2\times 8}$ only, and this is reached by using the equations $u_6[10] = u_8[6] + u_6[10]$ and $u_6[13] = u_8[9] + u_8[13]$. As a consequence, its time complexity is equivalent to 2^{227} encryptions and the memory requirement is $2^{281,34}$ AES-blocks.

Finally, applying the data/time/memory trade-off leads to an attack using 2^{32} chosen plaintexts, with a time complexity equivalent to $2^{254,17}$ encryptions and a memory requirement of $2^{254,17}$ AES-blocks.

Key Recovery. First we note that about $2^{224} \times 2^{279,34} \times 2^{-467,6} = 2^{35,74}$ wrong values remain at the end of the attack. However, this attack can still be easily turned into a

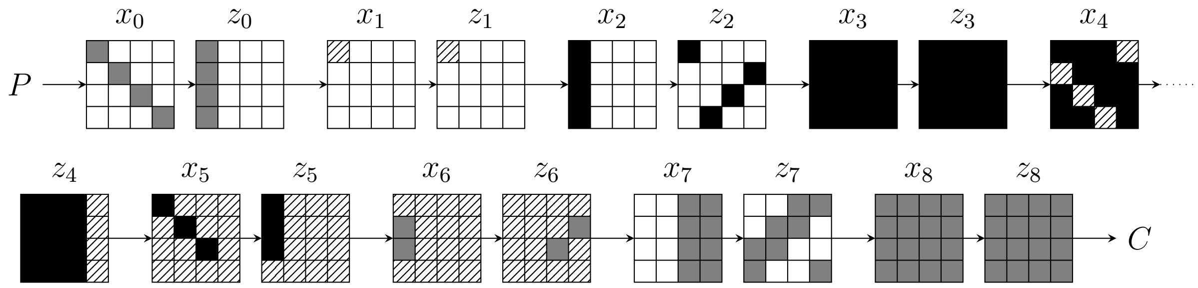


Figure 7.16: Attack on 9 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

key-recovery attack without increasing the complexity since the attacker has to guess only 8 key bytes to recover the master key.

Chapitre 8

The Differential Enumeration Technique

At Asiacrypt 2010, Dunkelman, Keller and Shamir develop many new ideas to solve the memory problems of the Demirci and Selçuk attacks. First of all, they show that instead of storing the whole sequence, we can only store the associated multiset, i.e. the unordered sequence with multiplicity rather than the ordered sequence. As it was explained in the previous chapter, this reduces the table by a factor 4 and avoids the need to guess one key byte during the attack. The second and main idea is the differential enumeration which allows to reduce the number of parameters that describes the set of functions from 24 to 16. However, to reduce this number, they rely on a special property on a truncated differential characteristic. The idea consists in using a differential truncated characteristic whose probability is not too small. The property of this characteristic is that the set of functions from one state to the state after 4 rounds can only take a restricted number of values, which is much smaller than the number of all functions. The direct consequence is an increase of the amount of needed data, but the memory requirement is reduced to 2^{128} and the same analysis also applies to the 128-bit version. However, this attack is not better than the impossible differential attack even though many trade-off could be used.

Dunkelman, Keller and Shamir's Attack. In [DKS10a], a new attack is developed using ideas from differential and meet-in-the-middle attacks. In the first stage, differential attacks find a differential characteristic with high or low probability covering many rounds. Then, in the online stage, the adversary asks for the encryption of many pairs: for each pair, the adversary tries to decrypt by guessing the last subkey and if the differential characteristic is followed, then the adversary increases the counter of the associated subkey. If the probability of the characteristic is high enough, then the counter corresponding to the right secret-key would be among the higher counters. In some case, it is also possible to add some rounds at the beginning by guessing part of the first subkeys.

Here, Dunkelman et al. propose a novel differential attack. Instead of increasing a counter once a pair is found, the adversary uses another test to eliminate the wrong guesses of the first or last subkeys. This test decides with probability one whether the middle rounds are covered with the differential. The idea is that the middle rounds follow

a part of the differential and the function f that associates each byte of the input state to one byte of the output state can be stored efficiently. Demirci and Selçuk propose to store in a table the function with no differential characteristic, which turns out to be much larger that this one. Consequently, in Dunkelman et al.'s attack, the adversary guesses the first and last subkeys and looks for a pair that follows the beginning and last rounds of the differential characteristic. Once such a pair is found, the adversary takes one of the messages that follows the characteristic and constructs a structure to encrypt which is related to a δ-set for the intermediate rounds. From the encryption of this set, the adversary can decrypt the last rounds and check whether the encryption of this δ-set belongs to the table. If this is the case, then the part of the first and last subkeys are correct and an exhaustive search on the other parts of the key allows to find the whole key.

To construct the table, the idea is similar to the attack. We need to find a pair of messages that satisfies the truncated differential characteristic. Then, we take one message in the pair and we compute the function f. Dunkelman et al. use a rebound technique to find the pair that follows the characteristic.

Our Results. Dunkelman et al. show that by using a particular 4-round differential characteristic with a not too small probability, the active states in the middle of the characteristic can only take 2^{64} values. In their characteristic, they also need to consider the same 8 key bytes as Demirci and Selçuk. They claim that *"In order to reduce the size of the precomputed table, we would like to choose the δ-set such that several of these parameters will equal to predetermined constants. Of course, the key bytes are not known to the adversary and thus cannot be "replaced" by such constants"*. Here, we show that it is possible to enumerate the whole set of solutions more efficiently than by taking all the values for the key bytes such that every value of these bytes are possible. We show that the whole set can take only 2^{80} values with this efficient enumeration technique. Of course, it might be possible to improve this result to 2^{64} but not any further since the key bytes may take all the 2^{64} possible values. Using the same ideas, we show that it is possible to have an efficient enumeration for a 5-round differential characteristic which allows us to mount an attack on 9 rounds for `AES-256`. The bottleneck of the attack is no longer the memory, but the time and data complexities.

In this chapter, we show that the number of parameters describing the functions can be further reduced to 10 and that this attack is now more efficient than the impossible differential attack [LDKK08]. We describe the best key-recovery attacks on 7 rounds of all versions of `AES` with all complexities below 2^{100}, as the related-key attack of Biryukov and Nikolič in [BN10]. We also show improved key-recovery attacks on 8 rounds of `AES-192` and on 8 and 9 rounds of `AES-256`. To this end, we use several tradeoffs proposed by Dunkelman et al. and we use a more careful analysis of the enumeration technique. Those results have been published in [DFJ13] and in [DF13].

8.1 Unified View of Previously Known MITM Attacks on AES

In this section, we present a unified view of the previously known meet-in-the-middle (MITM) attacks on AES[GM00, DS08, DKS10a], where n rounds of the block cipher can be split into three consecutive parts of n_1, n_2 and n_3 rounds, $n = n_1 + n_2 + n_3$, such that a particular set of messages may verify a certain property that we denote $\star$ in the sequel in the middle n_2 rounds (Figure 8.1).

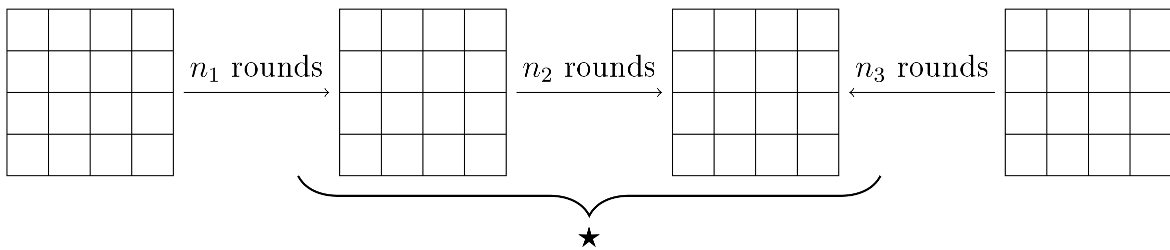


Figure 8.1: General scheme of the meet-in-the-middle attack on AES, where some messages in the middle rounds may verify a certain $\star$ property used to perform the meet-in-the-middle.

The general attack uses three successive steps:

Precomputation phase

1. In this phase, we build a lookup table T containing all the possible sequences constructed from a δ-set such that one message verifies the $\star$ property.

Online phase

2. Then, in the online phase, we need to identify a δ-set containing a message m verifying the desired property.
3. Finally, we partially decrypt the associated δ-set through the last n_3 rounds and check whether it belongs to T.

The two steps of the online phase require to guess some key bytes while the goal of this attack is to filter some of their values. In the best case, only the right ones should pass the test.

Demirci and Selçuk Attack.

The starting point is to consider the set of functions

$$f : \{0,1\}^8 \to \{0,1\}^8$$

that maps a byte of a δ-set to another byte of the state after four AES rounds. A convenient way is to view f as an ordered byte sequence $(f(0), \ldots, f(255))$ so that it can be represented by 256 bytes. The crucial observation made by the generalizing Gilbert and Minier attack is that this set is tiny since it can be described using 25 byte-parameters

($2^{25 \cdot 8} = 2^{200}$) compared with the set of all functions of this type which counts as many as $2^{8 \cdot 2^8} = 2^{2048}$ elements. Considering the differences $(f(0) - f(0), f(1) - f(0), \ldots, f(255) - f(0))$ rather than values, the set of functions can be described by 24 parameters. Dunkelman et al. identify these parameters as follows:
- the full state x_3 of message 0,
- four bytes of state x_2 of message 0,
- four bytes of subkey k_3.

The four bytes of the state x_2 only depend on the column of z_1 where the active byte of the δ-set is located; for instance, if it is column 0, then those bytes are $x_2[0, 1, 2, 3]$. Similarly, the four bytes of k_3 depend on the column of x_5 where the byte we want to determine is located; as an example, if it is column 0, then those bytes are $k_3[0, 5, 10, 15]$.

In their attacks [DS08], Demirci and Selçuk use the $\star$ property that does not filter any message. Consequently, they do not require to identify a particular message m. The data complexity of their basic attack is very small and around 2^{32}. However, since there is no particular property, the size of the table T is very large and the basic attack only works for the `AES-256`. To mount an attack on the `AES-192`, they consider some time/memory tradeoff. More precisely, the table T does not contain all the possible states, but only a fraction α. Consequently, a specific δ-set may not be in the table T, so that we have to wait for this event and redo the attack $O(1/\alpha)$ times on average. The attack becomes probabilistic and the memory requirement makes the attack possible for `AES-192`. The consequence of this advanced version of the attack, which also works for `AES-256`, is that the amount of data increases a lot. The time and memory requirement of the precomputation phase is due to the construction of table T that contains messages for the $n_2 = 4$ middle rounds, which counts as many as $2^{8 \cdot 24} = 2^{192}$ ordered sequences of 256 bytes.

Finally, it is possible to remove from each function some output values. Since we know that these functions can be described by the key of 24 or 32 bytes, one can reduce T by a factor 10 or 8 by storing only the first differences. Such an observation has been used by Wei et al. in [WLH11].

Dunkelman et al. Attack.

In [DKS10a], Dunkelman, Keller and Shamir introduced two new improvements to further reduce the memory complexity of [DS08]. The first one uses multisets, behaving as unordered sequences, and the authors show that there is still enough information so that the attack succeeds. The second improvement uses a particular 4-round differential characteristic (Figure 8.2) to reduce the size of the precomputed lookup table T, at the expense of trying more pairs of messages to expect at least one to conform to the truncated characteristic.

The main idea of the differential characteristic is to fix the values of as many state-bytes as possible to a constant. Assume now we have a message m such that we have a pair (m, m') that satisfies the whole 7-round differential characteristic and our goal is to recover the key. Contrary to classical differential attacks, where the adversary guesses some bytes of the last subkey and eliminates the wrong guess, the smart idea of Dunkelman et al. is

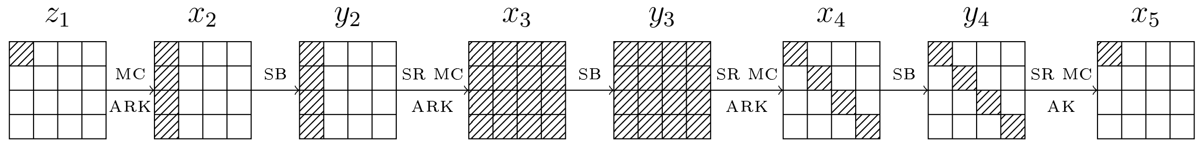


Figure 8.2: The four middle rounds used in the 7-round attack from [DKS10a]. Dashed bytes are active, others inactive.

to use a table to recover the right key more efficiently. Usually, differential attacks do not use memory to recover the key or to find the right pair. The attack principle consists in constructing the δ-set from m which can be made since we already have to guess some key bytes to check if the pair (m, m') has followed the right differential characteristic. Then, the table allows to identify the right key from the encryption of the δ-set.

It is now easy to see that the differential characteristic can be described using only 16 bytes. The states x_3 and y_3 can only take 2^{32} possible differences each, so that the number of solutions for these two states is 2^{64}. We also have the 4 key-bytes of u_2 and the 4 key-bytes of k_3 corresponding to the active bytes of Figure 8.2 in states z_2 and x_4.

Table 8.1 shows the best cryptanalysis of AES variants, including our new results detailed in this chapter.

8.2 New Attack on AES

In this section, we describe our basic attack on AES, which is independent of the key schedule algorithms. We begin in Section 8.2.1 by describing an efficient way to enumerate and store all the possible multisets in the middle that are used to mount the meet-in-the-middle attack. We continue in Section 8.2.2 by applying the general scheme previously described to construct a key-recovery attack on all AES versions reduced to 7 rounds. Finally, in Section 8.2.3, we show that modifying slightly the property for the middle rounds allows to trade some memory for data and time.

8.2.1 Efficient Tabulation

As in the previous results, our attack also uses a large memory lookup table constructed in the precomputation phase, and used in the online phase. Dunkelman, Keller and Shamir showed that if a message m belongs to a pair of states conforming to the truncated differential characteristic of Figure 8.2, then the multiset of differences $\Delta x_5[0]$ obtained from the $\delta-$set constructed from m in x_1 can only take 2^{128} values, because 16 of the 24 parameters used to build the multisets can take only 2^{64} values instead of 2^{128}. We make the following proposition that reduces the size of the table by a factor 2^{48}.

Property 20. *If a message m belongs to a pair of states conforming to the truncated differential characteristic of Figure 8.2, then the multiset of differences $\Delta x_5[0]$ obtained from the $\delta-$set constructed from m in x_1 can only take 2^{80} values. More precisely, the 24 parameters (which are state bytes of m) can take only 2^{80} values in that case. Conversely,*

for each of these 2^{80} values there exists a tuple (m, m', k) such that m is set to the chosen value and, the pair (m, m') follows the truncated characteristic.

Proof. The proof uses rebound-like arguments borrowed from the hash function cryptanalysis domain [MRST09]. Let (m, m') be a right pair. We show in the following how the knowledge of 10 particular bytes restricts the values of the 24 parameters used to construct the multisets, namely:

$$x_2[0,1,2,3],\ x_3[0,\ldots,15],\ x_4[0,5,10,15]. \tag{8.1}$$

In the sequel, we use the state names mentioned in Figure 8.3. The 10 bytes

$$\Delta z_1[0],\ x_2[0,1,2,3],\quad \Delta w_4[0],\ z_4[0,1,2,3]. \tag{8.2}$$

can take as many as 2^{80} possible values, and for each of them, we can determine the values of all the differences shown on Figure 8.3: linearly in x_2, applying the SBox to reach y_2, linearly for x_3 and similarly in the other direction starting from z_4. By the differential

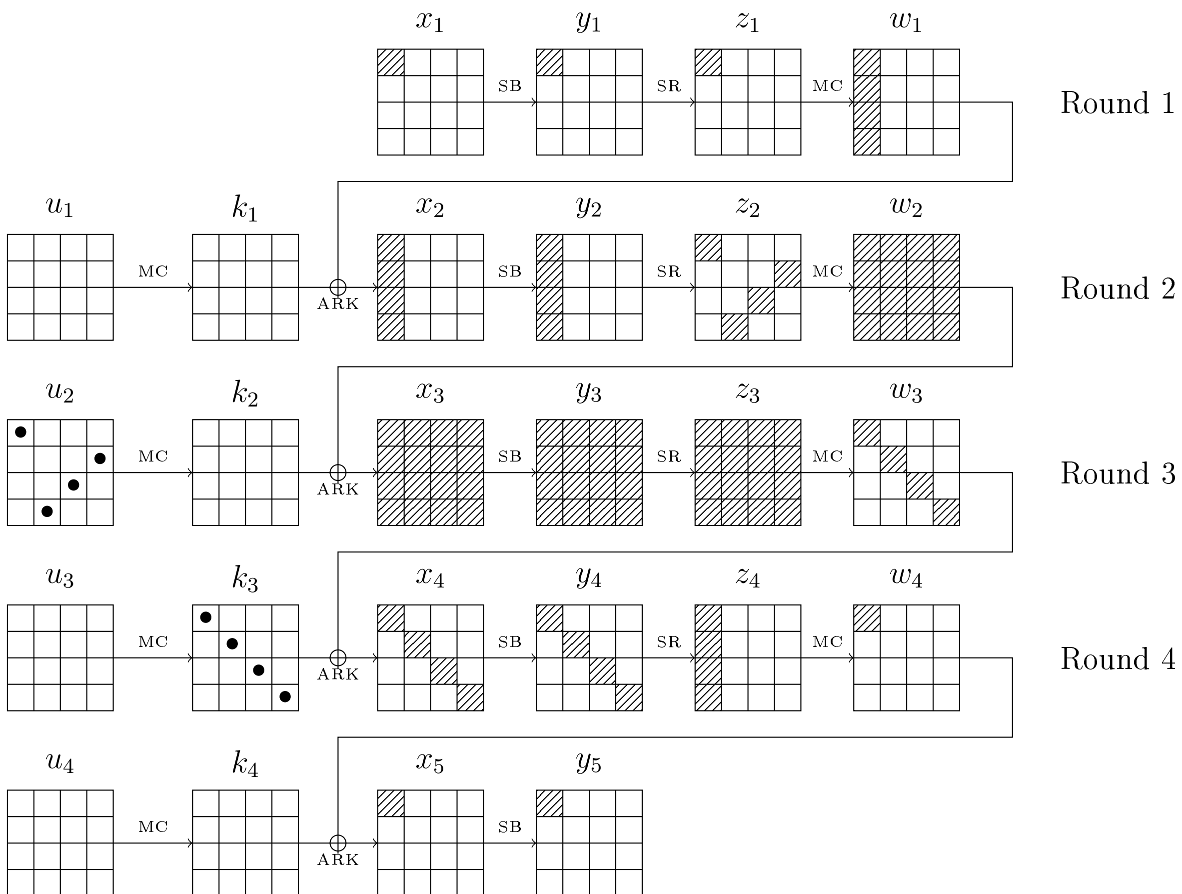


Figure 8.3: Truncated differential characteristic used in the middle of the 7-round attacks on AES. A hatched byte denotes a non-zero difference, whereas a while cell has no difference.

property of the AES SBox (Property 12), we get on average one value for each of the

16 bytes of state x_3[6]. From the known values around the two AddRoundKey layers of rounds 3 and 4, this suggests four bytes of the equivalent subkey $u_2 = \mathsf{MC}^{-1}(k_2)$ and four others in subkey k_3: those are $u_2[0]$, $u_2[7]$, $u_2[10]$, $u_2[13]$ and $k_3[0]$, $k_3[5]$, $k_3[10]$, $k_3[15]$; they are marked by a bullet (•) in Figure 8.3.

The converse is now trivial: the only difficulty is to prove that for each value of the 8 key bytes, there exists a corresponding master key. This actually gives a chosen-key distinguisher for 7 rounds of AES, as it has been done in Chapter 6.

To construct the multiset for each of the 2^{80} possible choice for the 10 bytes from (8.2), we consider all the $2^8 - 1$ possible values for the difference $\Delta y_1[0]$, and propagate them until x_5. This leads to a multiset of $2^8 - 1$ differences in $\Delta x_5[0]$. Finally, as the AES SBox behaves as a permutation over $\mathbb{F}_{256}$, the sequence in $\Delta y_1[0]$ allows to derive the sequence in $\Delta x_1[0]$. Note that in the present case where there is a single byte of difference between m and m' in the state x_1, both messages belongs to the same δ-set. This does not hold if we consider more active bytes as we will see in Section 8.3. We describe in an algorithmic manner this proof in Algorithm 10 of Appendix 8.A(ConstructTable). □

8.2.2 Simple Attack

Precomputation phase.

In the precomputation phase of the attack, we build the lookup table that contains the 2^{80} multisets for difference Δx_5 by following the proof of Property 20. This step is performed by first iterating on the 2^{80} possible values for the 10 bytes of (8.2) and for each of them, we deduce the possible values of the 24 original parameters. Then, for each of them, we construct the multiset of $2^8 - 1$ differences. Using the differential property of the AES Sbox, we can count exactly the number of multisets that are computed:

$$2^{80} \times \left(4 \times \frac{2^8 - 1}{(2^8 - 1)^2} + 2 \times \frac{(2^8 - 1)(2^7 - 1 - 1)}{(2^8 - 1)^2}\right)^{16} \approx 2^{80.09}. \quad (8.3)$$

Finally, the lookup table of the $2^{80.09}$ possible multisets that we simplify to 2^{80} requires about 2^{82} 128-bit blocks to be stored. To construct the table, we have to perform 2^{80} partial encryptions on 256 messages, which we estimate to be equivalent to 2^{84} encryptions.

Online phase.

The online phase splits into three parts: the first one finds pairs of messages that conform to the truncated differential characteristic of Figure 8.13, which embeds the previous 4-round characteristic in the middle rounds. The second step uses the found pairs to create a δ-set and test them against the precomputed table and retrieve the secret key in a final phase.

To generate one pair of messages conforming to the 7-full-round characteristic where there are only four active bytes in both the plaintext and the ciphertext differences,

6. In fact, only 2^{64} values of the 10 bytes lead to a solution for x_3 but for each value, there are 2^{16} solutions for x_3.

we prepare a structure of 2^{32} plaintexts where the diagonal takes all the possible 2^{32} values, and the remaining 12 bytes are fixed to some constants. Hence, each of the $2^{32} \times (2^{32}-1)/2 \approx 2^{63}$ pairs we can generate satisfies the plaintext difference. Among the 2^{63} corresponding ciphertext pairs, we expect $2^{63} \cdot 2^{-96} = 2^{-33}$ to verify the truncated difference pattern. Finding *one* such pair then requires 2^{33} structures of 2^{32} messages and $2^{32+33} = 2^{65}$ encryptions under the secret key. Using this secret key, the probability that the whole truncated characteristic of Figure 8.13 is verified is $2^{-2\times3\times8} = 2^{-48}$ because of the two $4 \to 1$ transitions in the `MixColumns` of rounds 0 and 5. By repeating the previous procedure to find 2^{48} pairs, one is expected to verify the full 7-round characteristic. All in all, we ask the encryptions of $2^{48+65} = 2^{113}$ messages to find 2^{48} pairs of messages. Note that we do not have to examine each pair in order to find the right one. Indeed, if a pair verifies the full 7-round characteristic, then the ciphertext difference has only four active bytes. Thus, we can store the structures in a hash table indexed by the 12 inactive bytes to get the right pairs in average time of one.

For each of the 2^{48} pairs, we get $2^{8\times(8-2\times3)} \cdot 2^8 = 2^{24}$ suggestions for the 9 key bytes:

$$k_{-1}[0,5,10,15],\ u_5[0],\ u_6[0,7,10,13]. \tag{8.4}$$

Indeed, there are 2^8 possibilities for the bytes from k_{-1} since the pair of diagonals in x_0 need to be active only in w_0 after the `MixColumns` operation. Among the 2^{32} possible values for those bytes, only $2^{32} \times 2^{-24} = 2^8$ verifies the truncated pattern. The same reasoning applies for $u_6[0,7,10,13]$, and the last byte $u_5[0]$ can take all the 2^8 values.

For all 2^{24} possibilities, we construct a δ-set to use the precomputed table. To do so, we partially decrypt the diagonal of one message, using the four known bytes from k_{-1} and consider the 2^8-1 possible non-zero differences for $\Delta x_1[0]$. This gives one set of 2^8 plaintexts, whose corresponding ciphertexts may be partially decrypted using the four known bytes from u_6 and the one from u_5. Once decrypted, we can construct the multiset of differences for Δx_5 and check if it lies in the precomputed lookup table. If not, we can discard the subkey with certainty. On the other hand, the probability for a wrong guess to pass this test is smaller than $2^{80} \cdot 2^{-467.6} = 2^{-387.6}$ so, as we try $2^{48} \cdot 2^{24} = 2^{72}$ multisets, only the right subkey should verify the test. Note that the probability is $2^{-467.6}$ (and not $2^{-506.17}$) because the number of ordered sequences associated to a multiset is not constant.

We summarize the above description in the following Algorithm 9, where the initial call to the function ConstructTable$(0,0)$ constructs the lookup table for Δx_1 and Δx_5 both at position zero (Figure 8.3) and is defined in Appendix 8.A.

Algorithm 9: – A simple attack.

```
T_{0,0} ← ConstructTable(0,0)                          // Appendix 8.A ;
while true do                                          // 2^81 times on average
    Ask for a structure S of 2^32 plaintexts P_m where bytes in diagonals 0 assume all
    values;
    Empty a hash table T of list of plaintexts;
    forall the corresponding ciphertexts C_m do
        index ← MC^{-1}(C_m)[1,2,3,4,5,6,8,9,11,12,14,15];
        forall the P ∈ T[index] do
            Consider the pair (P, P_m)      // 2^{-33} pairs by structure on average;
            forall the k_{-1}[0,5,10,15] s.t. Δw_0[1,2,3] = 0 do      // 2^8 times on
            average
                Construct δ-set D from P    // The δ-set belongs to the structure;
                forall the u_6[0,7,10,13] s.t. Δz_5[1,2,3] = 0 do
                    Decrypt column 0 of x_6 for D;
                    forall the u_5[0] do                                // 2^8 times
                        Decrypt byte 0 of x_5 for D;
                        Construct multiset M of Δx_5;
                        if M ∈ T_{0,0} then return ExhaustiveSearch ();
                    end
                end
            end
        end
        T[index] ← T[index] ∪ {P_m};
    end
end
```

To evaluate the complexity of the online phase of the simple attack, we count the number of AES encryptions. First, we ask the encryption of 2^{113} chosen-plaintexts, so that the time complexity for that step is already 2^{113} encryptions. Then, for each of the 2^{48} found pairs, we perform 2^{24} partial encryptions/decryptions of a δ-set. We evaluate the time complexity of this part to $2^{48+24+8} \cdot 2^{-5} = 2^{75}$ encryptions since we can do the computations in a good ordering as shown in Algorithm 9. All in all, the time complexity is dominated by 2^{113} encryptions, the data complexity is 2^{113} chosen-plaintexts, and the memory complexity is 2^{82} since it requires to store 2^{80} multisets.

8.2.3 Efficient Attack: New Property ★

Unlike the previous attacks where the bottleneck complexity is the memory, our attack uses a smaller table which makes the time complexity to find the pairs the dominating one. Therefore, we would like to decrease the time spent in that phase. The natural idea is to find a new property ★ for the four middle rounds that can be checked more efficiently. To do so, we reuse the idea of Dunkelman et al. from [DKS10a], which adds an

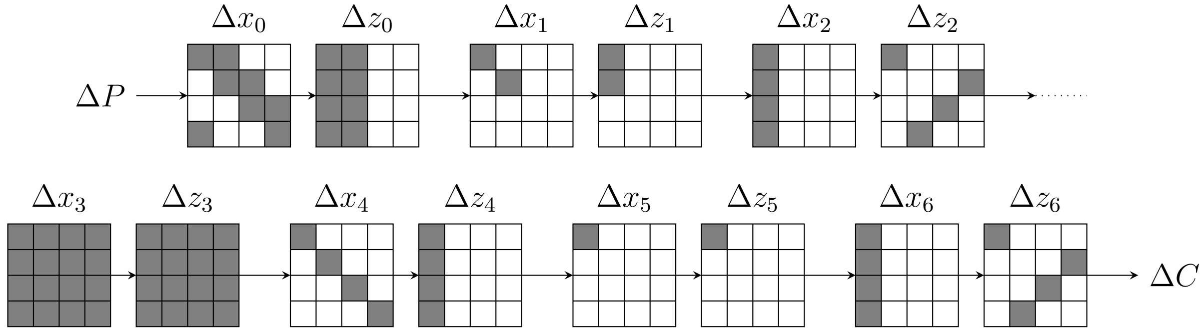


Figure 8.4: Example of a truncated differential characteristic used in the efficient attack on 7 rounds.

active byte in the second round of the differential characteristic. The sequence of active bytes becomes:

$$8 \xrightarrow{R_0} 2 \xrightarrow{R_1} 4 \xrightarrow{R_2} 16 \xrightarrow{R_3} 4 \xrightarrow{R_4} 1 \xrightarrow{R_5} 4 \xrightarrow{R_6} 16, \qquad (8.5)$$

with the constraint that the two active bytes of the second round belong to the same diagonal to be transformed in a column in the next round.

As a consequence, it is now easier to find pairs conforming to that truncated differential characteristic. Indeed, the size of the structures of plaintexts may take as many as 2^{64} different values, so that we can construct at most $2^{64} \cdot (2^{64} - 1)/2 = 2^{127}$ pairs from each structure. Therefore, it is enough to ask the encryption of $2^{8 \cdot 3 \cdot 3}/2^{127 - 8 \cdot 12} = 2^{41}$ structures to get 2^{72} pairs with the desired output difference pattern, and expect one to conform to the 7-round characteristic of Figure 8.4. Consequently in this new setting, we only need 2^{105} chosen plaintexts. In return, the number of pairs that the adversary has to consider is increased by a factor 2^{24} and so is the time complexity. Furthermore, we now need 11 parameters to generate the 24 parameters of the precomputed table, increasing the memory requirement by a factor 2^8. These parameters are the previous 10 ones and the difference in the second active byte of z_2. All in all, the time complexity of this attack is $2^{75+24} = 2^{99}$ encryptions, the data complexity is 2^{105} chosen plaintexts and the memory requirement is $2^{82+8} = 2^{90}$ 128-bit blocks.

Note that the time spent on one pair is the same for both the simple attack and the new one. Indeed, let K be the key bytes needed to construct the multiset. We suppose that we have a set of pairs such that one follows the differential. To find it, and incidentally some key-byte values, we proceed as follows: for each pair (m, m'), enumerate all possible values of K such that (m, m', K) have a non-zero probability to follow the differential. For each of them, construct the corresponding multiset from m or m'. If it belongs to the table, then we expect that it follows the differential characteristic since the table has been constructed that way. Otherwise, we know with probability 1 that either the pair (m, m') does not satisfy the characteristic, or the guessed value from K is wrong.

Assuming that the bytes of diagonals 0 and 2 of the structure of plaintexts takes all the values [7], the two differences in the first state of the second round can take four different

7. Those are bytes 0, 2, 5, 7, 8, 10, 13 and 15.

positions: $(0, 10)$, $(1, 11)$, $(2, 8)$ and $(3, 9)$. Similarly, the position of the active byte in the penultimate round is not constrained; it can be placed anywhere on the 16 positions. We can also consiser the opposite: one active byte at the beginning, and two active bytes in the end. These possibilities actually define tweaked versions of the property $\star$ and allows to trade some time for memory: with less data, we can check more tables for the same final probability of success. Namely, by storing $4 \times 16 + \binom{4}{2} \times 4 = 2^8$ tables to cover all the cases by adapting the proof of Property 20, the encryption of $2^{41}/2^8 = 2^{33}$ structures of 2^{64} plaintexts suffices to expect a hit in one of the 2^8 tables. Therefore, the memory complexity reaches 2^{98} AES blocks and the time complexity remains unchanged since we analyze 2^8 times less pairs, but the quantity of work to check *one* pair is multiplied by the same factor. We describe this efficient attack in an algorithmic manner in Appendix 8.B

8.2.4 Turning the distinguisher into a key recovery attack

In this section, we present an efficient way to turn this distinguisher into a key recovery attack. First, let us summarize what the adversary has in his possession at the end of the efficient attack: a pair (m, m') following the truncated differential characteristic, a δ-set containing m, the knowledge of 9 key bytes and the corresponding multiset for which we found a match in the precomputed table. Thus, there are still 2^{56}, 2^{120} or 2^{184} possible keys, if we consider AES-128, AES-192 or AES-256 respectively. As a consequence, performing an exhaustive search to find the missing key bytes would drastically increase the complexity of the whole attack, except for the 128-bit version. Even in that case, it seems nontrivial to recover the 2^{56} possible keys in less than 2^{96}, as the 9 key bytes do not belong to the same subkey.

A natural way to recover the missing bytes would be to replay the efficient attack by using different positions for the input and output differences. Unfortunately, this increases the complexity, and it would also interfere with the trade-off since we could not look for all the possible positions of the differences anymore.

We propose a method that recovers the two last subkeys in a negligible time compared to the 2^{99} encryptions of the efficient attack. First the adversary guesses the 11 parameters used to build the table of multisets, computes the value the corresponding 24 parameters and keeps the only one used to build the checked multiset. In particular, he obtains the value of the all intermediate state x_3 and one column of x_2. As a consequence, and for any position of the active byte of x_5, the Demerci and Selçuk original attack may be performed really quickly. Indeed, among the 9 (resp. 24) bytes to guess to perform the online (resp. offline) phase, at least 4 (resp. 20) are already known and the data needed is also in his possession. Finally, the adversary do this attack for each position of the active byte of x_5 and thus retrieves the two last subkeys.

8.3 Extension to More Rounds

8.3.1 Eight-Round Attacks on AES-192 and AES-256

We can extend the simple attack on the AES presented Section 8.2.2 to an 8-round attack for both 192- and 256-bit versions by adding one additional round at the end. This attack is schematized on Figure 8.5.

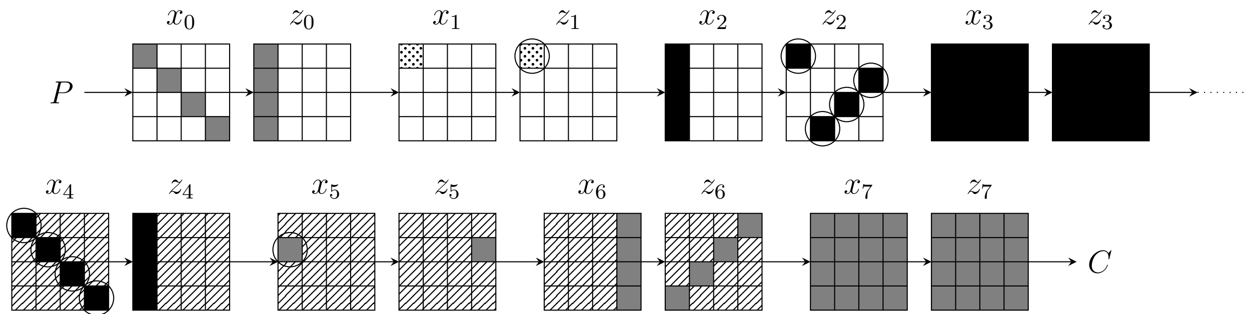


Figure 8.5: Scheme of the attack on 8 rounds. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes.

The main difficulty compared to the previous attack is that we cannot apply a first step to the structure to filter the wrong pairs. Indeed, now for each pair from the structure, there exists at least one key such that the pair follows the differential characteristic. Then our goal is to enumerate, for each pair and as fast as possible, the key bytes needed to identify a δ-set and construct the associated multiset assuming that the pair is a right one.

The main idea to do so is the following: if there is a single non-zero difference in a column of a state before (resp. after) the MixColumns operation, then the difference on same column in the state after (resp. before) can only assume $2^8 - 1$ values among all the $(2^8 - 1)^4$ possible ones. Combining this with the key schedule equations and to the differential property of the AES SBox (Property 12), this leads to an attack requiring 2^{113} chosen plaintexts, 2^{82} 128-bit blocks of storage and a time complexity equivalent to 2^{172} (resp. 2^{196}) encryptions on AES-192 (resp. AES-256).

To reach this time complexity, the position of the output active byte must be chosen carefully. The position of the input active byte for both the pair and the δ-set must be identical, as well as the output active byte of the pair and the byte that is to be checked. Then, the output difference must be located at position 1, 6, 11 or 12 in the case of AES-192. As for the AES-256, it can be located anywhere, except on bytes 0, 5, 10 and 15. Finally, in both cases, the position of the input difference does not matter.

Assume that the positions of the input and output active bytes are respectively 0 and 1. In the first stage of the attack, we ask for the encryption of 2^{81} structures of 2^{32} plaintexts. Then, the following procedure applied on each of the $2^{81} \cdot 2^{32+31} = 2^{144}$ pairs

allows to enumerate the 2^{24} possible values for the needed key bytes in about 2^{24} simple operations for the 192-bit version:

1. (a) Guess the difference in column 0 of x_0.
   (b) Deduce the actual values in this column.
   (c) Deduce bytes 0, 5, 10 and 15 of k_{-1}.
   (d) Store all these values in a hash table T_{-1} indexed by $k_{-1}[15]$.
2. Guess the difference in column 3 of x_6.
3. (a) Guess the difference in columns 0 and 1 of x_7.
   (b) Deduce the actual values of these two columns.
   (c) Deduce the actual values of $x_6[14]$ and $x_6[15]$.
   (d) Deduce $u_6[3]$, $u_6[6]$ and bytes 0, 1, 4, 7, 10, 11, 13 and 14 of k_7 (or u_7 if we do not omit the last `MixColumns`).
   (e) Store all these values in a hash table T_7.
4. (a) Similarly, guess the difference in the two other columns of x_7 and deduce $u_6[9]$, $u_6[12]$ and the 8 others bytes of the last subkey.
   (b) Retrieve $u_6[3]$, $u_6[6]$ and bytes 0, 1, 4, 7, 10, 11, 13 and 14 of k_7 (resp. u_7) using T_7 since $u_6[3]$ and $u_6[6]$ are linearly dependent of k_7 (and also of u_7).
   (c) Deduce $u_5[13]$ and $k_{-1}[15]$ from k_7.
   (d) Get bytes 0, 5 and 10 of k_{-1} using T_{-1}.

The fact we can deduce $u_5[13]$, $u_6[3]$, $u_6[6]$ comes from the following observation.

Property 21. *By the key schedule of* `AES-192`*, knowledge of the subkey k_7 allows to linearly deduce columns* 0 *and* 1 *of k_6 and column* 3 *of k_5.*

In contrast, to deduce $k_{-1}[15]$ from k_7, we need a more complicated observation made by Dunkelman et al. in [DKS10a].

Property 22 (Key bridging, [DKS10a])**.** *By the key schedule of* `AES-192`*, the knowledge of columns* 0, 1, 3 *of the subkey k_7 allows to deduce column* 3 *of the whitening key k_{-1}.*

Note that it is now easy to see why the choice of the input active byte does not affect the complexity and why only four positions for the output active byte lead to the minimal complexity.

Finally, for each of the 2^{144} pairs and for each of the 2^{24} subkeys corresponding to one pair, the adversary identifies the δ-set and verifies whether the corresponding multiset belongs to the precomputed table. Thus, the time complexity of this part is equivalent to $2^{144} \cdot 2^{24} \cdot 2^{8} \cdot 2^{-4} = 2^{172}$ encryptions.

In the case of the 256-bit version, k_6 and k_7 are independent and the only key schedule property we can use is the following one.

Property 23. *By the key schedule of* `AES-256`*, knowledge of the subkey k_7 allows to linearly deduce columns 1, 2 and 3 of k_5.*

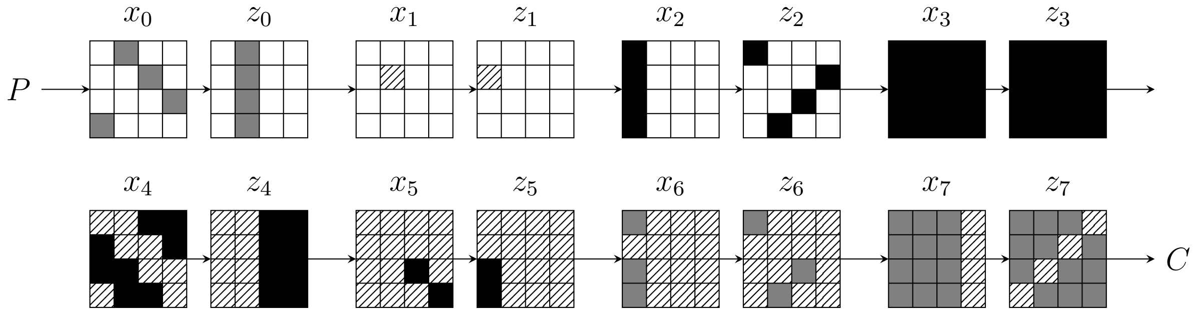


Figure 8.6: Attack on 8 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares

Then, there are 2^{48} possible values for the required key bytes and a procedure like the previous one enumerates them in 2^{48} simple operations.

It is possible to save some data in exchange for memory by considering several differentials in parallel. We can bypass the fact that all the positions for the output active byte does not lead in the same complexity by performing the check on y_5 instead of x_5. This is done by just adding one parameter to the precomputed table and increases its size by a factor 2^8. Then, we can look for all the $4 \cdot 16 = 2^6$ differentials in parallel on the same structure. All in all, the data complexity and the memory requirement become respectively 2^{107} chosen plaintexts and 2^{96} 128-bit blocks.

8.3.2 Improved Eight-Round Attack

In the previous chapter we described around 2^{16} variants of the Demirci-Selçuk attack and the differential enumeration technique can be applied on each of them. We present here the attack on 8 rounds minimizing the overall complexity, which begins by considering the attack on AES-192 depicted on Figure 8.6.

The bytes needed to perform the offline phase ($\mathcal{B}_{off}$) are the first column of x_2, the entire state x_3, the two last columns of z_4 and bytes 2 and 3 of z_5. The bytes used in the online phase ($\mathcal{B}_{on}$) are the second column of z_0, the three first columns of x_7, and the first column of x_6 excepted byte 1. Thanks to Property 19, they take only $2^{8\times17} = 2^{136}$ values because $u_6[0] = u_7[4] + u_7[8]$ and $u_6[7] = u_7[11] + u_7[15]$. Finally, the time complexity is equivalent to 2^{138} encryptions and the memory requirement is $2^{241{,}34}$ AES-blocks.

Differential Enumeration. The idea of Dunkelman *et al.* is to store in the hash table only the multisets built from a δ-set containing a message m that belongs to a pair (m, m') following a well-chosen differential path. In our case this is the truncated differential $4 \to 1 \to 4 \to 16 \to 8 \to 2 \to 3 \to 12$ depicted on Figure 8.7.

Then the bytes of $\mathcal{B}_{off}$ can take only $2^{16\times8}$ values for such a pair. Indeed, if we guess the differences in circled bytes then we obtain the difference before and after the S-box for each bytes of $\mathcal{B}_{off}$ and thus we can derive their absolute value thanks to Property 12. As a consequence, the memory requirement is decreased by a factor 2^{112}. However, we now need to find a pair that follows this truncated differential path and so the procedure of the online phase becomes:

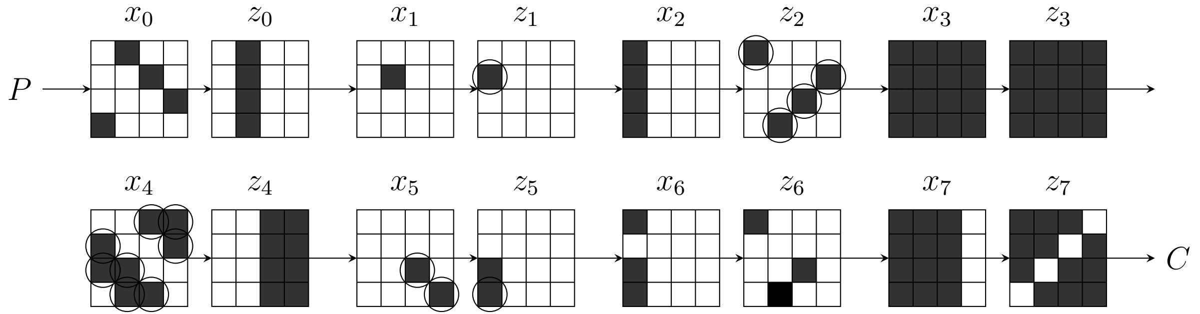


Figure 8.7: Differential characteristic on 8 AES rounds. The differences are null in white squares. The value of bytes of $\mathcal{B}_{off}$ can be derived from the differences in circled bytes.

1. Ask for a structure of 2^{32} plaintexts such that the second *diagonal* assume the 2^{32} possible values and others bytes are constant.
2. Store the corresponding ciphertexts in a hash table to identify the pairs that have a non-zero probability to follow the differential path.
3. For each of these pairs:
   (a) Guess $\Delta z_6[0]$, $\Delta z_6[7]$ and $\Delta z_6[10]$ and compute the difference in the three first columns of x_7.
   (b) Deduce the value of the three first columns of x_7 using Δz_7.
   (c) Deduce $u_6[0]$ and $u_6[7]$ using $u_7[4]$, $u_7[8]$, $u_7[11]$ and $u_7[15]$.
   (d) Deduce $z_6[0]$ and $z_6[7]$ and compute $\Delta x_6[0]$ and $\Delta x_6[3]$.
   (e) Check if the equation between $\Delta x_6[0]$ and $\Delta x_6[3]$ is satisfied.
   (f) Deduce $\Delta x_6[2]$ and then compute $x_6[2]$ using $\Delta z_6[10]$.
   (g) Guess $\Delta x_1[5]$ and compute the difference in the second column of z_0.
   (h) Deduce the value of the second column of z_0 using Δx_0.
   (i) Get the δ-set associated to one of the message of the pair and build the multiset from the corresponding ciphertexts.
   (j) Check whether the multiset exists in the hash table. If not, discard the key guess.
4. Restart with a new structure if no check found.

As each structure contains 2^{63} pairs and each of these pairs follows the differential with probability 2^{-144}, we need 2^{81} structures on average. Then, for each structure we have to study only $2^{63-32} = 2^{31}$ pairs and for each of them we have to perform $2^{24} \times 2^{8}$ partial encryptions that is equivalent to 2^{28} encryptions. All in all, this leads to an attack with 2^{113} chosen plaintexts, a time complexity equivalent to 2^{140} encryptions and a memory requirement of 2^{130} AES-blocks.

Reducing the data complexity. Note that for each possible choice of the active *diagonal* in the plaintext we found 96 attacks with the same complexity. As the corresponding differential paths are different it is possible to perform many attacks in parallel to save data in exchange of memory. For instance, if we use structure with three active *diagonals*,

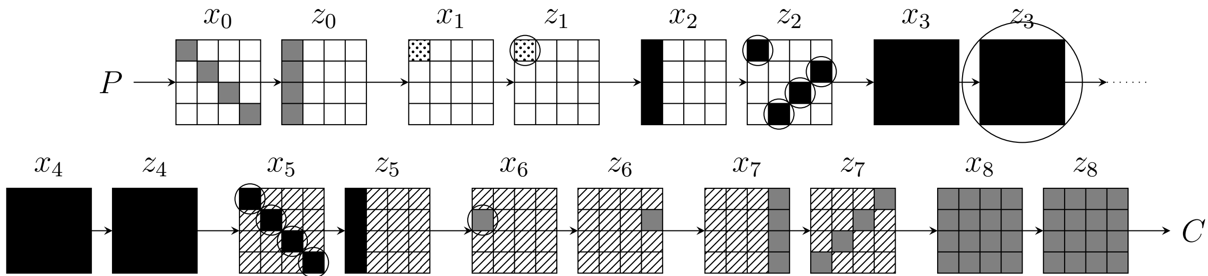


Figure 8.8: Scheme of the nine-round attack on AES-256. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes.

it is possible to reach a complexity of $2^{104,83}$ chosen plaintexts and $2^{138,17}$ AES-blocks, the time remaining unchanged.

Key Recovery. This attack can easily be turned into a key-recovery attack without increasing the complexity since only 9 key bytes are sufficient to recover the master key.

AES-256. This attack can be applied to the AES-256 excepted that the keyschedule does not allow us to reduce the time complexity anymore. This leads to an attack with 2^{113} chosen plaintexts, a time complexity equivalent to 2^{156} encryptions and a memory requirement of 2^{130} AES-blocks. For each possible choice of the active *diagonal* in the plaintext we found 384 attacks with the same complexity so it is possible to save more data than previously. For instance, if we use structure with three active *diagonals*, it is possible to reach a complexity of $2^{102,83}$ chosen plaintexts and $2^{140,17}$ AES-blocks, the time remaining unchanged.

8.3.3 Nine-Round Attack on AES-256

The 8-round attack on AES-256 described Section 8.3.1 can be extended to an attack on 9-round by adding one round right in the middle. This only increases the memory requirements: the time and data complexities remain unchanged. More precisely, the number of parameters needed to construct the precomputed table turns out to be $24+16 = 40$, but they can only assume $2^{8\times(10+16)} = 2^{208}$ different values. All in all, the data complexity of the attack stays at 2^{113} chosen-plaintexts, the time complexity remains 2^{196} encryptions and the memory requirement reaches about 2^{210} 128-bit blocks. To reduce its complexity, we can cover only 2^{-7} of the possible multisets stored in the precomputed table. In return, the data and time complexities are increased by a factor 2^7 by replaying the attack several times. This way, we reach the complexities mentioned in Table 8.1. This attack is schematized on Figure 8.8.

8.3.4 Nine-Round Attack on AES-192

A similar attack can be performed on AES-192 but it has to be mounted on other columns to enjoy keyschedule equations and the check is performed after the 7th SubBytes.

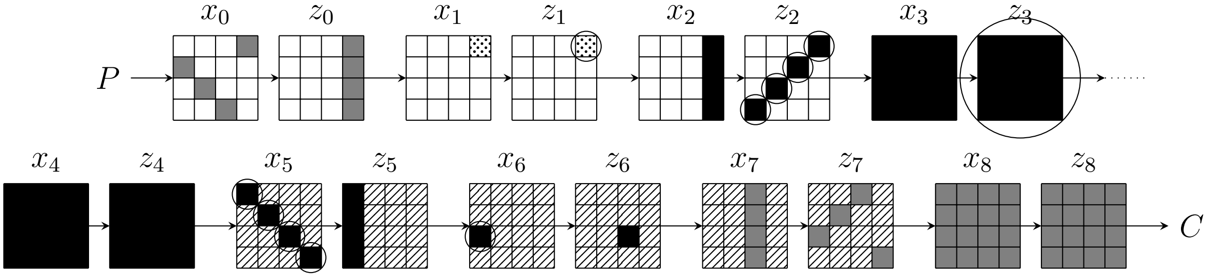


Figure 8.9: Scheme of the nine-round attack on AES-192. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes (helped by keyschedule).

This attack is depicted on Figure 8.9 and requires 2^{113} chosen plaintexts, a time complexity of approximately 2^{180} encryptions and 2^{194} 128-bit blocks of space. Using the classical data/time/memory trade-off we can reach an attack requiring 2^{120} chosen plaintexts and with a complexity of 2^{187} for both time and memory.

The bytes needed to compute the multisets can assume only 2^{216} values and lead to the knowledge of bytes 3, 6, 9 and 12 of u_3, the whole subkey k_4, bytes 0, 5, 10 and 15 of k_5 and byte 2 of k_6. However those 25 key bytes are related by the keyschedule of AES-192 and actually $u_3[3]$, $u_3[6]$ and $k_6[2]$ can be computed from the other ones, reducing the number of multisets stored by a factor 2^{24}. Furthermore, it is fairly easy to build those 2^{192} multisets in 2^{192} encryption-like operations.

Actually we forgot to run our tool on 9-round AES-192 and then did not publish this attack. Furthermore, Li, Jia and Wang recently described a slightly better one in [LJW13].

8.4 Conclusion

In this chapter, we have provided improved cryptanalysis of reduced round variants of all the AES versions in the standard single-key model, where the adversary wants to recover the secret key. In particular, we present an attack on 7-round of all AES versions that runs in less than 2^{100} encryptions of chosen-plaintexts. To the best of our knowledge, this is currently the most efficient result on AES-128 in this model. Additionally, we show we can turn this algorithm into attacks for AES-192 and AES-256 on 8 rounds, in time equivalent to 2^{140} and 2^{156} encryptions respectively, and we even reach an attack on 9 rounds of AES-256 in about 2^{203} encryptions.

Those results fit into the scheme on both differential and meet-in-the-middle attacks, which have been extensively studied in the past. More precisely, our algorithms improve on known techniques by drastically reducing the memory requirements so that the overall bottleneck switches from memory complexity in the previous meet-in-the-middle attacks to time or data complexity in our case.

We described the attacks leading to minimal overall complexity but, as in the previous chapter, we have exhausted the almost 2^{16} variants to find the best attacks. Some of our

Last mixcolumn performed (rows: number of guess in the online phase; columns: number of guess in the offline phase)

	10	11	12	13	14	15
7						8.0
8		8.0			7.0	7.0
9	8.0					
10						
11		9.6				
12		10.6				4.0
13		14.0				6.0
14		10.2			4.0	6.3
15		12.2			6.0	7.5

Last mixcolumn omitted (rows: number of guess in the online phase; columns: number of guess in the offline phase)

	10	11	12	13	14	15
7						8.0
8		8.0			7.0	7.0
9	8.0					
10						
11						
12		9.6				
13		14.1			0.0	4.0
14		7.2			4.0	6.4
15		11.0			6.1	7.9

Figure 8.10: Differential Enumeration: results on 7 rounds AES-128. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity.

results are summarized on Figures 8.10, 8.11 and 8.12. Best cryptanalytic results on AES in the single-key model are given reported Table 8.1.

Limitations. To save more data, Dunkelman *et al.* propose to consider differential paths with a bigger probability. We have exhausted the simple case where the new differential paths do not have active new bytes in the middle rounds. However, we did not try interesting cases where the active bytes of the pair and bytes of $\mathcal{B}_{on}$ and $\mathcal{B}_{off}$ are *desynchronized* since, besides the number of cases to handle, the complexity of our tweaked tool tends to explode as we cannot apply it to the keyschedule only.

As those complexities remain merely theoretical and also because the AES provides a good security margin, the block cipher is not threatened. Nevertheless, we believe the strategy behind these algorithms may pave the way for new cryptanalysis techniques.

number of guess in the offline phase

number of guess in the online phase	10	11	12	13	14	15	16	17	18	19	20	21	22
14												11.1	9.9
15											9.2	9.9	
16													
17											6.2	7.2	10.6
18							8.6				8.3	10.4	13.9
19							10.3	10.5		6.2	8.0	8.5	8.5
20					4.2	7.7	13.4	10.0		7.6	8.6	10.0	8.5
21	6.6	8.0			5.2	6.5	10.9	8.4		6.9	6.9	10.1	
22	7.3	11.0			3.3	4.3	12.2					8.0	
23		13.3					11.9						

last mixcolumn omitted

number of guess in the offline phase

number of guess in the online phase	10	11	12	13	14	15	16	17	18	19	20	21	22
12												6.0	
13												7.0	
14												10.9	9.9
15											9.2	9.9	
16											6.6	7.6	11.0
17							8.6				6.6	7.6	11.0
18							9.6				7.6	10.2	13.6
19							10.7	10.5		6.2	8.0	8.5	8.5
20					4.2	7.7	13.3	10.0		7.6	8.6	10.0	8.5
21	6.6	8.0			5.2	6.5	10.9	8.4		6.9	6.9	10.1	
22	7.3	10.8			3.3	4.3	12.2					8.0	
23		13.4					11.9						

last mixcolumn performed

Figure 8.11: Differential Enumeration: results on 8 rounds AES-192. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity.

number of guess in the offline phase

number of guess in the online phase

	21	22	23	24	25	26	27	28	29	30	31
19										9.5	10.3
20										9.6	9.6
21											
22											
23										9.6	10.4
24						8.0	11.2			7.6	7.6
25						6.0	14.0				
26											
27							12.2				
28	9.8	7.6					12.2			7.0	7.0
29	10.1	11.9								8.6	8.6
30	9.9	12.7					12.5				
31	6.8	12.9					11.8				

last mixcolumn omitted

number of guess in the offline phase

number of guess in the online phase

	21	22	23	24	25	26	27	28	29	30	31
19										9.5	10.3
20										9.6	9.6
21											
22											
23										9.6	10.4
24						8.0	7.6			7.6	7.6
25						6.0	14.2				
26											
27							12.2				
28	9.8	7.6					12.2			7.0	7.0
29	10.1	8.3								8.6	8.6
30	9.9	13.3					12.5				
31	6.8	12.9					11.8				

last mixcolumn performed

Figure 8.12: Differential Enumeration: results on 9 rounds AES-256. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity.

Table 8.1: Best cryptanalytic results on reduced AES variants in the secret-key model.

Version	Rounds	Data (CP)	Time	Memory	Technique	Reference
128	7	$2^{90.4}$	$2^{117.2}$ MA	2^{106}	ID	[MDRMH10]
	7	2^{116}	2^{116}	2^{116}	MITM	[DKS10a]
	7	$\mathbf{2^{105}}$	$\mathbf{2^{99}}$	$\mathbf{2^{90}}$	**MITM**	**Section 8.2**
	7	$\mathbf{2^{97}}$	$\mathbf{2^{99}}$	$\mathbf{2^{98}}$	**MITM**	**Section 8.2.3**
	8	2^{88}	$2^{125.3}$	2^{8}	Bicliques	[BKR11]
	10 (full)	2^{88}	$2^{126.2}$	2^{8}	Bicliques	[BKR11]
192	7	2^{116}	2^{116}	2^{116}	MITM	[DKS10a]
	7	$\mathbf{2^{8}}$	$\mathbf{2^{163}}$	$\mathbf{2^{153.34}}$	**MITM**	**Appendix 7.B.3**
	7	$\mathbf{2^{32}}$	$\mathbf{2^{129.67}}$	$\mathbf{2^{129.67}}$	**MITM**	**Section 7.3.5**
	7	$19 \cdot 2^{32}$	2^{155}	$19 \cdot 2^{32}$	Square	[FKL+00]
	7	$2^{91.2}$	2^{101}	$2^{139.2}$	ID	[LDKK08]
	7	2^{95}	2^{143}	2^{143}	MITM	[DS08]
	7	$\mathbf{2^{105}}$	$\mathbf{2^{99}}$	$\mathbf{2^{90}}$	**MITM**	**Section 8.2**
	7	$\mathbf{2^{97}}$	$\mathbf{2^{99}}$	$\mathbf{2^{98}}$	**MITM**	**Section 8.2.3**
	8	$\mathbf{2^{32}}$	$\mathbf{2^{182.17}}$	$\mathbf{2^{182.17}}$	**MITM**	**Appendix 7.B.8**
	8	2^{41}	$2^{187.63}$	2^{186}	MITM	[WLH11]
	8	$\mathbf{2^{104.83}}$	$\mathbf{2^{140}}$	$\mathbf{2^{138.17}}$	**MITM**	**Section 8.3.2**
	8	$\mathbf{2^{107}}$	$\mathbf{2^{172}}$	$\mathbf{2^{96}}$	**MITM**	**Section 8.3.1**
	8	$\mathbf{2^{113}}$	$\mathbf{2^{140}}$	$\mathbf{2^{130}}$	**MITM**	**Section 8.3.2**
	8	2^{113}	2^{172}	2^{129}	MITM	[DKS10a]
	8	$\mathbf{2^{113}}$	$\mathbf{2^{172}}$	$\mathbf{2^{82}}$	**MITM**	**Section 8.3.1**
	9	2^{80}	$2^{188.8}$	2^{8}	Bicliques	[BKR11]
	12 (full)	2^{80}	$2^{189.4}$	2^{8}	Bicliques	[BKR11]
256	**7**	$\mathbf{2^{8}}$	$\mathbf{2^{170.34}}$	$\mathbf{2^{186}}$	**MITM**	**Appendix 7.B.4**
	7	$\mathbf{2^{16}}$	$\mathbf{2^{178}}$	$\mathbf{2^{153.34}}$	**MITM**	**Section 7.3.4**
	7	$\mathbf{2^{32}}$	$\mathbf{2^{133.67}}$	$\mathbf{2^{133.67}}$	**MITM**	**Appendix 7.B.7**
	7	$21 \cdot 2^{32}$	2^{172}	$21 \cdot 2^{32}$	Square	[FKL+00]
	7	2^{95}	2^{143}	2^{143}	MITM	[DS08]
	7	2^{116}	2^{116}	2^{116}	MITM	[DKS10a]
	7	$\mathbf{2^{105}}$	$\mathbf{2^{99}}$	$\mathbf{2^{90}}$	**MITM**	**Section 8.2**
	7	$\mathbf{2^{97}}$	$\mathbf{2^{99}}$	$\mathbf{2^{98}}$	**MITM**	**Section 8.2.3**
	8	$\mathbf{2^{8}}$	$\mathbf{2^{234.17}}$	$\mathbf{2^{234.17}}$	**MITM**	**Appendix 7.B.5**
	8	$\mathbf{2^{32}}$	$\mathbf{2^{195}}$	$\mathbf{2^{193.34}}$	**MITM**	**Appendix 7.B.9**
	8	$2^{34.2}$	$2^{205.8}$	$2^{205.8}$	MITM	[DS08]
	8	$\mathbf{2^{102.83}}$	$\mathbf{2^{156}}$	$\mathbf{2^{140.17}}$	**MITM**	**Section 8.3.2**
	8	$\mathbf{2^{107}}$	$\mathbf{2^{196}}$	$\mathbf{2^{96}}$	**MITM**	**Section 8.3.1**
	8	$\mathbf{2^{113}}$	$\mathbf{2^{156}}$	$\mathbf{2^{130}}$	**MITM**	**Section 8.3.2**
	8	2^{113}	2^{196}	2^{129}	MITM	[DKS10a]
	8	$\mathbf{2^{113}}$	$\mathbf{2^{196}}$	$\mathbf{2^{82}}$	**MITM**	**Section 8.3.1**
	9	$\mathbf{2^{32}}$	$\mathbf{2^{254.17}}$	$\mathbf{2^{254.17}}$	**MITM**	**Appendix 7.B.10**
	9	$\mathbf{2^{120}}$	$\mathbf{2^{203}}$	$\mathbf{2^{203}}$	**MITM**	**Section 8.3.3**
	9	2^{120}	$2^{251.9}$	2^{8}	Bicliques	[BKR11]
	14 (full)	2^{40}	$2^{254.4}$	2^{8}	Bicliques	[BKR11]

CP: Chosen-plaintext. ID: Impossible Differential. MITM: Meet-in-the-Middle.

8.A Construction of the Tables

Algorithm 10: Construction of the table used in the simple attack.

```
Function ConstructTable(i, j)
    b_i ← i − 4(i mod 4) mod 16   // Retrieving the right positions;
    c_i ← ⌊b_i/4⌋                 // because of the ShiftRows;
    c_j ← ⌊j/4⌋;
    Empty a lookup table T;
    foreach value of Δz_1[b_i], x_2[4c_i], x_2[4c_i + 1], x_2[4c_i + 2], x_2[4c_i + 3] do
        Deduce differences in Δx_3;
        foreach value of Δw_4[j], w_4[4c_j], w_4[4c_j + 1], w_4[4c_j + 2], w_4[4c_j + 3] do
            Deduce differences in Δy_3;
            Use the differential property of the AES SBox to deduce the values in x_3
            and x'_3;
            Deduce SR^{-1}(u_2)[4c_i], SR^{-1}(u_2)[4c_i + 1], SR^{-1}(u_2)[4c_i + 2],
            SR^{-1}(u_2)[4c_i + 3];
            Deduce SR(k_3)[4c_j], SR(k_3)[4c_j + 1], SR(k_3)[4c_j + 2], SR(k_3)[4c_j + 3];
            Empty a multiset M;
            forall the differences Δz_1[b_i] do
                Obtain a column x_2, and then a state x_3;
                Add Δx_5[j] to M;
            end
            Add M to the lookup table T;
        end
    end
    return T of size ≈ 2^80
end
```

Algorithm 11: Construction of the table used in the efficient attack.

```
Function ConstructTable2(i, j)
    b_i ← i − 4(i mod 4) mod 16 // x_1[i] must be located on column 0;
    c_i ← ⌊b_i/4⌋;
    k ← ((i + 1) mod 4) + 4        // Position of the active byte on column 1 of x_1;
    b_k ← k − 4(k mod 4) mod 16;
    c_j ← ⌊j/4⌋;
    Empty a lookup table T;
    foreach value of Δz_1[b_i], Δz_1[b_k], x_2[4c_i], x_2[4c_i + 1], x_2[4c_i + 2], x_2[4c_i + 3] do
        Deduce differences in Δx_3;
        foreach value of Δw_4[j], w_4[4c_j], w_4[4c_j + 1], w_4[4c_j + 2], w_4[4c_j + 3] do
            Deduce differences in Δy_3;
            Use the differential property of the AES SBox to deduce the values in x_3 and x'_3;
            Deduce SR^{-1}(u_2)[4c_i], SR^{-1}(u_2)[4c_i + 1], SR^{-1}(u_2)[4c_i + 2], SR^{-1}(u_2)[4c_i + 3];
            Deduce SR(k_3)[4c_j], SR(k_3)[4c_j + 1], SR(k_3)[4c_j + 2], SR(k_3)[4c_j + 3];
            Empty a multiset M;
            forall the differences Δz_1[b_i] do
                Obtain a column x_2, and then a state x_3;
                Add Δx_5[j] to M;
            end
            Add M to the lookup table T;
        end
    end
    return T of size ≈ 2^88
end
```

8.B Efficient Attack

Algorithm 12: – An efficient attack.

```
Function EfficientAttack()
    forall the (i, j) ∈ {0, ..., 3} × {0, ..., 15} do                // The 2^6 tables
        T_{i,j} ← ConstructTable2(i, j);
    end
    while true do                                                   // ≈ 2^35 times
        Ask for a structure S of 2^64 plaintexts P_m where bytes in diagonals 0 and 1 assume all
        values;
        forall the k ∈ {0, ..., 3} do                               // Non-zero column of Δx_6
            Empty a hash table T of list of plaintexts;
            forall the corresponding ciphertexts C_m do
                index ← (SR^{-1} ∘ MC^{-1}(C_m))[{0, ..., 15} − {4k, ..., 4k + 3}];
                forall the P ∈ T[index] do
                    Consider the pair (P, P_m)        // ≈ 2^33 pairs by structure;
                    forall the (i, l_j) ∈ {0, ..., 3} × {0, ..., 3} do
                        j ← 4k − 3l_j  mod 16;
                        OnlinePhase((P, P_m), i, j, T_{i,j}, S);
                    end
                end
                T[index] ← T[index] ∪ {P_m};
            end
        end
    end
end
Function OnlinePhase((m, m'), i, j, T, S)
    b_j ← (j − 4 × (j  mod 4))  mod 16        // Retrieving right positions;
    c_j ← ⌊b_j/4⌋                         // because of the ShiftRows;
    Col_j ← {4c_j, ..., 4c_j + 3} ;
    forall the k_{-1}[0, 5, 10, 15] s.t. Δw_0[{0, ..., 3} − {i}] = 0 do
        Construct δ-set D from m;
        forall the SR(u_6)[Col_j] s.t. Δz_5[Col_j − {j}] = 0 do
            Decrypt column c_j of x_6 for D;
            forall the u_5[b_j] do
                Decrypt byte j of x_5 for D;
                Construct multiset M of Δx_5;
                if M ∈ T then  return ExhaustiveSearch() ;
            end
        end
    end
end
```

8.C Truncated differential characteristics used in the simple attack

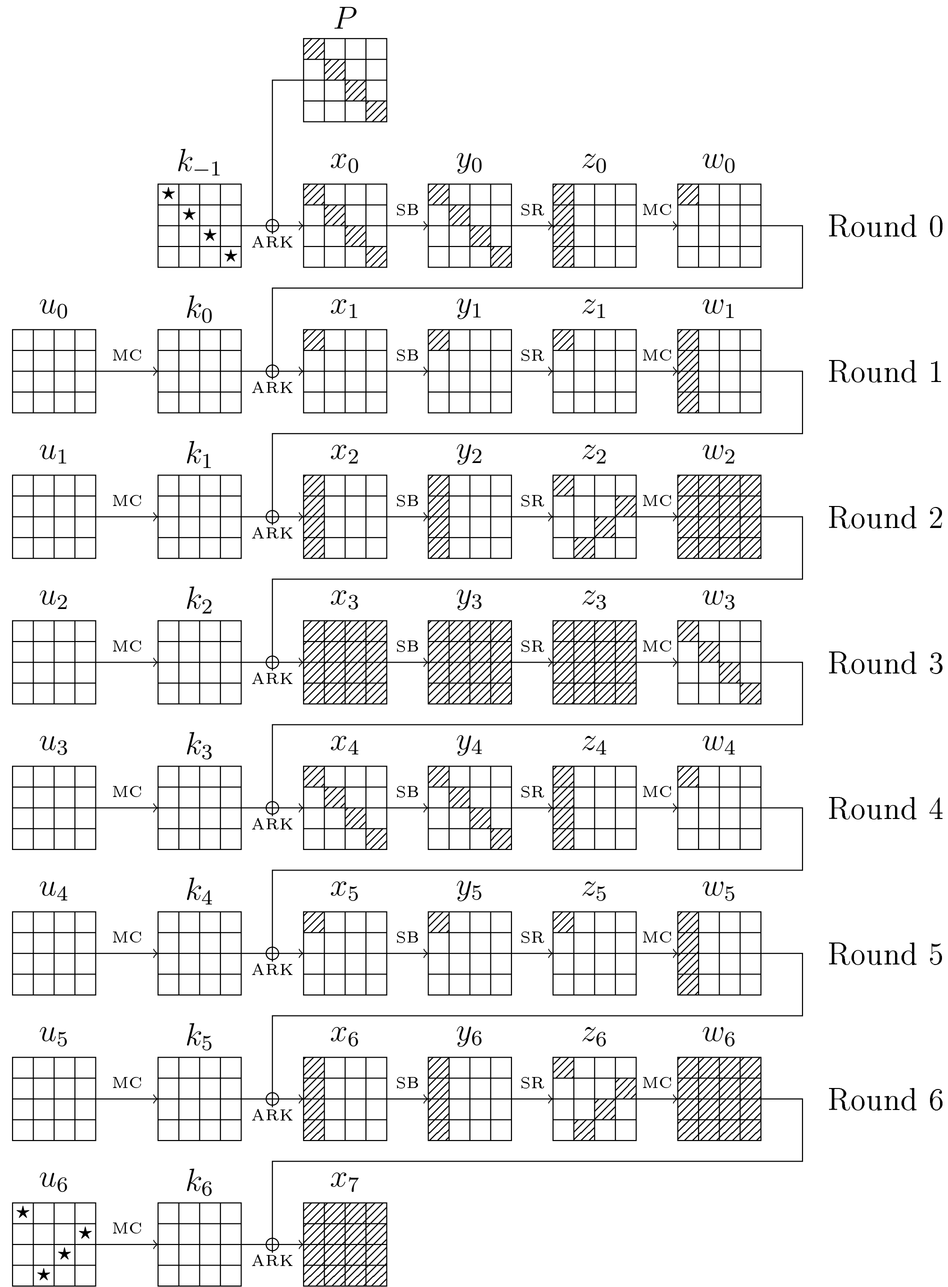


Figure 8.13: Complete 7-round truncated differential characteristic used in the simple attack of section 8.2.

Bibliographie

[AK97] Ross J. Anderson and Markus G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Security Protocols Workshop*, Lecture Notes in Computer Science, pages 125–136. Springer, 1997.

[BBC+08] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. Sosemanuk, a fast software-oriented stream cipher. In Robshaw and Billet [RB08], pages 98–118.

[BDD+12] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Pierre-Alain Fouque, Nathan Keller, and Vincent Rijmen. Low-data complexity attacks on aes. *IEEE Transactions on Information Theory*, 58(11) :7002–7017, 2012.

[BDF11] Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced AES and applications. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2011.

[BDK07] Eli Biham, Orr Dunkelman, and Nathan Keller. Improved slide attacks. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2007.

[BDK+10] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key recovery attacks of practical complexity on aes-256 variants with up to 10 rounds. In Gilbert [Gil10], pages 299–319.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT*, Lecture Notes in Computer Science, pages 37–51. Springer, 1997.

[BGN05] Eli Biham, Louis Granboulan, and Phong Q. Nguyen. Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In *FSE*, pages 359–367, 2005.

[Bir06] Alex Biryukov. The Design of a Stream Cipher LEX. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 67–75. Springer, 2006.

[Bir08a] Alex Biryukov. Design of a New Stream Cipher-LEX. In Robshaw and Billet [RB08], pages 48–56.

[Bir08b] Alex Biryukov. Design of a new stream cipher-lex. In Robshaw and Billet [RB08], pages 48–56.

[BK00] Eli Biham and Nathan Keller. Cryptanalysis of Reduced Variants of Rijndael. In *3rd AES Conference, New York, USA*, 2000.

[BK07a] Alex Biryukov and Dmitry Khovratovich. Two new techniques of side-channel cryptanalysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2007.

[BK07b] Alex Biryukov and Dmitry Khovratovich. Two New Techniques of Side-Channel Cryptanalysis. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 195–208. Springer, 2007.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full `AES-192` and `AES-256`. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and related-key attack on the full aes-256. In Halevi [Hal09], pages 231–249.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full `AES`. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer, 2011.

[BN09] Alex Biryukov and Ivica Nikolic. A New Security Analysis of `AES-128`. CRYPTO 2009 rump session, slides only, 2009.

[BN10] Alex Biryukov and Ivica Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers : Application to aes, camellia, khazad and others. In Gilbert [Gil10], pages 322–344.

[Bog07] Andrey Bogdanov. Improved Side-Channel Collision Attacks on AES. In *Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 84–95. Springer, 2007.

[BPW06] Johannes Buchmann, Andrei Pyshkin, and Ralf-Philipp Weinmann. A Zero-Dimensional Gröbner Basis for AES-128. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2006.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO*, Lecture Notes in Computer Science, pages 513–525. Springer, 1997.

[BS03] Johannes Bloemer and Jean-Pierre Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography*, Lecture Notes in Computer Science, pages 162–181. Springer, 2003.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.

[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.

[BW00] Alex Biryukov and David Wagner. Advanced slide attacks. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer, 2000.

[CBW08] Nicolas Courtois, Gregory V. Bard, and David Wagner. Algebraic and slide attacks on keeloq. In Nyberg [Nyb08], pages 97–115.

[CFGR10] Christophe Clavier, Benoit Feix, Georges Gagnerot, and Mylène Roussellet. Passive and Active Combined Attacks on AES Combining Fault Attacks and Side Channel Analysis. In *FDTC*, pages 10–19, 2010.

[Cid04] Carlos Cid. Some Algebraic Aspects of the Advanced Encryption Standard. In Dobbertin et al. [DRS05], pages 58–66.

[CKK+01] Jung Hee Cheon, MunJu Kim, Kwangjo Kim, Jung-Yeun Lee, and Sung-Woo Kang. Improved impossible differential cryptanalysis of rijndael and crypton. In Kim [Kim02], pages 39–49.

[CL05] Carlos Cid and Gaëtan Leurent. An Analysis of the XSL Algorithm. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 333–352. Springer, 2005.

[CP02] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Zheng [Zhe02], pages 267–287.

[CR06] Christophe De Cannière and Christian Rechberger. Finding sha-1 characteristics : General results and applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[CT05] Hamid Choukri and Michael Tunstall. Round Reduction Using Faults. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC' 05, pages 13–24, 2005.

[CY03] Chien-Ning Chen and Sung-Ming Yen. Differential Fault Analysis on AES Key Schedule and Some Countermeasures. In *ACISP*, Lecture Notes in Computer Science, pages 118–129. Springer, 2003.

[DF13] Patrick Derbez and Pierre-Alain Fouque. Exhausting demirci-selcuk meet-in-the-middle attacks against reduced-round aes. In *FSE*, 2013.

[DFJ12] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Faster Chosen-Key Distinguishers on Reduced-Round AES. In S. Galbraith and M. Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2012.

[DFJ13] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved key recovery attacks on reduced-round aes in the single-key setting. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2013.

[DK08] Orr Dunkelman and Nathan Keller. A New Attack on the LEX Stream Cipher. In Josef Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 2008.

[DK10a] Orr Dunkelman and Nathan Keller. Cryptanalysis of the Stream Cipher LEX, 2010. Available at `http://www.ma.huji.ac.il/~nkeller/Crypt-jour-LEX.pdf`.

[DK10b] Orr Dunkelman and Nathan Keller. The effects of the omission of last round's mixcolumns on aes. *Inf. Process. Lett.*, 110(8-9) :304–308, 2010.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher square. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[DKS10a] Orr Dunkelman, Nathan Keller, and Adi Shamir. Improved Single-Key Attacks on 8-Round `AES-192` and `AES-256`. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 158–176. Springer, 2010.

[DKS10b] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the kasumi cryptosystem used in gsm and 3g telephony. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2010.

[DKS11] Orr Dunkelman, Nathan Keller, and Adi Shamir. Alred blues : New attacks on aes-based mac's. Cryptology ePrint Archive, Report 2011/095, 2011. `http://eprint.iacr.org/`.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential Fault Analysis on A.E.S. In *ACNS*, Lecture Notes in Computer Science, pages 293–306. Springer, 2003.

[DR05a] Joan Daemen and Vincent Rijmen. A New MAC Construction ALRED and a Specific Instance ALPHA-MAC. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DR05b] Joan Daemen and Vincent Rijmen. A new mac construction alred and a specific instance alpha-mac. In *Fast Software Encryption : 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DR05c] Joan Daemen and Vincent Rijmen. The Pelican MAC Function. Cryptology ePrint Archive, Report 2005/088, 2005. `http://eprint.iacr.org/`.

[DRS05] Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors. *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, volume 3373 of *Lecture Notes in Computer Science*. Springer, 2005.

[DS08] Hüseyin Demirci and Ali Aydin Selçuk. A meet-in-the-middle attack on 8-round `AES`. In Nyberg [Nyb08], pages 116–126.

[DS09] Itai Dinur and Adi Shamir. Side channel cube attacks on block ciphers. *IACR Cryptology ePrint Archive*, 2009 :127, 2009.

[Dun09] Orr Dunkelman, editor. *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*. Springer, 2009.

[FKL+00] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner, and Doug Whiting. Improved cryptanalysis of rijndael. In Bruce Schneier, editor, *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2000.

[FLN07] Pierre-Alain Fouque, Gaëtan Leurent, and Phong Q. Nguyen. Automatic search of differential path in md4. *IACR Cryptology ePrint Archive*, 2007 :206, 2007.

[FP09] Thomas Fuhr and Thomas Peyrin. Cryptanalysis of radiogatún. In Dunkelman [Dun09], pages 122–138.

[Fur01] Soichi Furuya. Slide attacks with a known-plaintext cryptanalysis. In Kim [Kim02], pages 214–225.

[Gil10] Henri Gilbert, editor. *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[Gir04] Christophe Giraud. DFA on AES. In *AES Conference*, Lecture Notes in Computer Science, pages 27–41. Springer, 2004.

[GM00] Henri Gilbert and Marine Minier. A collision attack on 7 rounds of `Rijndael`. In `AES` *Candidate Conference*, pages 230–241, 2000.

[GP10] Henri Gilbert and Thomas Peyrin. Super-sbox cryptanalysis : Improved attacks for `AES`-like permutations. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 2010.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The led block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

[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.

[HCTW04] Hagai Bar-El Hamid, Hamid Choukri, David Naccache Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In *http ://eprint.iacr.org/2004/100.pdf*, 2004.

[HJ11] Martin Hell and Thomas Johansson. Breaking the stream ciphers f-fcsr-h and f-fcsr-16 in real time. *J. Cryptology*, 24(3) :427–445, 2011.

[Iso11] Takanori Isobe. A single-key attack on the full gost block cipher. In Joux [Jou11], pages 290–305.

[Jou11] Antoine Joux, editor. *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*. Springer, 2011.

[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.

[Kel04] Liam Keliher. Refined analysis of bounds related to linear and differential cryptanalysis for the aes. In Dobbertin et al. [DRS05], pages 42–57.

[Kim02] Kwangjo Kim, editor. *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, volume 2288 of *Lecture Notes in Computer Science*. Springer, 2002.

[Kim10] Chong Hee Kim. Differential Fault Analysis against AES-192 and AES-256 with Minimal Faults. *Fault Diagnosis and Tolerance in Cryptography, Workshop on*, 0 :3–9, 2010.

[KMT01a] Liam Keliher, Henk Meijer, and Stafford E. Tavares. Improving the upper bound on the maximum average linear hull probability for rijndael. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2001.

[KMT01b] Liam Keliher, Henk Meijer, and Stafford E. Tavares. New method for upper bounding the maximum average linear hull probability for spns. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2001.

[Knu98a] Lars R. Knudsen. DEAL - a 128 bit block cipher. In *Technical report 151, Departement of Informatics, University of Bergen, Norway*, 1998.

[Knu98b] Lars R. Knudsen. DEAL - a 128 bit block cipher. In *AES Round 1 Technical Evaluation, NIST*, 1998.

[KR07] Lars R. Knudsen and Vincent Rijmen. Known-Key Distinguishers for Some Block Ciphers. In Kurosawa [Kur07], pages 315–324.

[KRM08] Mehran Mozaffari Kermani and Arash Reyhani-Masoleh. A Lightweight Concurrent Fault Detection Scheme for the AES S-Boxes Using Normal Basis. In Oswald and Rohatgi [OR08], pages 113–129.

[Kuc09] Ozgul Kucuk. The hash function hamsi. Submission to NIST (updated), 2009.

[Kur07] Kaoru Kurosawa, editor. *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.

[LDKK08] Jiqiang Lu, Orr Dunkelman, Nathan Keller, and Jongsung Kim. New impossible differential attacks on AES. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2008.

[LJW13] Leibo Li, Keting Jia, and Xiaoyun Wang. Improved meet-in-the-middle attacks on aes-192 and prince. Cryptology ePrint Archive, Report 2013/573, 2013. http://eprint.iacr.org/.

[Mat93] Mitsuru Matsui. Linear cryptoanalysis method for des cipher. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993.

[MDRMH10] Hamid Mala, Mohammad Dakhilalian, Vincent Rijmen, and Mahmoud Modarres-Hashemi. Improved Impossible Differential Cryptanalysis of 7-Round AES-128. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 282–291. Springer, 2010.

[MP08] Stéphane Manuel and Thomas Peyrin. Collisions on sha-0 in one hour. In Nyberg [Nyb08], pages 16–35.

[MPP09] Marine Minier, Raphael C.-W. Phan, and Benjamin Pousse. Distinguishers for Ciphers and Known Key Attack against Rijndael with Large Blocks. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2009.

[MPRS09] Florian Mendel, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2009.

[MR02] Sean Murphy and Matthew J. B. Robshaw. Essential Algebraic Structure within the AES. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The rebound attack : Cryptanalysis of reduced Whirlpool and Grøstl. In Dunkelman [Dun09], pages 260–276.

[MSS06] Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. A Generalized Method of Differential Fault Attack Against AES Cryptosystem. In *CHES*, Lecture Notes in Computer Science, pages 91–100. Springer, 2006.

[Muk09] Debdeep Mukhopadhyay. An Improved Fault Based Attack of the Advanced Encryption Standard. In *Proceedings of the 2nd International Conference on Cryptology in Africa : Progress in Cryptology*, AFRICACRYPT '09, pages 421–434. Springer, 2009.

[MV04] Jean Monnerat and Serge Vaudenay. On Some Weak Extensions of AES and BES. In Javier Lopez, Sihan Qing, and Eiji Okamoto, editors, *ICICS*, volume 3269 of *Lecture Notes in Computer Science*, pages 414–426. Springer, 2004.

[NIS01] NIST. Advanced Encryption Standard (AES), FIPS 197. Technical report, NIST, November 2001.

[NPSS10] Ivica Nikolic, Josef Pieprzyk, Przemyslaw Sokolowski, and Ron Steinfeld. Known and Chosen Key Differential Distinguishers for Block Ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *ICISC*, volume 6829 of *Lecture Notes in Computer Science*, pages 29–48. Springer, 2010.

[NSA98] NSA. Skipjack and kea algorithm specifications, May 1998.

[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.

[OR08] Elisabeth Oswald and Pankaj Rohatgi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*. Springer, 2008.

[Pey07] Thomas Peyrin. Cryptanalysis of grindahl. In Kurosawa [Kur07], pages 551–567.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[PSC+02] Sangwoo Park, Soo Hak Sung, Seongtaek Chee, E-Joong Yoon, and Jongin Lim. On the security of rijndael-like structures against differential and linear cryptanalysis. In Zheng [Zhe02], pages 176–191.

[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.

[PY06] Raphael C.-W. Phan and Sung-Ming Yen. Amplifying Side-Channel Attacks with Techniques from Block Cipher Cryptanalysis. In Josep Domingo-Ferrer, Joachim Posegga, and Daniel Schreckling, editors, *CARDIS*, volume 3928 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2006.

[RB08] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.

[Riv09] Matthieu Rivain. Differential Fault Analysis on DES Middle Rounds. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 2009.

[SEHK12] Yu Sasaki, Sareh Emami, Deukjo Hong, and Ashish Kumar. Improved known-key distinguishers on feistel-sp ciphers and application to camellia. In Willy Susilo, Yi Mu, and Jennifer Seberry, editors, *ACISP*, volume 7372 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 2012.

[SHJ09] Paul Stankovski, Martin Hell, and Thomas Johansson. An efficient state recovery attack on x-fcsr-256. In Dunkelman [Dun09], pages 23–37.

[SLFP04] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A Collision-Attack on AES : Combining Side Channel- and Differential-Attack. In *CHES*, Lecture Notes in Computer Science, pages 163–175. Springer, 2004.

[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.

[SSHA08] Akashi Satoh, Takeshi Sugawara, Naofumi Homma, and Takafumi Aoki. High-Performance Concurrent Error Detection Scheme for AES Hardware. In Oswald and Rohatgi [OR08], pages 100–112.

[SY11] Yu Sasaki and Kan Yasuda. Known-Key Distinguishers on 11-Round Feistel and Collision Attacks on Its Hashing Modes. In Joux [Jou11], pages 397–415.

[TFY07] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA Mechanism on the AES Key Schedule. In *FDTC '07 : Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 62–74. IEEE Computer Society, 2007.

[TM09] M. Tunstall and D. Mukhopadhyay. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. Cryptolog ePrint Archive, Report 2009/575, 2009. `http://eprint.iacr.org/`.

[WLH11] Yongzhuang Wei, Jiqiang Lu, and Yupu Hu. Meet-in-the-middle attack on 8 rounds of the aes block cipher under 192 key bits. In Feng Bao and Jian Weng, editors, *ISPEC*, volume 6672 of *Lecture Notes in Computer Science*, pages 222–232. Springer, 2011.

[YWJ+09] Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, and Xiaoyun Wang. New birthday attacks on some macs based on block ciphers. In Halevi [Hal09], pages 209–230.

[Zhe02] Yuliang Zheng, editor. *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*. Springer, 2002.

Table des figures

3.1 Description of one AES round and the ordering of bytes in an internal state. 44
3.2 Key schedules of the variants of the AES: AES-128, AES-192 and AES-256. 45
3.3 An attack on one round AES given one known plaintext with time complexity of 2^{32} and memory complexity of 2^{24}. Bytes marked by gray are known (from previous steps of analysis). Bytes marked with tilted lines are retrieved using Property 15. 49
3.4 The attack with two known plaintexts on two round AES. 53
3.5 Two chosen plaintexts attack on two AES rounds. Gray bytes indicate the presence of a difference, and hatched bytes indicate the presence of a known difference. If byte i is known in x_0, then the actual values of all the bytes with the same number can be found. 56
3.6 The attack with one known plaintext on two round AES. 58
3.7 The attack on two rounds of AES without the second `MixColumns` using two known plaintexts. Bytes marked in black are guessed, and bytes marked in gray are known at this phase of the attack. 61
3.8 The attack on three rounds of AES using one known plaintext. 63
3.9 One known plaintext attack on 4.5 AES rounds. Black bytes are enumerated and stored in a hash table. White bytes are enumerated. Gray bytes are linear combinations of white and black bytes. Hatched bytes play no role. The number indicates the step of the attack in which the value of each byte is discovered. 67
3.10 The 3-Round Truncated Differential Used in the 6-round Differential Attack. 68

4.1 Differential path used in the attack against Pelican-MAC. Gray squares denote the presence of a difference. Hatched squares denote a known difference. 73
4.2 State Bytes which Compose the Output in Odd and Even Rounds of LEX. The gray bytes are the leaked bytes. 74
4.3 Gray squares are leaked to form the key-stream. The differences are null in squares with a 0. The differences in the hatched squares can be deduced from the leaked bytes and the existence of zero differences. 75
4.4 Second stage of the attack. 77
4.5 Third phase of the attack. 78

5.1 Fault injected on one byte between `MixColumns` at the 7th round and `MixColumns` at the 8th round on AES-128. Black bytes are active, white bytes are not. 81

5.2 Fault injected on one byte between `MixColumns` at the 6th round and `MixColumns` at the 7th round on AES-128. Black bytes are active, white bytes are not. 82

5.3 Colored bytes are active. Differences in black bytes are non-zero. 85

6.1 Assuming $n_i \leqslant n_o$, the attacker searches for a pair of input to the random permutation F_k differing in n_i known byte positions such that the output differs in n_o known byte positions. A gray cell indicates a byte with a truncated difference. 90

6.2 The 7-round truncated differential characteristic used to distinguish the `AES-128` from a random permutation. Black bytes are active, white bytes are not. 90

6.3 The 7-round distinguishing attack focuses of the middle rounds. Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences. 91

6.4 Differences around the `SubBytes` layer of round 3: each Δ_j is fixed, whereas the δ_i are yet to be determined. 92

6.5 The 7-round distinguishing attack focuses of the middle rounds. Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences. 93

6.6 Generating a compatible key: gray bytes are known, and numbers indicate the order in which we guess or determine the bytes. 93

6.7 The 8-round truncated differential characteristic used to distinguish the `AES-128`. Black bytes are active, white bytes are not. 94

6.8 Black bytes have known values and differences, hatched bytes have known differences and white bytes have unknown values and/or differences. 95

6.9 Black bytes have known values and differences, gray bytes have known values, hatched bytes have known differences and white bytes have unknown values and/or differences. 96

6.10 Generating a compatible key: gray bytes are known, and numbers indicate the order in which we guess or determine the bytes. 98

7.1 4 AES-rounds. The 25 black bytes are the parameters of Property 18. Hatched bytes play no role. The differences are null in white squares 104

7.2 Online phase of Demirci and Selçuk attack. $\mathcal{B}_{on}$ is composed by gray and black bytes. Gray bytes are used to identify a δ-set and to order it. Black bytes are used to build the sequence from ciphertexts. Hatched bytes play no role. The differences are null in white squares. 105

7.3 Best variants on 7 rounds `AES`-192. 109

7.4 Best variants on 7 rounds `AES`-256. 109

7.5 Best variants on 8 rounds. . . . 110

7.6 Attack on 6 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 111

7.7 Attack on 7 AES rounds (key length : 256 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 113

7.8 Attack on 7 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 113

7.9 First part of the attack on 5 AES rounds. Black bytes are enumerated and stored in a hash table. Gray bytes are enumerated. Hatched bytes play no role. The differences are null in white squares 115

7.10 First part of the attack on 6 AES rounds. Black bytes are enumerated and stored in a hash table. Gray bytes are enumerated. Hatched bytes play no role. The differences are null in white squares 116

7.11 Attack on 6 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 118

7.12 Attack on 7 AES rounds (key length : 192 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 119

7.13 Attack on 7 AES rounds (key length : 256 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 120

7.14 Attack on 8 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 120

7.15 Attack on 7 AES rounds (key length : 128 bits). Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 121

7.16 Attack on 9 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 124

8.1 General scheme of the meet-in-the-middle attack on AES, where some messages in the middle rounds may verify a certain ★ property used to perform the meet-in-the-middle. 127

8.2 The four middle rounds used in the 7-round attack from [DKS10a]. Dashed bytes are active, others inactive. 129

8.3 Truncated differential characteristic used in the middle of the 7-round attacks on AES. A hatched byte denotes a non-zero difference, whereas a while cell has no difference. 130

8.4 Example of a truncated differential characteristic used in the efficient attack on 7 rounds. 134

8.5 Scheme of the attack on 8 rounds. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes. 136
8.6 Attack on 8 AES rounds. Bytes of $\mathcal{B}_{off}$ are in black. Bytes of $\mathcal{B}_{on}$ are in gray. Hatched bytes play no role. The differences are null in white squares 138
8.7 Differential characteristic on 8 AES rounds. The differences are null in white squares. The value of bytes of $\mathcal{B}_{off}$ can be derived from the differences in circled bytes. 139
8.8 Scheme of the nine-round attack on AES-256. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes. 140
8.9 Scheme of the nine-round attack on AES-192. Gray bytes are needed to identify a δ-set and to build the multiset. Black bytes are needed to construct the table. White bytes are constant for a δ-set. If differences in hashed bytes are null then black bytes can be derived from the difference in circled bytes (helped by keyschedule). 141
8.10 Differential Enumeration: results on 7 rounds AES-128. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity. 142
8.11 Differential Enumeration: results on 8 rounds AES-192. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity. 143
8.12 Differential Enumeration: results on 9 rounds AES-256. All attacks have a data complexity of 2^{113} chosen plaintexts. Numbers in cells are the $\log_2$ of the numbers of attacks found with the same complexity. 144
8.13 Complete 7-round truncated differential characteristic used in the simple attack of section 8.2. 149

Résumé

Cette thèse est dédiée à la cryptanalyse de l'AES (Advanced Encryption Standard) qui est l'un des systèmes de chiffrement par bloc les plus répandu dans le monde. Nous y présentons une nouvelle technique pour résoudre un type particulier d'équations spécialement conçu pour attaquer l'AES. Cette technique est basée sur l'algèbre linéaire ainsi que sur la technique de la « Rencontre par le Milieu » et offre pour un système donné, plusieurs algorithmes de résolution de complexités différentes mais prédictibles. Ainsi nous avons conçu un programme pour trouver l'algorithme le plus rapide. Dans un premier temps nous l'avons appliqué directement aux systèmes d'équations décrivant un nombre réduit de tours d'AES et avons trouvé de nouvelles attaques lorsque la quantité de couples clair/chiffré est très limitée, améliorant celles trouvées manuellement par d'autres chercheurs. La technique étant générale nous avons pu utiliser le programme pour étudier d'autres modèles comme celui des attaques par fautes et celui des attaques à clé choisie ainsi que d'autres primitives cryptographiques comme la fonction d'authentification Pelican-MAC et le système de chiffrement par flot LEX. Enfin nous présentons une généralisation des attaques de Demirci et Selçuk publiées à la conférence FSE2008 ainsi qu'un algorithme qui nous a permis de trouver les meilleures attaques de cette classe, avec certaines parmi les meilleures connues à ce jour. Cet algorithme repose sur l'utilisation du précédent programme afin de déterminer le nombre de valeurs prises par des sous-ensembles d'octets de clé ou des états internes ainsi que la complexité de les énumérer.

Mots-clés: cryptanalyse, chiffrement symétrique, AES, recherche automatique d'attaques, rencontre par le milieu

Abstract

This thesis is dedicated to the cryptanalysis of the AES (Advanced Encryption Standard) which is one of the most widely deployed block ciphers. We present a new technique to solve a particular kind of equations designed to attack the AES. This technique relies on both the linear algebra and the "Meet-in-the-Middle" technique and, for any system of equations, leads to many solvers with different but predictable complexity. Thus we built a program in order to find the fastest solver. Initially we applied it directly to the systems of equations describing round-reduced versions of the AES and found new attacks when the data available to the adversary is very limited, improving the previous ones manually found by others researchers. As the technique is generic, we were able to use this program to study different models as faults or chosen-key attacks and different cryptographic primitives as both the message authentication code Pelican-MAC and the stream cipher LEX. Finally, we show a generalization of the attacks of Demirci and Selçuk published at the FSE2008 conference, together with an algorithm that allowed us to find the best attacks of this class, with some of them belonging to the best known ones. This algorithm relies on the previous program in order to determine the number of values assumed by a subset of key and state bytes as well as the complexity of enumerating them.

Keywords: cryptanalysis, AES, automatic attacks finder, meet-in-the-middle