Cryptanalyse des algorithmes de type Even-Mansour

Chrysanthi Mavromati

NNT : 2017SACLV036

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À L'UNIVERSITÉ DE
VERSAILLES-SAINT-QUENTIN-EN-YVELINES

Ecole doctorale n°508
Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

par

MME. CHRYSANTHI MAVROMATI

Cryptanalyse des algorithmes de type Even-Mansour

Thèse présentée et soutenue à Paris, le 24 janvier 2017.

Composition du Jury :

M.	LOUIS GOUBIN	Professeur Université de Versailles-Saint-Quentin-en-Yvelines	(Président du jury)
Mme.	ANNE CANTEAUT	Directeur de recherche INRIA Paris	(Rapporteur)
M.	DAVID NACCACHE	Professeur École Normale Supérieure	(Rapporteur)
M.	ANTOINE JOUX	Professeur Université Pierre et Marie Curie	(Directeur de thèse)
M.	PIERRE-ALAIN FOUQUE	Professeur Université Rennes I	(Examinateur)
M.	HENRI GILBERT	Ingénieur de recherche ANSSI	(Examinateur)
M.	PASCAL PAILLIER	Ingénieur de recherche CryptoExperts	(Examinateur)

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Cryptanalyse des algorithmes de type Even-Mansour

Mots clefs : Even-Mansour, chiffrement par blocs, MAC, PRINCE, Chaskey, attaques par collision

Résumé : Les algorithmes cryptographiques actuels se répartissent en deux grandes familles : les algorithmes symétriques et les algorithmes asymétriques. En 1991, S. Even et Y. Mansour ont proposé une construction simple d'un algorithme de chiffrement par blocs en utilisant une permutation aléatoire. Récemment, surtout pour répondre aux nouveaux enjeux de la cryptographie à bas coût, plusieurs algorithmes ont été proposés dont la construction est basée sur le schéma Even-Mansour.
Les travaux réalisés dans cette thèse ont pour objet l'analyse de ce type d'algorithmes. À cette fin, nous proposons une nouvelle attaque générique sur le schéma Even-Mansour. Ensuite, afin de montrer l'importance particulière du modèle multi-utilisateurs, nous appliquons cette attaque générique dans ce modèle. Ces deux attaques sur Even-Mansour introduisent deux nouvelles idées algorithmiques : les chaînes parallèles et la construction d'un graphe qui illustre les liens entre les clés des utilisateurs du modèle multi-utilisateurs. Finalement, basés sur ces idées, nous proposons des attaques sur les algorithmes de chiffrement par blocs DESX et PRINCE et sur le code d'authentification de message Chaskey.

Title : Cryptanalysis of Even-Mansour type algorithms

Keywords : Even-Mansour, bloc ciphers, MAC, PRINCE, Chaskey, collision based attacks

Abstract : Current cryptographic algorithms are divided into two families: secret-key algorithms (or symmetric algorithms) and public-key algorithms. Secret-key cryptography is characterized by the sharing of the same key K used by both legitimate users of the cryptosystem. Bloc ciphers are one of the main primitives of symmetric cryptography. In 1991, S. Even and Y. Mansour proposed a minimal construction of a bloc cipher which uses a random permutation. Recently, in the context of lightweight cryptography, many algorithms based on the Even-Mansour scheme have been proposed. In this thesis, we focus on the analysis of this type of algorithms. To this purpose, we propose a generic attack on the Even-Mansour scheme. To show the particular importance of the multi-user model, we adapt our attack to this context.
With these attacks, we introduce two new algorithmic ideas: the parallel chains and the construction of a graph which represents the relations between the keys of the users of the multi-user model. Finally, we use these ideas and we present attacks on the bloc ciphers DESX and PRINCE and on the message authentication code (MAC) Chaskey.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Remerciements

Les premières lignes de ces remerciements vont naturellement à Antoine Joux, mon directeur de thèse. Je le remercie de m'avoir permis de connaître le plaisir de travailler avec lui, de m'avoir montré son savoir faire et d'avoir été toujours disponible quand j'avais besoin de son aide. Il a été pour moi une source d'inspiration inestimable et je serai toujours impressionnée par son enthousiasme pour la recherche. Et parce que les mots ont une valeur plus grande quand c'est notre langue maternelle : Antoine, σε ευχαριστώ πολύ για όλα! Je suis sûre que tu arriveras à déchiffrer ce petit message !

Je suis très reconnaissante à Anne Canteaut et David Naccache d'avoir accepté de rapporter cette thèse. Malgré un emploi de temps plus que chargé, ils m'ont fait l'honneur de relire et corriger ce manuscrit. Je tiens à remercier particulièrement Anne pour sa grande gentillesse, ses précieux conseils et ses plusieurs relectures de qualité. Sans elle, ce manuscrit, n'aurait certainement pas cette forme. Mais surtout, je la remercie de m'avoir accueillie à l'équipe-projet SECRET de l'INRIA il y a quelques années et de m'avoir donné une place dans le bureau 12 !

Je tiens aussi à remercier Pierre-Alain Fouque, Henri Gilbert, Louis Goubin et Pascal Paillier d'avoir accepté de participer à ce jury de thèse. Je profite de l'occasion pour remercier encore une fois Pierre-Alain d'avoir travaillé avec moi pour notre article à Asiacrypt. Notre soumission finale quelques minutes avant la deadline alors que j'étais dans une station de ski en Savoie et que j'essayais d'avoir Internet via mon téléphone, restera toujours gravée dans ma mémoire.

Il m'aurait été bien évidemment impossible de réaliser cette thèse sans le support et le financement de Sogeti. Merci à tous les membres de l'équipe R&D avec qui j'ai eu le plaisir de travailler et qui m'ont donné une vision différente du monde de la sécurité. Je ne vais pas tous les nommer de peur d'en oublier mais je suis sûre que certains vont se reconnaître dans ces lignes. Cependant, je voudrais remercier plus particulièrement Jean-Marc Bianchini et Christine Guillauneux de m'avoir fait confiance et d'avoir pris le risque de financer la toute première thèse au sein de Sogeti. Je ne peux pas ne pas remercier Yann le Glouahec (Ivan n'aura pas de remerciements !), même s'il n'est plus un collègue depuis un moment maintenant. Merci pour toutes les discussions que nous avons fait en vrai et sur IRC, pour toutes les bières que nous avons bues ensemble en ragotant (je promets que je vais retrouver du temps), pour les encouragements pour la thèse mais, aussi, merci de m'avoir appris que EAX n'est pas seulement un mode

opératoire !

Faire une thèse partagée entre deux établissements c'est déjà difficile, faire une thèse partagée entre trois établissements c'est pratiquement impossible ! Finalement, je n'ai pas passé de temps à Versailles mais je remercie toute l'équipe CRYPTO de leur bonne humeur et de leur gentillesse à chaque fois qu'on se rencontrait. Je remercie plus particulièrement Louis Goubin de son aide précieuse et pour sa disponibilité à chaque fois que j'en avais besoin.

En parlant de trois établissements, je me rends compte que je dois aussi parler d'un quatrième. Merci à Pascal Paillier, Matthieu Finiasz, Matthieu Rivain, Cécile Delerablée, Thomas Baignères, Tancrède Lepoint et Dahmun Goudarzi de m'avoir accueillie plusieurs fois chez CryptoExperts. Ces passages vont bien me manquer. Je profite pour re-dire un grand merci à Matthieu Finiasz qui a toujours été prêt à m'aider et de ses bons conseils pendant toutes ces années.

Mes années en thèse n'auraient pas été les mêmes sans la compagnie des autres thésards de la toute nouvelle équipe Almasty. Tout d'abord, je tiens à remercier Cécile Pierrot de sa gentillesse, d'avoir passé du temps avec moi pendant mes passages à Jussieu et de m'avoir fait une petite place pour que je puisse travailler avec eux. En deuxième place (juste par ordre chronologique d'arrivée) merci à Alexandre Gélin de sa bonne humeur, de son support pendant la période difficile de la fin de la rédaction et d'avoir été courageux pendant mes répétitions. Alexandre, à chaque fois que je pense à Jussieu et ses murs gris, tu donnes un peu de couleur à cette image avec tes choix de chemises colorées ! Je n'oublie pas de remercier les nouveaux venus Thomas Espitau et Natalia Kharchenko et de leur souhaiter bonne chance pour la suite.

Je voudrais remercier tous mes amis à Paris, qui n'ont pas de lien direct avec ma thèse, mais, qui m'ont aidé à construire ici mon nouveau chez moi. Merci à Jonathan pour toutes les discussions en vrai et sur hangouts, pour toutes les soirées bien arrosées passées dans le 13^{e} arrondissement. Merci d'avoir écouté mes galères de jeune maman et de m'avoir remonté le moral tant de fois. Merci à Evgenia et Aggelos, que j'ai rencontrés pendant mes tous premiers jours à Paris et qui sont maintenant plus que des amis. Merci à pilki de m'avoir prêté son ordinateur quand le mien a décidé de me lâcher en plein milieu de la rédaction ! Quand on est expatrié, on passe beaucoup de temps au téléphone avec nos proches. Du coup, je voudrais remercier Athena de toutes les heures que nous avons passées au téléphone à discuter de tout un tas de choses. Des choses pas toujours liées à la thèse et au travail mais qui font un bien fou au moral !

Je tiens aussi à remercier mes parents et ma sœur de leur support continu. Ils ont su me faire confiance et ont toujours participé de près ou de loin à tous mes projets. Je voudrais surtout remercier mes parents d'avoir fait le long trajet depuis Ioannina pour assister à ma soutenance bien qu'ils ne parlent pas français et qu'ils ne connaissent pas du tout le domaine de la cryptographie. Ευχαριστώ πολύ, μέσα από την καρδιά μου, για την υπομονή και την βοήθειά σας αλλά και την εμπιστοσύνη που μου δείξατε όλα αυτά τα χρόνια !

Finalement, parce qu'habituellement ça se fait comme ça, Stéphane, de tout mon cœur, merci pour tout ! Merci de ton support sans faille pendant toutes ces années avant et pendant la thèse. Merci de croire en moi et de m'avoir menacée pour envoyer mon tout premier mail à Antoine quand j'étais trop timide pour le contacter. Merci d'avoir tout lu et relu, des mails tous simples sans intérêt jusqu'au manuscrit de thèse, sans jamais te plaindre. Merci d'avoir été là et de m'avoir supportée pendant cette dernière année qui nous a amené tant de joie mais pendant laquelle les difficultés et les malchances s'accumulaient sans arrêt. Merci de m'avoir aidée à surmonter la rude période du retour au travail pendant laquelle je n'étais ni assez efficace au boulot ni assez bonne mère. Mais surtout, merci d'être là, tous les jours, avec notre petit Hector pour me rappeler l'essentiel de ma vie !

Overview

This document covers the work that I have carried out during the period $2013-2016$ as a PhD student. It was financed by an Industrial Agreement for Training through Research (CIFRE Agreement) signed between the company SOGETI France and the University of Versailles-Saint-Quentin-en-Yvelines. All results described here are located in the area of symmetric key cryptography.

Modern cryptography is splitted into two major branches: symmetric (or secret-key) cryptography and public-key cryptography. Its goal is the design of mechanisms protecting the confidentiality, integrity and authentication of data. Symmetric cryptography includes mainly four types of algorithms: block ciphers, stream ciphers, hash functions and message authentication codes (MAC). Block ciphers operate on fixed-length groups of bits (blocks of data). Such blocks are typically 128 bits in length and they are transformed into blocks of the same size under the action of a secret key.

At Asiacrypt 1991, S. Even and Y. Mansour [EM91, EM97] described a very efficient design to construct a block cipher. Their scheme is the following: they build a keyed permutation family by using a public permutation and two whitening keys. The first key is xored with the plaintext, then the public permutation is applied and finally the second key is xored to the permutation's output to obtain the final value. Thanks to its simplicity, the Even-Mansour scheme inspired many algorithms proposed in recent years. More precisely, many algorithms proposed in the context of lightweight cryptography are using its construction and replace the public permutation by a keyed function.

In this thesis, we have studied algorithms based on the Even-Mansour scheme. We propose here generic attacks on the Even-Mansour scheme. We apply the basic techniques of this attack on other algorithms that are using the same construction: the lightweight block cipher PRINCE [BCG$^+$12], the MAC algorithm Chaskey [MMH$^+$14a] and the block cipher DESX, which actually inspired the work of Even and Mansour. Our attacks apply to the multi-users setting as well as to the classic model with a single user.

Chapter 1

The first chapter of this thesis is an introduction to the field of modern cryptography. Its purpose is to introduce the notions that will be useful for the understanding of the results of this thesis. We present here the main primitives of symmetric cryptography. We emphasize on block ciphers and message authentication codes (MAC). We also give a brief description of lightweight cryptography.

Chapter 2

This chapter deals with the notion of finding collisions and how they can be used in a cryptographic context. At the beginning of the chapter, we explain generic results on collisions and we show why their existence is crucial to the security of many cryptographic primitives. Then, we study collisions generated after iterating a random function. We also describe several algorithmic techniques that allow us to detect this type of collisions more efficiently than generic methods. Finally, we focus on parallelizable collision search methods that reduce time complexity of several cryptographic attacks. More precisely, we present the parallelizable collision search method of van Oorschot and Wiener [vOW99] that uses the distinguished points technique. All attacks presented in this thesis are actually inspired by this method.

Chapter 3

In the third chapter we present our attacks on the Even-Mansour [EM91, EM97] scheme. First, we describe this scheme and then we present a new type of collision attack that we presented at the conference Asiacrypt 2014 in collaboration with Pierre-Alain Fouque and Antoine Joux [FJM14]. This attack uses two new algorithmic ideas. The first idea modifies the basic technique of van Oorschot and Wiener: instead of waiting for two chains to merge, we now require that they become *parallel*. Using this first idea, we present a generic attack on the Even-Mansour scheme. The second idea is the construction of a graph to recover the users' keys. We combine the two ideas and present an attack on Even-Mansour scheme in the multi-user model. In this model, which is not always studied in the field of symetric cryptography, the adversary tries to recover keys of many users in parallel more efficiently than with classical attacks, *i.e.* the number of recovered keys multiplied by the time complexity to find a single key, by amortizing the cost among several users.

Chapter 4

We present, in this chapter, our work on the Even-Mansour type algorithms. After the attacks presented in Chapter 3, we naturally wonder whether this new type

of attack can be applied on other algorithms. Here, we present attacks on the algorithms PRINCE [BCG+12], DESX and Chaskey [MMH+14a] which are all following the construction of Even-Mansour scheme. However, despite the similarities of these constructions, the generic attack on Even-Mansour can not be simply applied in these algorithms. An important difference is that these algorithms do not use a public permutation but a function parameterized by the user's key. We managed to modify the basic attack on Even-Mansour and apply it on PRINCE. This work was also presented at Asiacrypt 2014 [FJM14]. The similarities between the algorithms PRINCE and DESX also allowed us to present attacks against DESX in certain contexts. Finally, we present our attacks on the MAC Chaskey which were presented at the conference Selected Areas of Cryptography - SAC 2015 [Mav15].

Introduction générale

Les travaux de recherche de cette thèse ont été effectués pendant les années $2013 - 2016$. Ils ont été réalisés dans le cadre d'une Convention Industrielle de Formation par la Recherche (CIFRE) signée entre la société SOGETI France et l'Université de Versailles-Saint-Quentin-en-Yvelines. Tous les travaux présentés pendant ces trois années de thèse s'inscrivent dans le domaine de la cryptographie symétrique.

La cryptographie moderne se divise en deux grandes branches : la cryptographie symétrique (ou, autrement appelée, cryptographie à clé secrète) et la cryptographie asymétrique (ou cryptographie à clé publique). Son but est la conception de mécanismes permettant de protéger principalement la confidentialité, l'intégrité et l'authentification de données. La cryptographie symétrique inclut essentiellement quatre familles d'algorithmes : les algorithmes de chiffrement par blocs, les algorithmes de chiffrement à flot, les fonctions de hachage et les algorithmes d'authentification de message (MAC). Les algorithmes de chiffrement par blocs opèrent sur un bloc de taille fixe et sont construits à partir d'une fonction itérative (appelée fonction de tour) et d'une clé maître K. La fonction de tour doit faire des opérations simples dans le but de garantir la rapidité d'exécution de l'algorithme.

En 1991, S. Even et Y. Mansour proposent un nouvel algorithme de chiffrement par blocs appelé schéma Even-Mansour [EM91, EM97]. Leur but est de construire un algorithme minimaliste ayant une preuve formelle de sécurité. Ce schéma utilise une permutation publique et deux clés de blanchiment pour construire une permutation avec clé. Grâce à la simplicité de sa construction, le schéma Even-Mansour est devenu la base de plusieurs algorithmes proposés ultérieurement. Plus particulièrement, plusieurs algorithmes appartenant au domaine de la cryptographie à bas coût (*lightweight cryptography*) ont été inspirés par ce schéma.

Pendant cette thèse, nous nous sommes donc intéressés aux algorithmes de type Even-Mansour. Plus précisément, nous avons étudié ce schéma et nous avons proposé des attaques génériques sur le schéma lui-même qui introduisent deux nouvelles méthodes algorithmiques. En outre, nous avons réussi à appliquer ces méthodes à l'algorithme à bas coût de chiffrement par blocs PRINCE [BCG$^+$12], le code d'authentification de message Chaskey [MMH$^+$14a] et à l'algorithme de chiffrement par blocs DESX. Nos attaques s'appliquent aussi bien dans un modèle classique avec un seul utilisateur

que dans un modèle multi-utilisateurs.

Chapitre 1

Le premier chapitre de cette thèse est une introduction au domaine de la cryptographie moderne. Plus précisément, nous présentons les principales constructions de la cryptographie symétrique qui nous seront utiles pour la suite : les algorithmes de chiffrement par blocs et les algorithmes de code d'authentification de message (MAC). Nous présentons également une description brève de la cryptographie à bas coût. Le but de ce chapitre est d'introduire les notions qui seront utiles pour la compréhension des résultats de cette thèse.

Chapitre 2

Dans le deuxième chapitre, nous nous intéressons aux techniques de recherche de collisions et, plus particulièrement, à la façon dont elles peuvent être utilisées dans un contexte cryptographique. Au début du chapitre, nous expliquons des résultats génériques concernant l'existence de collisions dans certains contextes cryptographiques. Ensuite, nous nous intéressons aux collisions générées après avoir itéré une fonction aléatoire plusieurs fois et nous décrivons certains algorithmes plus efficaces que les méthodes génériques pour détecter ces collisions. Finalement, nous nous intéressons aux algorithmes de recherche parallélisable de collisions qui réduisent la complexité en temps de plusieurs attaques cryptographiques. Plus particulièrement, nous présentons la méthode de van Oorschot et Wiener [vOW99] qui est la base de nos attaques. Cette méthode utilise des chaînes calculées en itérant une fonction aléatoire. En utilisant la méthode des points distingués, nous détectons les chaînes qui fusionnent et nous arrivons donc à détecter la collision recherchée.

Chapitre 3

Le troisième chapitre est consacré au schéma Even-Mansour. Tout d'abord, nous décrivons ce schéma et, ensuite, nous présentons un nouveau type d'attaque par collision sur le schéma Even-Mansour que nous avons présenté à la conférence *Asiacrypt* 2014 en collaboration avec Pierre-Alain Fouque et Antoine Joux [FJM14]. Cette attaque utilise deux nouvelles idées algorithmiques. La première idée modifie l'algorithme de recherche parallélisable des collisions de van Oorschot et Wiener. Dans notre cas, pour détecter les collisions, nous n'attendons pas que les chaînes fusionnent mais nous voulons que les chaînes deviennent *parallèles*. En utilisant cette idée, nous présentons une attaque générique sur le schéma Even-Mansour. La deuxième idée est la construction d'un graphe pour récupérer les clés des utilisateurs. En combinant les deux idées, nous présentons une attaque sur Even-Mansour dans un modèle multi-utilisateurs. Dans ce modèle, qui n'est pas toujours étudié dans le domaine de la cryptographie symétrique,

l'adversaire souhaite attaquer tout un groupe d'utilisateurs pour un coût aussi faible que possible.

Chapitre 4

Nous présentons dans ce chapitre nos travaux sur certains algorithmes spécifiques de type Even-Mansour. Après les attaques présentées au chapitre 3, la question que nous nous posons naturellement est de savoir si ce nouveau type d'attaque peut avoir des applications à d'autres algorithmes. Nous présentons ici des attaques sur les algorithmes PRINCE [BCG+12], DESX et Chaskey [MMH+14a] qui suivent le modèle du schéma Even-Mansour. Cependant, malgré les similarités de leurs constructions, l'attaque générique sur Even-Mansour ne peut pas s'appliquer naïvement sur ces algorithmes. Une différence importante est que ces algorithmes n'utilisent pas une permutation publique mais une fonction paramétrée par une partie de la clé de l'utilisateur. Nous avons réussi à modifier l'attaque sur Even-Mansour et à l'appliquer sur PRINCE dans un modèle multi-utilisateurs et dans un modèle classique. Ces travaux ont été également présentés à *Asiacrypt* 2014 [FJM14]. Les similarités entre les algorithmes PRINCE et DESX nous ont aussi permis de présenter des attaques contre DESX dans certains contextes. Finalement, nous expliquons nos attaques sur le MAC Chaskey qui ont été présentées à la conférence *Selected Areas of Cryptography - SAC* 2015 [Mav15].

1. Introduction

1.1 Introduction à la cryptographie

La cryptologie, autrement appelée la science du secret, apparait dans l'antiquité. Son but est la protection de l'information. Son nom provient de deux mots grecs : κρυπτός qui signifie "caché" et λόγος qui signifie "discours parlé ou écrit". Les deux branches de la cryptologie sont la cryptographie et la cryptanalyse. Le but de la cryptographie est la conception de mécanismes permettant de garantir la protection de l'information, transmise via un canal non sécurisé, contre différents types d'adversaires. Par opposition, la finalité de la cryptanalyse est d'affaiblir la sécurité de ces mécanismes.

Les premiers mécanismes cryptographiques connus remontent à l'antiquité. Par exemple, un de premiers systèmes cryptographiques est la *scytale* qui a été utilisée en Grèce en 487 avant J.-C. Il s'agit d'une simple transposition de lettres. Pour chiffrer un message, l'expéditeur enroulait une bande de cuir autour d'un bâton en bois (scytale), il écrivait le message sur la bande horizontalement et, ensuite, il enlevait la bande du bâton. Le destinataire devrait alors avoir la clé secrète, qui était dans ce cas la taille du diamètre du bâton, pour qu'il puisse lire le message. D'autres systèmes cryptographiques de l'antiquité sont le carré de Polybe (Grèce) et le cryptosystème de César (Rome). Plus

tard, nous retrouvons en France le cryptosystème de Vigenère. Finalement, le masque jetable, ou également appelé cryptosystème de Vernam, a été inventé aux États-Unis en 1917. En outre, la cryptographie a joué un rôle majeur pendant les deux guerres mondiales au 20^{e} siècle. Plus précisément, un autre procédé cryptographique célèbre, a été la machine ENIGMA utilisée par l'Allemagne nazie pendant la deuxième guerre mondiale.

Même si la cryptographie a toujours été largement utilisée dans les domaines militaires et diplomatiques, ses bases n'ont été théorisées qu'à la fin du 19^{e} siècle. Auguste Kerckhoffs écrit dans son article "La cryptographie militaire" [Ker83] (Journal des sciences militaires, février 1883) que la sécurité d'un cryptosystème ne doit reposer que sur le secret de la clé et non sur le secret du procédé utilisé. Autrement dit, hormis la clé, tous les paramètres du système doivent être supposés publiquement connus. Une autre publication fondamentale du domaine a été l'article *"The communication theory of secrecy systems"* de Claude Shannon en 1949. Dans cet article, Shannon pose les bases mathématiques d'un système de communication chiffrée à partir du modèle de la théorie de l'information.

De nos jours, la cryptographie reste toujours un outil très important dans le domaine militaire mais, grâce au développement des systèmes informatiques, elle est aussi largement utilisée dans la vie de tous les jours, par exemple pour les cartes bancaires, achats sur internet, badges d'accès, etc. Cela a créé de nouveaux besoins et a aidé à l'avancement de la cryptographie vers la science qu'elle est actuellement.

Généralités. Un système cryptographique vise à protéger l'information en garantissant trois propriétés importantes.

1. La *confidentialité* : protéger la confidentialité d'un message signifie que les données transmises ne sont pas dévoilées à une tierce personne.

2. L'*intégrité* : en protégeant l'intégrité, l'expéditeur et le destinataire du message s'assurent que les données n'ont pas été modifiées entre l'émission et la réception.

3. L'*authentification* : cette propriété assure au destinataire que le message provient bien de l'expéditeur.

Ces problèmes se posent souvent dans notre vie quotidienne. Même si la confidentialité n'est pas toujours demandée, il est souvent important d'identifier les utilisateurs d'un service et d'authentifier ce service, par exemple pour accéder à un réseau wifi, pour faire un paiement sur Internet en utilisant une carte de paiement, etc.

La cryptologie utilise deux procédés principaux, le chiffrement qui est une transformation appliquée à un message pour assurer sa confidentialité et/ou son intégrité et le déchiffrement qui consiste à retrouver, à l'aide d'une clé secrète, l'information initiale contenue dans le message chiffré (transformation inverse du chiffrement).

L'exemple classique d'utilisation des procédés cryptographiques met en présence deux protagonistes, Alice et Bob, qui souhaitent communiquer entre eux. Alice souhaite envoyer un message m à Bob à travers un canal de communication susceptible d'être espionné et donc supposé non sécurisé. Pour cela, elle aura besoin d'un algorithme de chiffrement $\mathcal{E}$ qui prend en paramètre la clé de chiffrement K_e et d'un algorithme de déchiffrement $\mathcal{D}$ qui prend en paramètre la clé de déchiffrement K_d.

Les algorithmes cryptographiques se répartissent en deux grandes familles qui diffèrent par l'utilisation par Alice et Bob d'une clé commune ou de deux clés distinctes : les algorithmes *symétriques*, ou à *clé secrète* et les algorithmes *asymétriques*, ou à *clé publique*

Chiffrement symétrique. La cryptographie symétrique est la plus ancienne forme de cryptographie. Elle nécessite le partage d'une clé secrète par les deux parties. Cette clé sera utilisée par le mécanisme du chiffrement pour chiffrer le message et par le mécanisme du déchiffrement pour que toute personne la possédant soit en mesure de retrouver le message clair à partir du chiffré. Les algorithmes de chiffrement symétrique utilisent surtout des opérations *simples* et donc ont l'avantage d'être extrêmement rapides. Cependant, deux problèmes se posent inévitablement. D'une part celui du nombre de clés à générer pour l'échange sécurisé entre plusieurs personnes du même groupe. En effet, chaque personne doit posséder une clé différente pour chaque communication. D'autre part celui de l'échange sécurisé des clés. La solution à ces questions a été apporté par la cryptographie asymétrique.

Chiffrement asymétrique. La cryptographie asymétrique a été inventée par Diffie et Hellman en 1976 [DH76]. Dans cette famille d'algorithmes, Alice et Bob possèdent chacun leur propre paire de clés : une clé publique et une clé privée. La clé publique peut être diffusée à toutes les autres entités sans aucune restriction et elle sert au chiffrement des messages. Par opposition, la clé privée, comme son nom l'indique, doit être connue uniquement de son propriétaire et elle est utilisée par le mécanisme de déchiffrement. Avec ce mécanisme, les deux entités peuvent communiquer sans avoir besoin d'échanger leurs clés secrètes. Néanmoins, les cryptosystèmes à clé publique sont plus longs, ce qui les rend inadaptés à l'échange de longs messages.

Chiffrement hybride. Cependant, les systèmes de chiffrement modernes les plus utilisés sont les systèmes *hybrides* qui utilisent à la fois les algorithmes à clé secrète et à clé publique. Dans ce cas, si Alice souhaite envoyer un message m à Bob, elle chiffrera m en utilisant les algorithmes symétriques et une clé secrète symétrique à usage unique K. En même temps, elle échange de façon sécurisé K en la chiffrant avec les algorithmes à clé publique et la clé publique de Bob.

Les travaux de cette thèse portent uniquement sur le chiffrement symétrique que nous allons voir plus en détail dans la suite de ce chapitre. En effet, nous rappelons ici les notions nécessaires à la compréhension des travaux effectués pendant cette thèse. Dans la suite de ce document, nous ne considérons que le cas où les clés de chiffrement et de déchiffrement sont égales et donc $K_e = K_d = K$.

1.2 La cryptographie symétrique

Comme déjà expliqué, la cryptographie symétrique est caractérisée par le partage d'une même clé K par les deux utilisateurs légitimes du cryptosystème. Nous utilisons deux algorithmes inverses l'un de l'autre, les algorithmes de chiffrement et de déchiffrement, dont le fonctionnement peut être rendu public sans que leur sécurité soit menacée. Donc, la sécurité de ce type de schéma cryptographique repose sur le secret de la clé K. Avant de décrire les principales constructions de la cryptographie symétrique, il convient de définir dans un premier temps ce type d'algorithmes. Étant donné un message m, une clé K et un algorithme de chiffrement symétrique $\mathcal{E}$, le chiffré de m est :

$$c = \mathcal{E}(K, m) = \mathcal{E}_K(m).$$

Pour retrouver le message clair m à partir du chiffré c, de la clé K et de l'algorithme de déchiffrement $\mathcal{D}$, nous appliquons :

$$m = \mathcal{D}(K, C) = \mathcal{D}_K(c).$$

La figure 1.1 montre le fonctionnement d'un tel système.

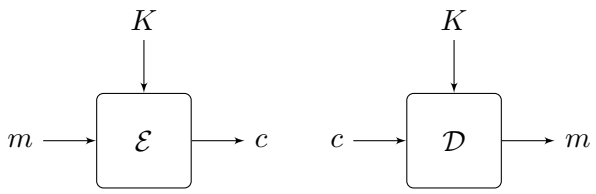

FIGURE 1.1 – Système de chiffrement symétrique

Les principales constructions de chiffrement symétrique sont :

Les algorithmes de chiffrement à flot. Un algorithme de chiffrement à flot consiste à combiner, en utilisant la fonction de OU exclusif (XOR), le message clair avec une suite binaire de même longueur, appelée suite chiffrante. Cette suite chiffrante est générée par un générateur pseudo-aléatoire, de façon indépendante à la fois du clair et du chiffré. Un des principaux avantages du chiffrement à flot est l'aspect que chaque bit du message clair est traité (chiffré ou déchiffré) indépendamment des autres sans qu'il ne

soit nécessaire d'attendre la transmission des bits suivants. En outre, grâce à leur rapidité, ils sont surtout utilisés dans les systèmes où il est primordial de pouvoir chiffrer et déchiffrer rapidement.

Les algorithmes de chiffrement par blocs. Un algorithme de chiffrement par bloc est un algorithme de chiffrement qui opère sur un bloc de taille fixe. La taille b du bloc fait typiquement $64, 128$ ou 256 bits. Pour chiffrer un message m de taille quelconque, le message est d'abord coupé en blocs de taille b, *i.e.* $m = m_0||m_1||m_2||\ldots$. Afin de chiffrer un message entier, nous faisons généralement appel à un mode opératoire de chiffrement. Le premier algorithme de chiffrement par blocs qui a été adopté comme standard par le *National Bureau of Standards (NBS)* a été l'algorithme défini par l'acronyme DES (*Data Encryption Standard*) [FIP77]. À la fin des années 90, le *NIST (National Institut of Standards et Technology)*, successeur du *NBS* lance une compétition pour remplacer DES. L'algorithme Rijndael [DR99] gagne la compétition et, en 2001, a été standardisé sous le nom AES (*Advanced Encryption Standard*) [FIP01].

Par habitude, bien qu'elle n'utilisent pas de clé, les fonctions de hachage sont rattachées aux algorithmes de cryptographie symétrique.

Les fonctions de hachage. Une fonction de hachage est une fonction h permettant de calculer une empreinte $h(m)$ de taille fixe n à partir d'une donnée m de taille arbitraire :

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

$$m \mapsto h(m).$$

Une fonction de hachage est un algorithme entièrement public sans intervention d'aucune valeur secrète. En outre, contrairement aux primitives de chiffrement, les fonctions de hachage ne doivent pas être inversibles. Une propriété de sécurité qu'une fonction de hachage doit avoir est la résistance aux collisions. C'est-à-dire qu'il est difficile de trouver deux messages ayant le même haché, c'est-à-dire m et $m^{'}$ tels que $h(m) = h(m^{'})$. Les fonctions de hachage cryptographiques sont surtout utilisées pour vérifier l'intégrité des données, les signatures électroniques, la protection des mots de passe et pour dériver une clé à partir d'un mot de passe. Les standards actuels sont les fonctions SHA-2 [FIP02] et la nouvelle norme SHA-3 [FIP15].

Cependant, assurer la confidentialité ne suffit pas toujours. C'est pourquoi la cryptographie symétrique dispose de constructions permettant d'assurer aussi l'intégrité d'un message.

Les algorithmes de codes d'authentification de message (MAC). Un code d'authentification de message (MAC) est une famille de calculs d'empreinte mais qui sont paramétrés par une clé secrète. Ces mécanismes cryptographiques peuvent être construits à

partir soit d'une fonction de hachage cryptographique, soit d'une fonction de hachage universelle, soit d'un algorithme de chiffrement par blocs. Le MAC le plus connu est l'algorithme HMAC (*Hash-based Message Authentication Code*) [KBC97].

1.3 Les algorithmes de chiffrement par blocs

Définition 1.1. *Un algorithme de chiffrement par blocs opère sur un bloc de taille fixe qui vaut typiquement* $64, 128, 160$ *ou* 256 *bits. Ce bloc est transformé en utilisant la fonction de chiffrement* $\mathcal{E}$ *spécifiée par une clé* K*. La fonction de chiffrement* $\mathcal{E}$ *doit être inversible et son inverse s'appelle la fonction de déchiffrement* $\mathcal{D}$*. Plus précisément, si* k *est la taille de la clé* K *et* b *la taille du bloc, nous définissons les fonctions de chiffrement et de déchiffrement*

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$$
$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$$

telles que, pour tout bloc m *de taille* b *et toute clé* k *de taille* k *:*

$$\mathcal{D}(\mathcal{E}(m, K), K) = m.$$

La taille du bloc intervient dans l'estimation de la sécurité de ces mécanismes. Effectivement, l'emploi de blocs de taille trop petite rend certaines attaques très efficaces. Il est donc important de ne pas utiliser des blocs de taille plus petite que 128 bits.

1.3.1 Construction d'un algorithme de chiffrement par blocs

Tout algorithme de chiffrement par blocs est construit à partir d'une fonction itérative et d'une clé maître K. Plus précisément, il consiste à itérer plusieurs fois la même fonction interne f qui est paramétrée par une clé. La clé utilisée à chaque tour n'est généralement pas la clé maître K mais une clé de tour qui est dérivée de la clé maître en utilisant un algorithme de cadencement des clés.

Actuellement, il existe principalement deux façons de construire la fonction interne f d'un algorithme de chiffrement par blocs : les réseaux de Feistel et les réseaux de substitution-permutation (*SP networks*).

Les réseaux de Feistel. Cette construction (figure 1.2) découpe le bloc m en deux morceaux de tailles égales L_0 et R_0. À chaque tour, la fonction f transforme alors les blocs $\{L_i, R_i\}$ en $\{L_{i+1}, R_{i+1}\}$ comme suit :

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus f(K, R_i) \end{aligned}$$

Les réseaux de Feistel utilisent le même circuit pour chiffrer et déchiffrer car ils sont

facilement inversibles et que la fonction de déchiffrement est identique à celle de chiffrement, à l'inversion de l'ordre des clés de tour près :

$$\begin{aligned} L_i &= R_{i+1} \oplus f(K, L_{i+1}) \\ R_i &= L_{i+1} \end{aligned}$$

Nous remarquons aussi que la partie droite reste inchangée à chaque tour. C'est à cause de cela qu'un grand nombre de tours est nécessaire pour garantir la sécurité d'un tel système.

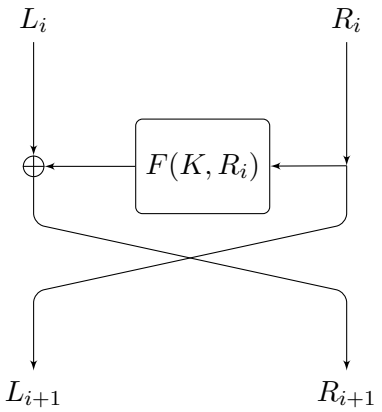

FIGURE 1.2 – Un tour d'un réseau de Feistel.

Cette construction a été proposée par le cryptologue allemand Horst Feistel et a été utilisée pour la première fois pour l'algorithme Lucifer proposé par Feistel lui-même et Don Coppersmith en 1973. L'ancien standard DES, qui est une version modifiée de Lucifer, est donc basée sur cette construction.

Les réseaux de substitution-permutation. Dans ce schéma, la fonction f est constituée de substitutions et de permutations de bits du bloc. Plus précisément, elle est constituée d'une succession de tours d'ajout de clé, de l'application d'une fonction non-linéaire (Boîtes-S) et d'une fonction linéaire (figure 1.3). L'algorithme AES, le standard actuel des algorithmes de chiffrement par blocs, suit ce principe de construction.

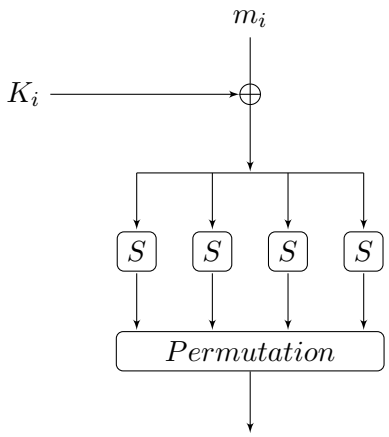

FIGURE 1.3 – Un exemple d'un tour du réseau substitution-permutation.

1.3.2 Notions de sécurité pour les algorithmes de chiffrement par blocs

Il faut préciser que lorsque nous évaluons la sécurité d'un algorithme de chiffrement, nous supposons que le fonctionnement de cet algorithme est connu. En effet, sa sécurité ne doit pas reposer sur le fait que le fonctionnement de celui-ci soit gardé secret.

La plus grande menace entre un algorithme de chiffrement par blocs est de retrouver la clé utilisée et ainsi de pouvoir déchiffrer tous les messages chiffrés avec celle-ci. Néanmoins, cette attaque n'est pas l'unique but d'un attaquant. Il existe de nombreux niveaux de sécurité en cryptographie qui s'appliquent aux algorithmes de chiffrement par blocs. Nous donnons ici les définitions des principaux.

Définition 1.2. *Non-inversibilité (OW : one-wayness). Soient un algorithme de chiffrement non-inversible $\mathcal{E}$ et un message clair m. Il est impossible de trouver m à partir de $\mathcal{E}(K, m)$ sans connaître la clé K.*

Définition 1.3. *Sécurité sémantique (IND : indistinguishability). Soit un adversaire ayant une puissance de calcul polynomiale, c'est-à-dire qu'il ne peut faire qu'un nombre polynomial de calculs, et un chiffré. Un chiffrement sûr sémantiquement assure que l'attaquant ne peut déduire aucune information sur le clair à partir du chiffré.*

Définition 1.4. *Non-malléabilité (NM : non-malleability). Soient $\mathcal{E}$ un algorithme de chiffrement non-malléable, m un message clair et $c = \mathcal{E}(K, m)$ le chiffré de m. Aucun attaquant polynomial ne doit être en mesure de dériver un deuxième chiffré $c^{'} = \mathcal{E}(K^{'}, m^{'})$ tel que m et $m^{'}$ soient reliés.*

Les propriétés précédentes peuvent être étudiées dans différents contextes, en fonction du type de données qui sont contrôlées par l'attaquant. Nous supposons donc que

l'adversaire, pour monter une attaque, a accès à certains échantillons de messages chiffrés avec une certaine clé. Les différents types d'attaques en fonction des échantillons que l'adversaire possède sont expliqués ci-dessous.

1. Attaque à chiffré seul (*Chiphertext only attack*). Ce modèle d'attaque suppose que l'adversaire dispose uniquement d'un certain nombre de messages chiffrés.

2. Attaque à clair connu (*Known plaintext attack*). Dans ce modèle, l'attaquant a un peu plus de pouvoir puisque il possède des messages clairs et leurs chiffrés correspondants.

3. Attaque à clair choisi (*Chosen plaintext attack*). Dans ce modèle d'attaque, l'adversaire peut choisir les clairs qui seront chiffrés. Ce scénario est un peu plus difficile à réaliser en pratique mais il reste réaliste.

4. Attaque à clair adaptatif choisi (*Adaptively chosen plaintext attack*). Même type d'attaque que l'attaque à clair choisi ci-dessus mais, ici, l'attaquant a la possibilité de choisir le dernier clair à chiffrer en fonction de chiffrés reçus précédemment.

5. Attaque à chiffré choisi (*Chosen ciphertext attack*). Ici, l'attaquant a accès à la machine du déchiffrement et choisit les textes chiffrés à déchiffrer.

6. Attaque à chiffré adaptatif choisi (*Adaptively chosen ciphertext attack*). Même type d'attaque que l'attaque à chiffré choisi mais l'adversaire a la possibilité de choisir chaque chiffré à déchiffrer en fonction des résultats obtenus précédemment.

7. Attaque à clair et chiffré choisis (*Chosen plaintext and ciphertext attack*). Ce modèle d'attaque, suppose que l'adversaire a la possibilité de chiffrer et de déchiffrer les messages souhaités. Ce cas revient à avoir à sa disposition une boîte noire qui chiffre et qui déchiffre. Comme pour les attaques précédentes, cette attaque peut être adaptative ou non.

Ces différents modèles permettent de créer des classes d'attaques distinctes, qui doivent être envisagées lors du déploiement des cryptosystèmes. Dans tous les cas, comme mentionné précédemment, en suivant les principes de Kerckhoffs, l'attaquant a toujours la possibilité de connaître le mécanisme de chiffrement.

Une fois les échantillons obtenus, l'attaquant essaye de récupérer la clé du système ou de récupérer le message clair (en entier ou non) s'il ne possède que des chiffrés. Si cela est le cas, l'algorithme proposé est considéré comme sûr, sinon il est considéré comme cassé. En outre, après avoir effectué l'attaque, son succès sera mesuré en fonction des ressources qu'elle consomme. Donc, pour évaluer la performance d'une attaque, nous évaluons la complexité :

1. en temps : le temps nécessaire pour que l'attaque aboutisse. Parfois, par abus, la complexité en temps est la seule exigence.

2. en données : le nombre de messages chiffrés nécessaires à l'attaque. Une attaque peut potentiellement ne pas coûter cher en temps. Cependant, il est possible, qu'elle nécessite un très grand nombre de données. Si le temps nécessaire pour générer ces données dépasse une utilisation normale du système, l'attaque ne peut pas être considérée comme pratique.

3. en mémoire : la quantité d'espace nécessaire pour le stockage des données utilisées. Comme pour la complexité en données, une attaque qui nécessite un très grand espace de stockage, peut ne pas être considérée comme pratique.

L'attaque la plus naïve pour un adversaire est la recherche exhaustive de la clé. Peu importe le niveau de sécurité des briques sur lesquelles la conception de l'algorithme a été basée, l'attaquant a toujours la possibilité de faire ce type d'attaque et la seule façon de se protéger contre elle est d'avoir une clé assez grande. Donc, lors de la conception des algorithmes de chiffrement, le but du cryptographe est de construire un algorithme pour lequel la meilleure attaque dont dispose l'adversaire est la recherche exhaustive de la clé.

1.3.3 Modes opératoires des algorithmes de chiffrement par blocs

En pratique, les messages à chiffrer font rarement la taille du bloc imposée par les algorithmes de chiffrement par blocs. Dans le cas où le message à chiffrer est plus petit que la taille du bloc, il faut ajouter du *padding* pour compléter le message. Dans le cas contraire, si le message est plus long que le bloc utilisé par l'algorithme de chiffrement, il faut lier les blocs de messages clairs et chiffrer les uns avec les autres dans un mode opératoire.

Définition 1.5. *Un mode opératoire définit comment construire, à partir d'une fonction qui opère sur des blocs de taille fixe, une transformation qui permet de chiffrer des messages de longueur arbitraire.*

Juste après la standardisation de l'algorithme DES, quatre modes opératoires ont été définis pour DES dans FIPS 81. Les modes décrits dans ce document sont toujours les modes le plus connus et utilisés. De façon similaire, après que l'AES a été standardisé, le NIST a publié des règles concernant son utilisation (SP800-38A) [NIS01]. Ces recommandations incluent les quatre modes précédents et définissent en plus un cinquième mode. Au total, ces cinq modes sont :

1. *Electronic Code Book Mode (ECB)*,

2. *Cipher Block Chaining Mode (CBC)*,
3. *Cipher Feedback Mode (CFB)*,
4. *Output Feedback Mode (OFB)*,
5. *Counter Mode (CTR)*.

Les cinq modes définis ci-dessus peuvent être utilisés avec tous les algorithmes de chiffrement par blocs. Cependant, tous les modes opératoires n'apportent pas les mêmes propriétés de sécurité à l'algorithme de chiffrement utilisé, ni les mêmes performances. Par exemple, quand le mode opératoire ECB est utilisé, chaque bloc de message à chiffrer est traité indépendamment des autres par la fonction de chiffrement. Il s'agît de la façon la plus simple de procéder. Ccependant, ce mode ne devrait jamais être utilisé car deux blocs identiques donnent toujours le même chiffré. Cela n'est pas le cas des autres modes opératoires. Dans le cas du CBC par exemple, le chaînage des blocs à chiffrer utilisé crée une dépendance entre un bloc chiffré et tous les blocs chiffrés précédents. Il est donc conseillé de ne pas utiliser ECB mais plutôt un des autres modes proposés.

En outre, ces cinq modes garantissent la confidentialité d'un message, mais pas l'intégrité. Néanmoins, il existe des modes opératoires qui garantissent à la fois la confidentialité et l'authenticité du message à envoyer. Ces modes sont appelés modes de chiffrement authentifié (*Authenticated Encryption (AE)*). Dans cette catégorie, nous distinguons les modes de chiffrement authentifié à une passe, comme par exemple IAPM [Jut00] et OCB [RBB03], et les modes de chiffrement authentifié à deux passes, par exemple CCM [NIS04], EAX [BRW04] et GCM [NIS07].

Paramétrage par un IV. Les modes opératoires CBC, CFB, OFB et CTR nécessitent tous l'utilisation d'une valeur d'initialisation (*Initial Value (IV)*). La façon de choisir l'IV varie en fonction des besoins. Par exemple, le mode CBC est sûr quand l'IV est choisi au hasard, mais ne l'est pas quand elle correspond à un compteur comme c'est le cas du mode CTR. Par ailleurs, pour les modes OFB et CTR, l'utilisation de la même clé combinée avec la même IV donnera la même *keystream* et cela a des impacts sur la sécurité du schéma.

Le remplissage d'un bloc (*padding*). Les modes CFB et CBC nécessitent que la taille du message à chiffrer soit un multiple de la taille du bloc que l'algorithme de chiffrement utilise. Donc, afin d'obtenir la taille entière du bloc, nous appliquons une transformation appelée *padding*. Il existe plusieurs façons de compléter un message mais le choix de ce padding n'est pas anodin pour la sécurité [Vau02, BU02, PY04]. Un padding couramment employé est la méthode suivante. Le bit 1 est d'abord ajouté au message, suivi d'un nombre de bits de 0. Le nombre de 0 ajoutés est choisi de telle sorte que le message final obtenu ait une taille multiple de la taille d'un bloc mais qu'il soit en même temps le plus petit possible.

1.4 Les algorithmes de code d'authentification de message (MAC)

Comme nous l'avons vu précédemment, les algorithmes de chiffrement par bloc peuvent garantir la confidentialité d'un message mais aussi son intégrité si un mode opératoire adéquat est utilisé. Néanmoins, pour s'assurer de l'intégrité d'un message un autre composant cryptographique est souvent utilisé : les codes d'authentification de message, plus connus sous l'acronyme MAC (*Message Authentication Code*).

Définition 1.6. *Un code d'authentification de message, ou MAC, est une famille de fonctions de hachage paramétrées par une clé secrète. Le but d'un MAC est d'authentifier l'origine des données parmi les détenteurs de la clé et garantir leur intégrité. Pour cela, il prend en entrée un message m et une clé secrète K de k bits et donne en sortie un tag τ qui est habituellement annexé à m.*

1.4.1 Propriétés de sécurité d'un MAC

Dans ce type de schéma, le but de l'attaquant est de produire un MAC valide et de le faire passer pour légitime. Formellement, nous pouvons distinguer deux façons différentes pour un adversaire de procéder.

1. *Attaque de récuperation de la clé.* Ici, l'attaquant essaye de retrouver la clé secrète et, dans ce cas, nous parlons de cassage total du schéma. Plus précisément, étant donné un ensemble de couples *(messages,tags)* $(m_1, \tau_1), \ldots, (m_n, \tau_n)$, pour un ensemble de messages $m_1, \ldots, m_n$ qui peut être choisi de façon adaptative par l'attaquant, il ne doit pas être possible de récupérer la clé K qui a été utilisée pour produire les tags.

2. *MAC forgery*. Dans le deuxième cas, étant donné un ensemble de couples *(messages, tags)* $(m_1, \tau_1), \ldots, (m_n, \tau_n)$, pour un ensemble de messages $m_1, \ldots, m_n$ qui peut être choisi de façon adaptative par l'attaquant, il ne doit pas être possible pour l'adversaire de calculer un couple *message-tag* $(m^{'}, \tau^{'})$ valide pour un nouveau message $m^{'}$.

Il est évident que le première type d'attaque est plus fort que le deuxième : un adversaire qui arrive à récupérer la clé utilisée, est ensuite capable de forger le MAC de n'importe quel message. Cependant, dans certains schémas, l'attaquant peut forger un tag pour un nouveau message sans avoir aucune information sur la clé K.

Concernant la complexité de ces propriétés, pour la première, le but des concepteurs d'un MAC est d'assurer qu'un attaquant ne puisse pas récupérer la clé en faisant moins que 2^k opérations où k est la taille de la clé K utilisée. Pour la deuxième, si le tag τ a t bits, la probabilité de forger un MAC valide pour un message doit être toujours de 2^{-t}. En outre, parmi $2^{t/2}$ tags, d'après le paradoxe des anniversaires, il y a une collision entre deux d'entre eux. Ces collisions peuvent souvent être exploitées pour forger

un MAC. Cependant, ce type d'attaques peut souvent être évité en produisant un tag suffisamment long.

Pour atteindre son but, l'attaquant peut faire soit une attaque à messages connus, c'est-à-dire qu'il a déjà accès à des couples (m, τ) de messages déjà authentifiés, soit une attaque à messages choisis, c'est-à-dire qu'il demande le MAC de messages qu'il choisit à un oracle de génération de MACs.

1.4.2 Construction d'un MAC

Généralement, les MAC sont construits à partir d'autres primitives cryptographiques. Les trois principales façons de faire cela utilisent :

1. des *fonctions de hachage cryptographiques*,
2. du *hachage universel*,
3. ou un *algorithme de chiffrement par blocs*.

Exemples de MAC utilisés. Dans la catégorie des MAC basé sur des fonctions de hachage cryptographiques, l'algorithme plus connu est HMAC (*Hash-based Message Authentication Code*) [KBC97]. HMAC est définit comme suit :

$$h(K \oplus opad || h(K \oplus ipad || m))$$

où h est une fonction de hachage cryptographique, K la clé utilisée, m le message à authentifier, $ipad$ l'octet `0x36` répété $n/8$ fois (avec n la taille de la sortie de la fonction h) et $opad$ l'octet `0x5C` répété $n/8$ fois.

Plusieurs MAC utilisent du hachage universel, par exemple UMAC [BHK+99] et VMAC [Kro06]. L'idée est ici de choisir une fonction de hachage au sein d'une famille de fonctions. Le choix de la fonction ne dépend que de la clé.

Finalement, les MAC basés sur les algorithmes de chiffrement par blocs sont généralement présentés comme un mode opératoire. Le plus connu est le CBC-MAC (figure 1.4) et son fonctionnement est très similaire au mode opératoire CBC.

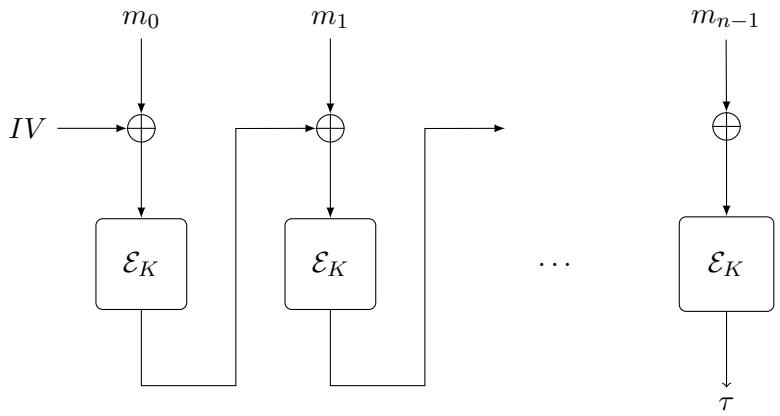

FIGURE 1.4 – Schéma du code d'authentification de message CBC-MAC.

En fait, CBC propage les erreurs lors du chiffrement, Il peut donc être utilisé pour vérifier l'intégrité de données. CBC-MAC consiste à chiffrer le message rempli avec un 1 puis des 0 et utilise 0 comme IV. Cette façon de procéder est considérée comme sûre pour générer des MAC pour des messages de taille fixe, mais elle ne l'est pas pour des messages de taille variable, même en ajoutant la longueur du message dans le dernier bloc [BKR94, PvO95]. Cependant, il existe plusieurs variantes qui résolvent les problèmes de sécurité du CBC-MAC : OMAC [IK03], PMAC [Rog00].

1.5 Les algorithmes de chiffrement à bas coût

Au cours des dernières années, les algorithmes cryptographiques sont de plus en plus utilisés dans différents systèmes informatiques. L'utilisation des algorithmes standardisés et bien étudiés, comme par exemple l'AES, est conseillée pour garantir un bon niveau de sécurité à ces systèmes. En même temps, la cryptographie est aussi largement utilisée dans des systèmes peu puissants, comme les systèmes embarqués, les capteurs sans fils, les dispositifs médicaux, les étiquettes RFID, etc. Cependant, ce type d'environnement ne disposent que de très peu de ressources et ne permettent pas l'utilisation de ces algorithmes standardisés. En effet, la majorité des algorithmes cryptographiques actuellement utilisés ne peuvent pas être adaptés pour des systèmes peu puissants. De nouveaux algorithmes, qui seront dédiés à ce type de systèmes, doivent donc être développés. Cette catégorie d'algorithmes appartient au domaine des algorithmes cryptographiques à bas coût (*lightweight cryptography*).

Les algorithmes à bas coût doivent donc répondre à des contraintes spécifiques comme un petit espace de stockage et une faible capacité de calcul, afin de leur permettre de s'exécuter rapidement sur un microprocesseur ou sur un petit circuit électronique. En outre, chaque nouveau schéma cryptographique à bas coût doit faire face à des compromis entre la sécurité, le coût et la performance. En règle générale, il est plus

simple de combiner deux de ces trois objectifs : sécurité et coût, sécurité et performance ou coût et performance. Cependant, il reste très difficile de combiner les trois objectifs en même temps.

Il est évident que le monde industriel est le principal intéressé pour le développement des algorithmes à bas coût. Cette demande a conduit à la conception d'un grand nombre de tels algorithmes. Dans ce cadre, de nombreux algorithmes de chiffrement par blocs ont été proposés, comme par exemple PRESENT [BKL+07], LED [GPPR11], Klein [GNL11], SIMON [BSS+13], SPECK [BSS+13], CLEFIA [SSA+07], KATAN et KTANTAN [CDK09], etc. Une preuve de l'intérêt de l'industrie pour ce type de construction est la proposition de CLEFIA par Sony[1] mais aussi sa standardisation, avec celle de l'algorithme PRESENT[2], par l'ISO/IEC Standard $29192-2$. Cependant, à cause de toutes ces contraintes du domaine, ces nouvelles constructions se basent sur des composants plus simples que d'habitude avec des marges de sécurité beaucoup plus faibles.

Par exemple, la majorité des algorithmes de chiffrement à bas coût utilisent des clés de chiffrement de petite taille. Comme il a déjà été mentionné dans ce chapitre, les clés de taille faible en cryptographie sont très vulnérables car un attaquant peut simplement faire une recherche exhaustive sur la clé.

En outre, en ce qui concerne les algorithmes de chiffrement par blocs à bas coût, leur majorité opère sur une taille de bloc de 64 bits. Par opposition, les algorithmes standards comme l'AES utilisent des blocs de 128 bits. Cela résulte en la dégradation du niveau de sécurité offert par le mode opératoire utilisé. Plus précisément, de nombreux modes, comme par exemple CBC, ne sont sûrs que s'ils traitent au maximum $2^{b/2}$ blocs de messages clairs, où b est la taille en bits du bloc. Il est donc évident que pour un mécanisme de chiffrement utilisant des blocs de 64 bits, cette limite peut être rapidement atteinte (32 Go).

De plus, les limitations que la cryptographie à bas coût impose a aussi un impact sur la construction des fonctions internes des algorithmes. Par exemple, les boîtes-S qui garantissent la non linéarité, doivent toujours avoir de bonnes propriétés cryptographiques. Néanmoins, dans ce cadre, elles doivent également posséder une faible complexité d'implémentation. Une façon d'accomplir cela serait de construire des boîtes-S qui opèrent sur moins de bits. Ces constructions sont donc plus risquées.

Plusieurs questions se posent naturellement sur l'utilisation de ces algorithmes. Est-ce qu'il faut affaiblir le niveau de sécurité des algorithmes afin de pouvoir construire des algorithmes très rapides qui peuvent être déployés dans de petits circuits ? Pour pouvoir répondre à cela, l'étude de ces primitives est une nécessité. En outre, l'étude de ces algorithmes et les attaques potentiellement proposées ne doivent pas seulement correspondre à un contexte théorique mais surtout représenter les défis et les problématiques d'un contexte réel.

[1] http ://www.sony.net/Products/cryptography/clefia

[2] http ://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm ?csnumber=56552

2. Recherche des collisions et techniques de cryptanalyse

Ce chapitre est consacré aux techniques de recherche de collisions et à la façon dont elles peuvent être utilisées dans un contexte cryptographique. Mathématiquement, nous définissons une collision comme le fait que deux entrées x et y distinctes d'une fonction f aient des images identiques c'est-à-dire $f(x) = f(y)$.

En cryptographie, les collisions jouent un rôle très important. Plus précisement, la recherche des collisions est un outil très important en cryptanalyse car plusieurs problèmes, comme par exemple le problème du logarithme discret, la sécurité des fonctions de hachage et les attaques *meet-in-the-middle* peuvent être réduits à la difficulté d'en trouver. Il est donc très important de connaître le nombre d'objets dont nous

avons besoin d'avoir dans un ensemble pour détecter une collision avec une probabilité plus grande que $1/2$. La réponse à cette question est donnée par le paradoxe des anniversaires qui nous dit que si nous choisissons k éléments dans un ensemble de taille N, nous nous attendons à avoir deux éléments identiques à partir du moment où $k \approx \mathcal{O}(\sqrt{N})$.

Nous commençons ce chapite en expliquant des résultats génériques concernant l'existence de collisions dans certains contextes en cryptographie. Ensuite, nous nous intéressons à un sujet plus spécifique : aux collisions générées après avoir itéré une fonction aléatoire plusieurs fois. En outre, nous décrivons certains algorithmes plus efficaces que les méthodes génériques pour détecter ces collisions. Finalement, nous nous intéressons aux algorithmes de recherche parallélisables de collisions qui réduisent la complexité en temps de plusieurs attaques cryptographiques et nous présentons l'idée du compromis temps-mémoire d'Hellman.

Pour ce chapitre, nous nous basons sur le livre de *Handbook of Applied Cryptography* [MVO96] de Menezes, Vanstone et van Oorschot et sur le livre *Algorithmic Cryptanalysis* [Jou09] d'Antoine Joux.

2.1 Les collisions en cryptographie

Nous présentons ici quelques résultats génériques concernant l'existence de collisions. Ensuite, nous donnons quelques exemples pour montrer comment l'existence de collisions peut détériorer la sécurité de certains cryptosystèmes.

Paradoxe des anniversaires. Le paradoxe des anniversaires est une propriété mathématique qui limite la résistance en collision d'une fonction aléatoire. Ce paradoxe désigne une propriété contre-intuitive : parmi un groupe d'au moins 23 personnes prises au hasard, il y a plus d'une chance sur deux pour que deux d'entre elles aient leur anniversaire le même jour. Cette propriété peut être expliquée par le raisonnement suivant.

Soit k éléments $x_1, \ldots, x_k$ tirés indépendamment et uniformément dans un ensemble de taille N. Parmi ces k éléments, nous pouvons choisir $k(k-1)/2$ paires (x_i, x_j). En outre, pour chaque paire, la probabilité d'avoir une collision ($x_i = x_j$) est N^{-1}. Si tous les paires sont indépendantes, il est possible de prouver que pour $k \geq \sqrt{N}$, la probabilité d'avoir une collision vaut :

$$\frac{k^2 - k}{2N} >\approx \frac{1}{2}.$$

Collisions entre deux ensembles et multi-collisions. Nous donnons ici deux résultats concernant la détection d'une collision entre deux ensembles séparés et la détection de multi-collisions.

Pour le cas de recherche d'une collision dans un ensemble nous avons vu que nous pouvons utiliser le paradoxe des anniversaires. Maintenant, supposons que nous ayons un ensemble de taille N et deux sous-ensembles distincts de taille k_1 et k_2. Notre but

est de trouver une collision entre les deux sous-ensembles. À partir de ces éléments, nous pouvons construire $k_1 \cdot k_2$ paires et donc nous pouvons avoir au total $k_1 \cdot k_2/N$ collisions.

Nous définissons une ℓ multi-collision comme une collision multiple survenue entre ℓ éléments. Dans le cas où nous cherchons une ℓ multi-collision entre k éléments dans le même ensemble de taille N, nous voyons que le nombre de multi-collisions attendues est :

$$\frac{k^\ell}{\ell! \cdot N^{\ell-1}}.$$

En outre, si nous cherchons une ℓ multi-collision entre plusieurs sous-ensembles de taille $k_1, \ldots, k_\ell$, le nombre de multi-collisions attendues est :

$$\frac{\prod_{i=1}^{\ell} k_i}{N^{\ell-1}}.$$

Résistance en collision de fonctions de hachage. Comme expliqué au chapitre 1, les fonctions de hachage sont utilisées pour calculer des empreintes de taille fixe à partir de données de taille arbitraire. Les fonctions de hachage sont utilisées, entre autres, en tant qu'identifiant unique pour les signatures *Hash-and-Sign*. En effet, pour que les opérations soient moins coûteuses, les schémas de signature, comme par exemple RSA, nécessitent l'application de la signature à un haché du message au lieu de l'appliquer directement au message. Ainsi, si nous ne voulons pas que le signataire puisse répudier ses signatures, la fonction de hachage doit être résistante aux collisions.

Selon la définition de la fonction de hachage, l'ensemble de départ est plus grand que l'ensemble d'arrivée. L'existence de collisions est alors inévitable pour une fonction de hachage. La résistance à la recherche de collisions ne peut donc pas être définie comme l'impossibilité mathématique de l'existence des collisions, mais comme la difficulté d'en trouver en pratique. Cette difficulté est évaluée par rapport au nombre d'opérations que l'attaquant doit faire.

La meilleure attaque générique pour chercher des collisions d'une fonction de hachage repose sur le paradoxe des anniversaire. Si la taille de l'espace d'arrivée (espace de hachés) est 2^n, d'après ce paradoxe, le nombre de hachés qu'il faut calculer pour trouver une collision est égal à $2^{n/2}$.

Collisions sur les modes CBC et CFB. L'existence de collisions joue aussi un rôle très important dans le fonctionnement des modes opératoires d'algorithmes de chiffrement par blocs. Les modes opératoires de chiffrement permettent d'utiliser un algorithme de chiffrement par blocs pour chiffrer des messages de longueur quelconque. Tout d'abord, un mode opératoire doit être efficace. En outre, il ne doit pas affaiblir l'algorithme de chiffrement par blocs utilisé. Il est alors très important d'étudier la sécurité des modes opératoires indépendamment de l'algorithme de chiffrement par blocs sous-jacent.

Les modes opératoires CBC et CFB révèlent de l'information sur le message clair si l'attaquant observe une collision entre deux entrées de l'algorithme de chiffrement par blocs. Pour expliquer l'attaque par collisions sur CBC et CFB, nous notons P_i le $i$ème bloc clair du message, I_i l'entrée de l'opération du chiffrement correspondant et C_i le chiffré correspondant.

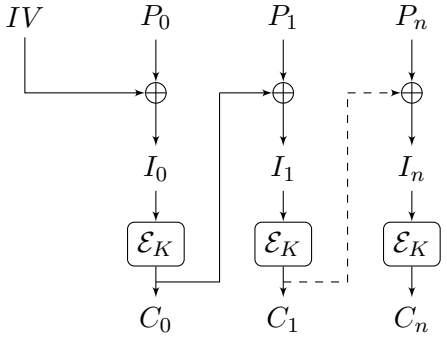

FIGURE 2.1 – Schéma du mode CBC.

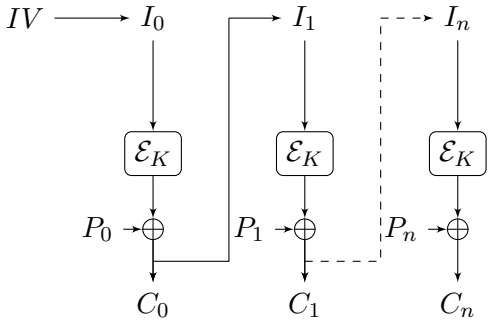

FIGURE 2.2 – Schéma du mode CFB.

Pour le mode CBC, si nous avons une collision entre deux blocs du message chiffré nous savons qu'il y a aussi collision entre les I correspondants, *i.e.* pour $i \neq j$ si $C_i = C_j$ nous savons que $I_i = I_j$. Pour CFB, une collision entre deux entrées à l'algorithme de chiffrement, nous révèle une collision entre les chiffrés des blocs précédents, *i.e.* pour $i \neq j$ si $I_i = I_j$ nous déduisons que $C_{i-1} = C_{j-1}$. Dans les deux cas, si une telle collision est détectée, l'attaquant récupère une relation entre les messages clairs :

$$P_i \oplus P_j = \Delta_{ij} \text{ où } \Delta_{ij} = C_{j-1} \oplus C_{i-1} \text{ (cas mode CBC)}$$
$$P_i \oplus P_j = \Delta_{ij} \text{ où } \Delta_{ij} = C_j \oplus C_i \text{ (cas mode CFB),}$$

Nous en déduisons que si l'attaquant connait une partie des messages clairs, cette relation lui révèle de l'information supplémentaire. Donc, si l'attaquant connait P_i, il récupère P_j.

Cette attaque suppose que tous les blocs ont été chiffrés avec la même clé. Par le paradoxe des anniversaires, nous aurons une collision entre deux blocs après avoir chiffré $2^{n/2}$ blocs, où n est la taille de bloc utilisé. Pour se protéger donc contre ce type d'attaque il est indispensable d'utiliser un bloc suffisamment grand et de changer régulièrement la clé utilisée.

Collisions sur les MACs. Le code d'authentification de message CBC-MAC a été présenté au chapitre 1. Pour calculer le CBC-MAC d'un message $m = m_1, m_2, \ldots m_n$ en utilisant la clé K nous calculons :

$$c_0 = 0^n \text{ et } c_i = E_K(m_i \oplus c_{i-1}) \text{ pour } i = 1, 2, \ldots, n.$$

La valeur donc du tag t calculé est $CBC_{E_K}(m) = c_n$.

Il a été prouvé par Bellare, Killian et Rogaway [BKR94] que CBC-MAC est sûr pour des messages de taille fixe. Cependant, CBC-MAC est vulnérable aux attaques basées sur le paradoxe des anniversaires. En effet, il a été montré par Prennel et van Oorschot [PvO95] que tous les MACs qui sont basés sur les fonctions itérées sont vulnérables à ce type d'attaques.

L'idée principal de ces attaques est de trouver deux messages m et $m^{'}$, d'un seul bloc chacun, qui donnent la même valeur MAC, *i.e.* $CBC_{E_K}(m) = CBC_{E_K}(m^{'}) = t_1$. Grâce au paradoxe des anniversaires, nous savons que nous trouvons deux tels messages après avoir calculé les tags de $2^{n/2}$ messages. Ensuite, nous ajoutons au message m une chaîne fixe a et nous calculons $CBC_{E_K}(m||a) = t_2$. Il en découle que $CBC_{E_K}(m^{'}||a) = t_2$. Par conséquence, nous obtenons une attaque *forgery* où la requête du tag d'un message nous permet de calculer le tag (qui est le même) d'un message différent. Pour les algorithmes de chiffrement par bloc qui utilisent des blocs de 64 bits, par le paradoxe des anniversaires, cette attaque devient effective après avoir calculé les codes d'authentification de 2^{32} messages. Afin d'atteindre une plus grande sécurité, il est possible d'utiliser des algorithmes de chiffrement par blocs qui utilisent de blocs d'au moins 128 bits (par exemple l'AES) ou des algorithmes MAC qui sont sûrs au-delà du paradoxe des anniversaires.

2.2 Collisions générées par des fonctions itérées

Nous nous intéressons ici aux collisions produites en utilisant des fonctions fixes itérées. L'exploitation de ce type de collisions peut avoir plusieurs applications en cryptographie symétrique mais aussi asymétrique. Par exemple, nous verrons dans la suite les algorithmes de Pollard qui appliquent ces techniques au problème du logarithme discret et à la factorisation.

2.2.1 Schéma général

Nous présentons ici le schéma général des collisions obtenues à partir de l'itération d'une fonction fixe. Le but ici est de générer d'abord une suite en utilisant la fonction fixe et ensuite de détecter une collision dans cette suite. Pour cela, soit un ensemble $\mathcal{S}$ et f une fonction définie sur cet ensemble, *i.e.* $f : \mathcal{S} \to \mathcal{S}$. Nous choisissons un point de départ aléatoire x_0 qui appartient à $\mathcal{S}$ et nous générons une suite (x_i) de la manière suivante :

$$x_{i+1} = f(x_i) \text{ pour } i = 0, 1, 2, \ldots.$$

Une collision est détectée si, pour $i \neq j$, $x_i = x_j$. Évidemment, si nous continuons à itérer la fonction f, les deux suites resteront égales car :

$$x_{i+1} = f(x_i) = f(x_j) = x_{j+1}.$$

Après avoir itéré la fonction f plusieurs fois, nous attendons que la suite reboucle sur une de ses valeurs. La figure 2.3 montre la procédure de génération de la suite (x_i) et la détection d'une collision.

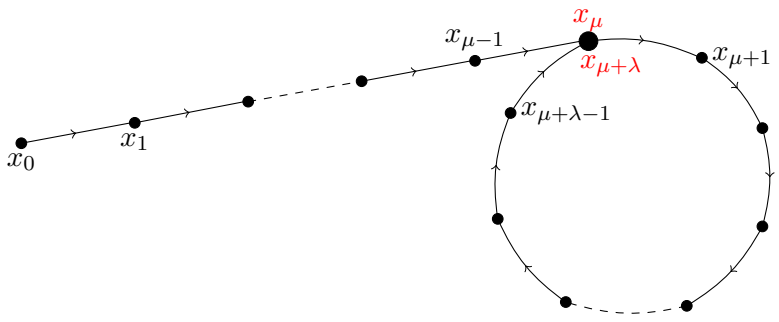

FIGURE 2.3 – Schéma du cycle ρ.

Nous remarquons que, comme le montre la figure, si nous souhaitons visualiser la génération de la suite, nous récupérons un schéma qui ressemble à la lettre grecque ρ. Nous distinguons deux parties : la queue et le cycle. Soit x_μ le point de collision. La queue contient donc tous les points de x_0 à x_μ et μ est la longueur de la queue. En outre, soit λ la longueur du cycle, *i.e.* le nombre d'itérations de la fonction f pour le parcours du cycle. Le cycle contient donc tous les points de x_μ à $x_{(\mu+\lambda-1)}$.

Si f était une permutation elle rebouclerait obligatoirement sur une de ses valeurs et son illustration ressemblerait à un cycle. Par contre, dans le cas d'une fonction, la suite recommence, en général, à un autre point. Pour mieux visualiser le fonctionnement, nous représentons le comportement de la fonction par un graphe orienté $\mathcal{G}_f$. Des définitions élémentaires sur les graphes sont données à la section 2.2.2. Cependant, nous montrons ici un exemple de représentation d'une fonction aléatoire par un graphe. Tous les éléments de $\mathcal{S}$ sont représentés par les nœuds du graphe et les arêtes donnent la valeur correspondante de la fonction f. Pour notre exemple, nous supposons que l'ensemble $\mathcal{S}$ contient les éléments $\{0, \ldots, 9\}$ et que la table ci-dessous nous donne les valeurs correspondantes de la fonction f :

x	0	1	2	3	4	5	6	7	8	9
$f(x)$	2	4	6	5	8	7	7	3	8	2

La figure 2.4 montre la forme du graphe de cette fonction :

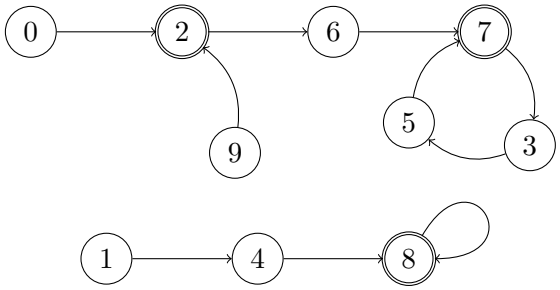

FIGURE 2.4 – Graphe du comportement de la fonction aléatoire f.

2.2.2 Analyse de fonctions aléatoires

Pour analyser les propriétés de ces suites, la fonction f est généralement modélisée comme une fonction aléatoire. Nous donnons ici quelques propriétés des fonctions aléatoires qui nous seront utiles pour la suite. L'ensemble de ces propriétés est basé sur l'analyse de Flajolet et Odlyzko [FO89] concernant les fonctions aléatoires représentées par un graphe orienté et leurs applications en cryptographie. Cependant, pour faciliter la lecture, nous donnons tout d'abord quelques définitions élémentaires de la théorie des graphes.

Notions de base sur les graphes

Nous distinguons deux catégories des graphes : les graphes non-orientés et les graphes orientés. Les termes utilisés pour les deux catégories sont presque identiques.

Définition 2.1. *Un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est défini par :*

- *un ensemble $\mathcal{S}$ (fini ou non) d'objets appelés nœuds ou sommets,*
- *et un ensemble fini $\mathcal{A}$ d'arêtes dont chacune relie un couple de sommets.*

Si s_i et s_j sont deux nœuds, nous noterons (s_i, s_j) l'arête qui a pour origine (extrémité initiale) le nœud s_i et pour extrémité terminale le nœud s_j. Si les arêtes $\{s_i, s_j\}$ et $\{s_j, s_i\}$ sont les mêmes pour tous les nœuds s_i et s_j, nous disons que le graphe n'est pas orienté et que s_i et s_j sont dit adjacents. Au contraire, si $\{s_i, s_j\}$ et $\{s_j, s_i\}$ ne sont pas les mêmes, nous disons que le graphe est orienté. Dans ce cas, l'arête (s_i, s_j) est dite sortante en s_i et incidente en s_j. De plus, s_j est un successeur de s_i, tandis que s_i est un prédécesseur de s_j.

La représentation d'un graphe orienté est définie comme en non-orienté avec simplement en plus une flèche sur chaque arête qui indique le sens du premier nœud vers le second. Dans notre cas, comme l'ordre sur les extrémités d'une arête est importante, nous nous intéressons ici aux graphes orientés. Les définitions ci-dessous concernent donc surtout ce type de graphes.

Définition 2.2. *Une boucle est une arête reliant un nœud à lui-même.*

Définition 2.3. *Nous appelons ordre d'un graphe le nombre de ses nœuds,* i.e. *card*$(\mathcal{S})$.
Nous appelons taille d'un graphe le nombre de ses arêtes, i.e. *card*$(\mathcal{A})$.

Définition 2.4. *Le degré entrant ou demi-degré intérieur d'un nœud est le nombre d'arêtes dirigées vers ce nœud. Au contraire, le degré sortant ou demi-degré extérieur d'un nœud est le nombre d'arêtes sortant de ce nœud.*
Nous appelons degré d'un nœud la somme du degré entrant et du degré sortant.

Définition 2.5. *Un chemin est une séquence finie et alternée de nœuds et d'arêtes, débutant et finissant par des nœuds, telle que chaque arête est sortante d'un nœud et incidente au nœud suivant dans la séquence.*
Un circuit ou un cycle est un chemin dont les extrémités coïncident.

Dans la suite de cette thèse, nous nous intéresserons aussi aux graphes connexes. De manière intuitive, la notion de connexité est triviale. Un graphe est connexe si nous pouvons atteindre n'importe quel nœud à partir d'un nœud quelconque en parcourant différentes arêtes.

Définition 2.6. *Un graphe $\mathcal{G}$ non orienté est dit connexe si deux nœuds quelconques sont reliés par un chemin. Un graphe $\mathcal{G}$ orienté est connexe si son graphe non orienté sous-jacent est connexe.*

Cela veut dire que pour deux nœuds s_i et s_j d'un graphe $\mathcal{G}$, il existe une suite d'arêtes qui permet d'accéder à s_i à partir de s_j.

Définition 2.7. *Un graphe $\mathcal{G}$ orienté est fortement connexe si pour tout couple de noeuds distincts s_i, s_j il existe un chemin orienté de s_i à s_j et un de s_j à s_i.*

Définition 2.8. *Une composante connexe $\mathcal{C}$ d'un graphe $\mathcal{G} = \{\mathcal{S}, \mathcal{A}\}$ est un sous-ensemble maximal de nœuds tels que deux quelconques d'entre eux soient reliés par un chemin : si $s_i \in \mathcal{C}$ alors*

- $\forall s_j \in \mathcal{C}$*, il existe un chemin reliant s_i et s_j*
- $\forall s_j \in \mathcal{S} \setminus \mathcal{C}$*, il n'existe pas de chemin reliant s_i et s_j.*

En outre, nous donnons la définition des arbres qui sont des graphes particulièrement importants.

Définition 2.9. *Un arbre est un graphe non orienté connexe sans cycles.*

Pour conclure la section des définitions, nous avons besoin pour la suite de définir la composante géante.

Définition 2.10. *La composante géante du graphe $\mathcal{G}$ est une composante du graphe qui regroupe la majorité des nœuds du graphe.*

Propriétés de fonctions aléatoires

Nous présentons ici, trois types de propriétés des fonctions aléatoires : les paramètres globaux, les paramètres locaux et les paramètres externes. Les propriétés globales correspondent aux propriétés générales du graphe. Au contraire, les propriétés locales nous donnent des caractéristiques du graphe à partir d'un point de départ aléatoire.

Comme présenté ci-dessus, chaque fonction aléatoire f dans un ensemble $\mathcal{S}$ de N éléments est représentée par un graphe $\mathcal{G}_f$ où tous les N éléments de $\mathcal{S}$ sont représentés par les nœuds du graphe et les arêtes donnent la valeur correspondante de la fonction f.

Paramètres globaux.

1. Un graphe $\mathcal{G}_f$ contient en moyenne $\log N/2$ composantes connexes.
2. Un nœud cyclique est un nœud qui appartient à un cycle du graphe. Le nombre moyen de nœuds cyclique dans un graphe est $\sqrt{\pi N/2}$.
3. Le nombre de nœuds terminaux que le graphe $\mathcal{G}_f$ contient vaut en moyenne à $e^{-1}N$.
4. Un point image de f est un point dans l'image de f. Le nombre des points images du graphe $\mathcal{G}_f$ est en moyenne $(1 - e^{-1})N$.
5. Un point s_i de $\mathcal{S}$ est appelé k^{e} point image itéré si et seulement s'il existe un autre point s_j de $\mathcal{S}$ tel que $s_i = f^{(k)}(s_j)$ où k est le nombre d'itérations de la fonction f. Le nombre de k^{e} points images itérés dans un graphe $\mathcal{G}_f$ est en moyenne $(1 - \tau_k^{-1})N$ où τ_k est défini par la récursivité : $\tau_0 = 0$ et $\tau_{k+1} = e^{\tau_k - 1}$.

Paramètres locaux.

1. Soit μ la longueur de la queue, *i.e.* la distance entre le point de départ et le cycle obtenu pendant l'itération de la fonction f. Alors μ vaut en moyenne $\sqrt{\pi N/8}$.
2. Soit λ la taille du cycle obtenu à partir d'un point de départ aléatoire en itérant la fonction f. Dans ce cas, λ est en moyenne $\sqrt{\pi N/8}$.
3. La longueur Rho ($\rho = \lambda + \mu$) vaut en moyenne $\sqrt{\pi N/2}$.
4. Nous considérons l'arbre qui se forme si nous prenons en compte tous les points qui entrent dans le même cycle par le même endroit. La taille moyenne de cet arbre est égale à $N/3$.
5. La taille de la composante connexe qui contient le point de départ est égale à $2N/3$.
6. En moyenne, le nombre de points qui sont des pré-images itérés de notre point départ, vaut $\sqrt{\pi N/8}$.

Paramètres externes.

1. Nous définissons le chemin le plus long (la plus longue queue avec le plus long cycle) comme le chemin qui se forme si nous choisissons le pire point de départ. La taille attendue du plus long chemin vaut en moyenne $c\sqrt{N}$ avec $c \approx 2.415$. En outre, la taille de la queue la plus longue est égale à $c_1\sqrt{N}$ avec $c_1 \approx 0.782$ et la taille du cycle le plus long est égale à $c_2\sqrt{N}$ avec $c_2 \approx 1.737$.

2. La taille moyenne de la composante géante du graphe est égale à $d_1 N$ avec $d_1 \approx 0.758$. De plus, la taille moyenne du plus grand arbre du graphe est égale à $d_2 N$ avec $d_2 \approx 0.48$.

2.3 Algorithmes de recherche des collisions

Plusieurs techniques de recherche des collisions existent. L'algorithme 1 présente une méthode générique qui ne nécessite aucune connaissance du fonctionnement de la fonction attaquée. Il fait appel à la fonction RECHERCHE qui cherche parmi les valeurs déjà obtenues la dernière valeur calculée. Si la même valeur est trouvée, la fonction RECHERCHE retourne l'index de la valeur déjà existante sur le tableau. Sinon, elle renvoie -1.

Algorithme 1 Recherche générique des collisions

ENTRÉES :
- $m \in \mathbb{N}$ avec $m > n$
- Un ensemble $\mathcal{X} : \{0,1\}^m$
- Une fonction aléatoire f

SORTIES :
- Deux valeurs $x_i, x_j \in \mathcal{X}$ telles que $x_i \neq x_j$ et $f(x_i) = f(x_j)$

$j \leftarrow -1$
tant que VRAI **faire**
  $x_i \leftarrow$ choisir aléatoirement dans $\{0,1\}^m$
  $y_i = f(x_i)$
  Stocker y_i dans $\mathcal{Y}$
  $j \leftarrow$ RECHERCHE(y_i)
  si $j \neq -1$ et $x_i \neq x_i$ **alors**
    retourne (x_i, x_j)
  fin si
fin tant que

Cependant, ce genre de techniques sont très longues à exécuter et elles utilisent une

très grande quantité de mémoire. À la suite donc de cette section, nous nous intéressons plutôt aux algorithmes qui utilisent de plus petites quantités de mémoire.

2.3.1 Algorithmes de détection de cycle

Nous avons vu jusqu'à maintenant comment nous pouvons représenter par un graphe les fonctions aléatoires ainsi que certaines de leurs propriétés dans le but de détecter les collisions qui ont été produites. Nous nous intéressons ici à la détection de ces collisions. Plus précisément, nous présentons quelques algorithmes connus qui nous permettent de détecter les collisions dans une suite et de déterminer la longueur de la queue et du cycle de cette suite.

Pour la suite de cette section, nous cherchons à trouver un cycle généré par la fonction $f : \mathcal{S} \to \mathcal{S}$, avec $\mathcal{S} = \{0, 1, 2, \ldots, N-1\}$, qui génère la suite $x_{i+1} = f(x_i)$.

L'algorithme de Floyd de détection de cycle

L'algorithme de Floyd de détection de cycle, encore appelé algorithme du lièvre et de la tortue, est un algorithme pour détecter les cycles dans des séquences arbitraires. Il a été décrit par Knuth [Knu97] mais attribué à Floyd. L'idée de base est qu'il n'est pas nécessaire de comparer chaque nouvelle valeur générée avec toutes les précédentes car cela serait équivalent de la recherche exhaustive. Son principe est donc de créer deux suites au lieu d'une qui progressent à des vitesses différentes.

Algorithme 2 Algorithme de Floyd de détection de cycle

ENTRÉES :

- Valeur d'initialisation X_0
- Nombre maximal d'itérations R

SORTIES :

- L'indice i telle que $x_i = y_i$

Soit $x \leftarrow X_0$
Soit $y \leftarrow X_0$
pour tout i de 1 à R **faire**
  $x \leftarrow f(x)$
  $y \leftarrow f((f(y))$
  si $x = y$ **alors**
    Collision trouvée entre i et $2i$
  fin si
fin pour
retourne Indice i

La première suite x, qui est plus lente comme la tortue, avance d'un élément à la

fois. Au contraire, la deuxième y, le lièvre, avance de deux éléments à chaque pas :

$$y_i = x_{2i}.$$

Plus précisément, la suite y peut être définie comme suit :

$$y_0 = x_0 \text{ et } y_{i+1} = f(f(y_i)).$$

Comme le montre le pseudo-code de l'algorithme 2, l'algorithme de Floyd est basé sur l'observation que, pour chaque entier $i \geq \mu$ et $k \geq 0$, nous avons que :

$$x_i = x_{i+k\lambda},$$

où λ est la longueur du cycle et μ l'indice du premier élément du cycle, c'est-à-dire la longueur de la queue.

Quand $i = k\lambda \geq \mu$, nous avons que $x_i = x_{2i}$. Ainsi, pour trouver ce i, nous initialisons les valeurs x, y et i et ensuite nous calculons les suites x et y en parallèle, en effectuant plusieurs évaluations de la fonction f, jusqu'à ce que nous constations que $x = y$.

Quand $x_i = x_{2i}$, i correspond au plus petit multiple de la longueur du cycle λ. En outre, il est égal ou supérieur à la longueur de la queue μ. L'algorithme de Floyd détecte qu'un cycle a été trouvé mais il ne retourne pas les valeurs λ et μ.

L'algorithme de Brent de détection de cycle

En 1980, Brent [Bre80] a proposé une optimisation de l'algorithme de Floyd. Cet algorithme est aussi linéaire, néanmoins il nécessite moins d'opérations que l'algorithme de Floyd qui, à chaque pas, fait trois évaluations de la fonction f. Cependant, il est un peu plus complexe à décrire.

Pour l'algorithme de Brent, nous considérons la séquence des points de départ $x_1, x_2, \ldots, x_{2^i}, \ldots$. Ces points de départ sont indexés par des puissances de deux. Pour chaque point de départ x_{2^i} nous calculons tous les points suivants x_{2^i+j}. Les calculs s'arrêtent quand nous trouvons soit un cycle soit un point $x_{2^{i+1}}$.

Dans le premier cas, nous ayons le résultat recherché. Dans le deuxième, nous recommençons en prenant comme point de départ $x_{2^{i+1}}$. Le pseudo-code de l'algorithme 3 montre ce fonctionnement.

Algorithme 3 Algorithme de Brent de détection de cycle

ENTRÉES :

- Valeur d'initialisation X_0
- Nombre maximal d'itérations R

Soit $x \leftarrow X_0$
Soit $y \leftarrow x$
Soit $trap \leftarrow 0$
Soit $nexttrap \leftarrow 1$
pour tout i de 1 à R **faire**
  $x \leftarrow f(x)$
  si $x = y$ **alors**
    Collision trouvée entre $trap$ et i
    Sortie
  fin si
  si $i = nexttrap$ **alors**
    $trap \leftarrow nexttrap$
    $nexttrap \leftarrow 2 \cdot trap$
    $y \leftarrow x$
  fin si
fin pour
Sortie échouée

L'algorithme de Nivasch de détection de cycle

En 2004 Nivasch [Niv04] propose un algorithme plus rapide pour la détection d'un cycle dans une suite. Le pseudo-code de l'algorithme 4 présente le fonctionnement de l'algorithme de Nivasch et il procède de la manière suivante.

Il utilise une pile qui est initialement vide et qui se remplit au fur et à mesure avec des paires (x_i, i) où x_i, qui correspond aux résultats de l'évaluation de la fonction f, et i sont deux séquences strictement croissantes. À chaque évaluation de la fonction f, si les premiers éléments de la pile sont plus grand que l'élément courant, nous faisons un pop de la pile des premiers éléments. Si nous détectons une collision dans la pile entre les éléments x_i et x_j, l'algorithme s'arrête et la longueur du cycle est $\lambda = j - i$.

L'algorithme détecte toujours la plus petite valeur du cycle. Une fois que cette valeur est ajoutée dans la pile, elle n'est jamais supprimée et l'algorithme arrêtera son exécution dès qu'il détecte cette valeur pour la deuxième fois.

Algorithme 4 Algorithme de Nivasch de détection de cycle

ENTRÉES :

- Valeur d'initialisation X_0
- Nombre maximal d'itérations R

$S \leftarrow [\,]$
Soit $x_0 \leftarrow X_0$
push$(S, (x_0, 0))$
pour tout i de 1 à R **faire**
  $x_i \leftarrow f(x_{i-1})$
  pour tout j de 1 à i **faire**
    si $x_i < x_j$ **alors**
      pop $S[j]$
    fin si
    si $x_i = x_j$ **alors**
      retourne $i - j$
    fin si
    push$(S, (x_i, i))$
  fin pour
fin pour

2.3.2 Algorithmes de récupération du début du cycle

Nous avons vu jusqu'à maintenant que les algorithmes de Floyd, Brent et Nivasch peuvent être utilisés pour détecter un cycle produit par l'itération de la fonction f. Éventuellement, en utilisant ces algorithmes, nous pouvons calculer un multiple de la longueur du cycle λ. Cependant, aucun d'entre eux ne nous donne μ qui est l'index du premier élément du cycle. Néanmoins, cet élément est très important car il nous révèle la vraie collision que nous cherchons.

L'algorithme 5 montre comment nous pouvons trouver le premier élément du début du cycle à partir du point de départ de la suite, la longueur du cycle et le nombre d'itérations de la fonction f. La première étape de cet algorithme consiste à trouver le dernier élément du cycle x_λ. Si nous détectons que x_λ est égal au point de départ x_0, nous en déduisons que la suite est cyclique. Cela veut dire qu'aucune collision existe dans cette suite et donc l'algorithme s'arrête avec un message d'erreur. Si la suite n'est pas cyclique, nous commençons le calcul de deux chaînes en parallèle : la première chaîne commence par le point de départ x_0 et la deuxième chaîne commence par le point x_λ. La vraie collision intervient après λ itérations de la fonction f, *i.e.* quand les deux suites vont collisionner.

Algorithme 5 Algorithme de récupération du premier élément du cycle

ENTRÉES :

- Valeur d'initialisation X_0
- Longueur du cycle λ
- Nombre maximal d'itérations R

SORTIES :

- Deux éléments x et y pour lesquels nous avons une collision

Soit $x \leftarrow X_0$
Soit $y \leftarrow X_0$
pour tout i de 1 à λ **faire**
  $y \leftarrow f(y)$
fin pour
si $x = y$ **alors**
  Erreur : aucune collision
  Sortie
fin si
pour tout i de 1 à R **faire**
  $x^{'} \leftarrow f(x)$
  $y^{'} \leftarrow f(y)$
  si $x^{'} = y^{'}$ **alors**
    Collision trouvée entre x et y
    Sortie
  fin si
  Soit $x \leftarrow x^{'}$
  Soit $y \leftarrow y^{'}$
fin pour

2.3.3 Recherche parallélisable des collisions

Comme nous l'avons expliqué au début de ce chapitre, même si aujourd'hui la puissance des ordinateurs augmente constamment, il est très utile de pouvoir paralléliser la recherche de collisions pour réduire la complexité en temps effective des algorithmes utilisés. Malheureusement l'algorithme de détection de cycle de Floyd ne peut pas se paralléliser efficacement. Si nous utilisons m processeurs, chaque processeur doit attendre que l'appel précèdent à la fonction f soit terminé pour pouvoir exécuter le prochain appel. Cela veut dire que si nous utilisons m processeurs, nous ne pouvons pas avoir une réduction de $1/m$ de la complexité de l'algorithme, *i.e.* la vitesse de l'algorithme n'augmente pas linéairement en fonction du nombre de processeurs utilisés.

Une autre méthode très efficace pour détecter des cycles est d'utiliser les points distingués. L'idée de l'utilisation de points distingués a été tout d'abord abordée par Ri-

vest [Den82] dans le contexte du compromis temps-mémoire de Hellman. Cependant, la première utilisation de points distingués dans un contexte pratique a été faite par Quisquater et Delescaille [QD89a, QD89b] pour proposer un algorithme de recherche de collisions pour DES. Ensuite, van Oorschot et Wiener [vOW99] proposent une méthode parallélisable de recherche des collisions basée sur les points distingués. Dans le même article, ils proposent des applications de cette méthode pour calculer les logarithmes discrets, attaquer des fonctions de hachage (MD5, RIPEMD, SHA-1, MDC-2 et MDC-4) et le double-DES.

Dans cette section, nous présentons tout d'abord la méthode de van Oorschot et Wiener [vOW99] de recherche parallélisable des collision basées sur les points distingués qui nous sera utile pour toutes les attaques présentées dans cette thèse. Finalement, dans la prochaine section, nous présentons quelques applications cryptographiques de méthodes de recherches de collisions.

Méthode de recherche parallélisable des collisions basée sur les points distingués

Nous présentons ici la méthode de recherche parallélisable des collisions basée sur les points distingués de van Oorschot et Wiener. La technique des points distingués est très efficace pour détecter des collisions. Cela consiste à stocker les valeurs qui satisfont une propriété particulière et à attendre qu'ils réapparaissent. Les points avec cette propriété particulière sont appelés les points distingués. Cependant, la principale difficulté ici est de choisir la bonne probabilité d'occurrence pour distinguer ces points. Cela veut dire que si les points distingués apparaissent très fréquemment, nous serons obligés de stocker un grand nombre de valeurs. Par conséquence, l'algorithme sera inefficace. Au contraire, il est possible que le cycle ne contienne aucun point distingué si ceux-ci sont trop rares. Dans ce cas, l'algorithme échouera.

Comme précédemment, nous cherchons des collisions pour une fonction f définie sur un ensemble $\mathcal{S}$ avec N éléments. En outre, nous définissons un ensemble $\mathcal{S}_0$, qui est un sous-ensemble de $\mathcal{S}$ et que nous appellerons le sous-ensemble des point distingués. $\mathcal{S}_0$ contient D points distingués.

Définition 2.11. *Un élément $d \in \mathcal{S}$ est un point distingué si sa représentation binaire a une propriété particulière qui peut être testée facilement.*

Nous devrons choisir la propriété particulière de nos points distingués de telle façon qu'il soit facile de les générer et de les reconnaitre. Un cas typique est de choisir comme points distingués des entiers de n bits qui ont d zéros à la fin. Alors, si notre ensemble $\mathcal{S}$ contient $N = 2^n$ éléments, le sous-ensemble de points distingués $\mathcal{S}_0$ contiendra $D = 2^{n-d}$ éléments. Afin de trier tous les x de $\mathcal{S}$ et trouver les points distingués, il suffit donc de tester si $0 \leq x \leq 2^{n-d}$.

Une fois la propriété des points distingués définie, nous construisons des chaines. Nous commençons par des points de départ aléatoires $x_0 \in \mathcal{S}$ et nous évaluons plusieurs fois la fonction $f : x_{i+1} = f(x_i)$. Quand un point distingué est détecté, *i.e.* $x_i \in \mathcal{S}_0$,

la construction de cette chaine est terminée. Nous pouvons ensuite commencer à construire une autre chaine à partir d'un point de départ différent.

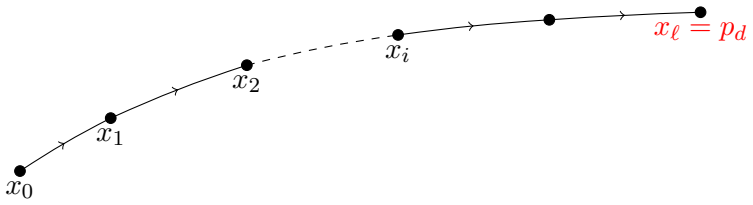

FIGURE 2.5 – Points distingués.

Pour chaque chaine nous stockons le point de départ x_0, le point d'arrivée x_ℓ, qui est un point distingué p_d et la taille de la chaine ℓ, c'est-à-dire le nombre d'itérations de la fonction f jusqu'au point distingué. Une fois les chaines construites et les données de chaque chaine stockées, nous pouvons détecter les collisions.

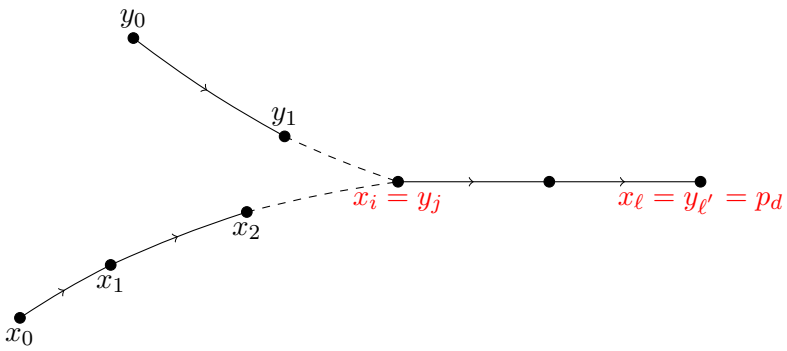

FIGURE 2.6 – Détection d'une collision.

Nous pouvons voir facilement sur la figure 2.6 que deux chaines qui passent par le même point vont obligatoirement terminer au même point d'arrivée, *i.e.* arriver au même point distingué. Afin de détecter une collision, nous utilisons le résultat inverse qui nous dit que deux chaines qui arrivent au même point distingué vont obligatoirement fusionner à un moment donné. Ce point de fusion entre les deux chaines nous donne la vraie collision. Il faut noter ici que si une de deux chaines est une sous-chaine de l'autre, nous arrivons forcement au même point distingué. Cependant, dans ce cas, nous trouvons une fausse collision, *i.e.* une collision que nous ne pouvons pas utiliser.

La question que nous nous posons naturellement ici est comment nous pouvons récupérer la vraie collision à partir d'une collision entre deux points distingués. Nous supposons que nous avons deux chaines qui collisionnent et arrivent au même point distingué p_d. En outre, nous supposons que ℓ est la longueur de la première chaine et ℓ'

la longueur de la deuxième avec $\ell \geq \ell'$. Nous trouvons le point de la plus longue chaine qui correspond au point de départ de la plus courte chaine en faisant exactement $\ell - \ell'$ itérations de la fonction f à partir du point de départ de la plus longue chaine. Dès que le point de correspondance est trouvé, il suffit maintenant de reconstruire en parallèle les deux chaines jusqu'au moment où nous trouvons le vrai point de collision.

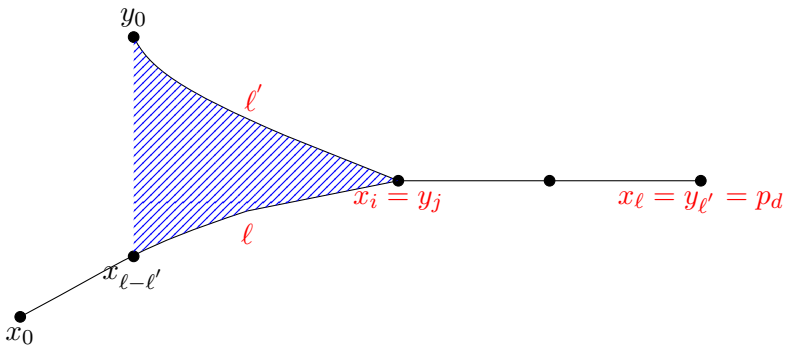

FIGURE 2.7 – Récupération de la collision.

Cette méthode peut être parallélisée facilement. Une machine doit jouer le rôle du serveur central et les autres machines jouent les rôles des clients. Chaque client démarre une marche pseudo-aléatoire en partant de points de départ distincts et en itérant la fonction f. Dès qu'un client trouve un point distingué p_d, il en informe le serveur central en lui donnant les valeurs correspondant à la chaine construite, *i.e.* le point de départ, le point distingué et le nombre d'itérations, puis il rédemarre à partir d'un nouveau point de départ. Dès que le serveur central détecte une collision, *i.e.* lorsqu'un même point distingué a été obtenu deux fois, il procède à la reconstruction de chaines concernées pour récupérer la collision.

Si nous voulons analyser cette méthode, nous remarquons que nous avons besoin de calculer au total $\mathcal{O}(\sqrt{N})$ points. Cela est aussi le cas des autres algorithmes de détection de cycle vus précédemment. De plus, soit θ la proportion de points qui satisfont la propriété des points distingués. Nous avons donc que $\theta = |S_0|/|S| = D/N$. Par conséquence, la longueur moyenne d'une chaine est $\ell = 1/\theta = N/D$. Il en découle que si nous avons P processeurs, nous pouvons calculer PN/D points. Finalement, pour calculer $\mathcal{O}(\sqrt{N})$ points, il faut choisir $D = \mathcal{O}(P\sqrt{N})$ points distingués. En ce qui concerne la mémoire utilisée, le serveur doit stocker P chaines et donc la complexité en mémoire est $\mathcal{O}(P)$. En outre, le temps nécessaire pour parcourir deux chaines est $\mathcal{O}(N/D)$. Ainsi, la borne supérieure de la complexité en temps de l'algorithme qui récupère la vraie collision à partir d'une collision entre deux points distingués est égale à $\mathcal{O}(N/D) = \mathcal{O}(\sqrt{N}/P)$.

Nous devrons aussi noter que, pour éviter les cas dégénérés, il faut interrompre le calcul d'une chaine qui n'arrive pas à un point distingué après $c \cdot N/D$ itérations où c

est une constante fixe avec $c > 1$. Il est donc utile de définir la taille maximale qu'une chaine peut avoir à $20 \cdot N/D$ et abandonner chaque chaine qui dépasse cette limite. La proportion de chaines qui dépassent cette taille est $(1 - N/D)^{20D/N} \approx e^{-20}$. En outre, chaque chaine abandonnée est 20 fois plus longue que la chaine moyenne et donc la fraction de travail abandonné est à peu près égale à $20e^{-20} < 5 \times 10^{-8}$.

Détection de plusieurs collisions

La recherche parallélisable de collisions nous permet aussi de détecter efficacement plusieurs collisions. D'ailleurs, il faut noter que pour récupérer k collisions, il faut avoir un nombre de clients plus élevé que k. En effet, il est évident que chaque chaine peut nous donner une seule collision et donc il n'est pas possible d'avoir k collisions si nous utilisons moins de k machines comme clients.

L'utilisation des algorithmes de détection de cycle pour trouver plusieurs collisions n'est pas optimale car, pour trouver k collisions, il faut répéter k fois l'algorithme. Au contraire, pour trouver k collisions en utilisant les méthodes de recherche parallélisable de collisions, le coût total est égal à $\mathcal{O}(\sqrt{kN})$ au lieu de $\mathcal{O}(k\sqrt{N})$.

2.4 Applications cryptographiques de méthodes de recherche de collisions

Les méthodes de détection de cycle que nous avons vues précédemment peuvent être utilisées pour trouver les collisions dans plusieurs schémas de la cryptographie asymétrique mais aussi symétrique. En ce qui concerne les applications à la cryptographie asymétrique, ces méthodes peuvent par exemple être utilisées pour résoudre les problèmes de la factorisation d'un entier et du logarithme discret. Par exemple, l'algorithme Rho de Pollard est une application de la recherche de collisions. Initialement utilisé pour la factorisation des entiers, il a aussi été adapté pour le calcul du logarithme discret dans un groupe cyclique arbitraire.

En cryptographie symétrique, les algorithmes de détection d'un cycle peuvent, par exemple, être appliqués pour la détection de collisions d'une fonction de hachage. Cependant, leur utilisation nécessite des ordinateurs très puissants. Par exemple, nous considérons la fonction de hachage MD5 qui nous donne en sortie un haché de 128 bits. Pour trouver une collision, par le paradoxe des anniversaires, nous savons que nous avons besoin de faire 2^{64} évaluations de la fonction de hachage. Même si cela est faisable avec les ordinateurs de nos jours, il est beaucoup plus difficile de l'effectuer avec un seul ordinateur. C'est pour cela qu'il est très intéressant de pouvoir faire de recherches parallélisables des collisions.

Dans cette section, nous présentons tout d'abord l'algorithme Rho de Pollard. Ensuite, nous nous intéressons aux applications de méthodes de recherche de collisions en cryptographie symétrique. Dans ce but, nous présentons l'algorithme de recherche des collisions sur DES de Quisquater et Delescaille et l'algorithme de recherche parallé-

lisable des collisions pour résoudre le problème du logarithme discret dans un groupe cyclique de van Oorschot et Wiener.

2.4.1 L'algorithme Rho de Pollard

L'algorithme Rho de Pollard pour la factorisation John M. Pollard a proposé en 1975 l'algorithme Rho de Pollard [Pol75] pour la factorisation. Son but est de factoriser un entier n qui est le produit de deux nombres premiers distincts. Il utilise largement les algorithmes de Floyd et Brent, présentés précédemment, dans le but de détecter les cycles dans une suite définie par une fonction récursive.

Étant donné que p est un facteur premier de n, l'algorithme cherche à trouver des collisions dans la suite produite par la fonction :

$$x_0 = 2, x_{i+1} = f(x_i) = x_i^2 + 1 \mod p, \text{ avec } i \geq 0.$$

L'algorithme de détection de cycle de Floyd est alors utilisé pour trouver deux points x_j et x_{2j} tels que $x_j = x_{2j} \mod p$. Pour trouver une collision, il suffit de tester si $\gcd(x_j - x_{2j}, n) > 1$. Le pseudocode 6 ci-dessous décrit le fonctionnement de cet algorithme.

Algorithme 6 Algorithme Rho de Pollard pour la factorisation des entiers.

ENTRÉES :

- Un entier n
- Nombre maximal d'itérations R

$x \leftarrow 2$
$y \leftarrow 2$
pour tout i de 1 à R **faire**
  $x \leftarrow x^2 + 1$, $y \leftarrow y^2 + 1$, $y \leftarrow y^2 + 1$
  $z = \gcd(x - y), n$
  si $1 < z \leq n$ **alors**
    Retourner z
    Sortie
  fin si
  si $z = n$ **alors**
    Sortie échec
  fin si
fin pour

Le temps nécessaire pour que l'algorithme Rho détecte un cycle (*i.e.* trouver un facteur p de n) est $\mathcal{O}(\sqrt{p})$. Si l'algorithme échoue, nous pouvons réessayer en choisissant une constante différente pour la fonction f, *i.e.* $x_{i+1} = f(x_i) = x_i^2 + c \mod p$ avec $c \neq 0, -2$.

L'algorithme Rho de Pollard pour le logarithme discret Comme c'est le cas de plusieurs algorithmes de factorisation, l'algorithme Rho de Pollard peut être adapté pour calculer des logarithmes discrets. Nous rappelons le problème du logarithme discret : soit $\mathcal{G}$ un groupe multiplicatif d'ordre p (p est un nombre premier), g un générateur de $\mathcal{G}$ et h un élément de $\mathcal{G}$. Résoudre le problème du logarithme discret correspond à trouver un entier α tel que $h = g^{\alpha}$.

Pour utiliser un algorithme de détection de cycle, comme avant, il faut définir une fonction aléatoire f dans $\mathcal{G}$. Supposons que le groupe $\mathcal{G}$ possède une partition en trois sous-ensembles de tailles équivalentes que nous noterons $\mathcal{G}_1, \mathcal{G}_2$ et $\mathcal{G}_3$. Soit f la fonction définie par :

$$x_0 = 1, \;\; x_{i+1} = f(x_i) = \begin{cases} x_i^2, & \text{si } x_i \in \mathcal{G}_1, \\ gx_i, & \text{si } x_i \in \mathcal{G}_2, \\ hx_i, & \text{si } x_i \in \mathcal{G}_3. \end{cases} \tag{2.1}$$

En partant d'un élément $x_0 = g^{a_0} h^{b_0}$, nous calculons la suite x_i ainsi que les suites :

$$a_{i+1} = \begin{cases} 2a_i \mod p, & \text{si } x_i \in \mathcal{G}_1 \\ a_i + 1 \mod p, & \text{si } x_i \in \mathcal{G}_2 \\ a_i, & \text{si } x_i \in \mathcal{G}_3, \end{cases} \tag{2.2}$$

$$b_{i+1} = \begin{cases} 2b_i \mod p, & \text{si } x_i \in \mathcal{G}_1 \\ b_i, & \text{si } x_i \in \mathcal{G}_2 \\ b_i + 1 \mod p, & \text{si } x_i \in \mathcal{G}_3, \end{cases} \tag{2.3}$$

avec $a_0 = 0$, $b_0 = 0$ et $i \geq 0$.

L'algorithme de détection de cycle de Floyd est donc utilisé pour trouver deux éléments x_i et x_{2i} tels que $x_i = x_{2i}$. Si $x_i = x_{2i}$, nous avons $g^{a_i} h^{b_i} = g^{a_{2i}} h^{b_{2i}}$ et donc $h^{b_i - b_{2i}} = g^{a_{2i} - a_i}$. De cela, nous déduisons que :

$$\alpha = \log_g h = \frac{a_{2i} - a_i}{b_i - b_{2i}} \mod p.$$

L'algorithme 7 montre les étapes de l'algorithme Rho de Pollard pour le calcul du logarithme discret. Pour le calcul des suites x_i, a_i et b_i, les fonctions décrites ci-dessus sont utilisées.

Algorithme 7 Algorithme Rho de Pollard pour le logarithme discret.

ENTRÉES :

- Un générateur g d'un groupe cyclique $\mathcal{G}$ d'ordre p
- Un élément h de $\mathcal{G}$

SORTIES :

- Le logarithme discret $\alpha = \log_g h$

$x_0 \leftarrow 1$
$a_0 \leftarrow 0$
$b_0 \leftarrow 0$
pour tout i de 1 à R **faire**
  Calculer x_i, a_i, b_i
  Calculer x_{2i}, a_{2i}, b_{2i}
  si $x_i = x_{2i}$ **alors**
    $r \leftarrow b_i - b_{2i} \mod p$
    si $r = 0$ **alors**
      Sortie échouée
    sinon
      $\alpha = r^{-1}(a_{2i} - a_i) \mod p$
      retourne α
    fin si
  fin si
fin pour

2.4.2 Algorithme de recherche de collisions sur DES de Quisquater et Delescaille

Quisquater et Delescaille présentent à Eurocrypt '89 [QD89a] et Crypto '89 [QD89b] un algorithme qui étudie le problème de recherche des collisions sur les clés du DES. Pour cela, ils utilisent la méthode des points distingués et ils trouvent la première collision sur les clés de DES. Trouver une collision sur les clés de DES revient à trouver une paire de clés k_1 et k_2 de 56 bits, avec k_1 différent de k_2, pour lesquelles nous avons que $\text{DES}_{k_1}(m) = \text{DES}_{k_2}(m)$, où m est un message clair. Ils ont réussi à trouver une première collision en utilisant un ordinateur 35 VAX et un ordinateur SUN qui ont fait des calculs pendant trois semaines.

Pour trouver les collisions, Quisquater et Delescaille présentent trois méthodes. La première est la méthode simple et naïve pour trouver des collisions sur DES. Pour cela, il suffit de choisir un message fixe m et ensuite calculer et stocker dans un tableau toutes les valeurs $\text{DES}_{k_i}(m)$ avec $i = 2^{32}$. Grâce au paradoxe des anniversaires, nous attendons d'avoir une collision dans ce tableau. Pour récupérer la collision, il suffit juste de trier le tableau. Cependant, cette méthode n'est pas efficace car elle a besoin de beaucoup de mémoire.

Algorithme 8 Algorithme de recherche de collisions sur DES de Quisquater et Delescaille.

ENTRÉES :

- La fonction f
- L'ensemble des points distingués $\mathcal{S}_0$

SORTIES :

- La collision détectée si cela existe

$pseudo\_collision \leftarrow 0$
$pseudo\_cycle \leftarrow 0$
$i \leftarrow 0$
répéter
  $m \leftarrow init(i)$
  $k \leftarrow 0$
  répéter
    $m \leftarrow f(m)$
    $k \leftarrow k + 1$
    si $m \in \mathcal{S}_0$ **alors**
      Ajouter m dans T
    fin si
    si $k > limit\_k$ **alors**
      $pseudo\_collision \rightarrow 1$
    fin si
  jusqu'à $pseudo\_collision$
jusqu'à $i > limit\_iteration$

Ensuite, ils décrivent une méthode sans mémoire pour trouver des collisions en utilisant les algorithmes de détection de cycle présentés précédemment. Ici, ils définissent la fonction $f : \{0,1\}^{56} \rightarrow \{0,1\}^{56}$ comme $f(m) = h(\text{DES}_{k_i}(m))$. La fonction h sert à tronquer la sortie du DES pour récupérer finalement 56 bits au lieu de 64. Finalement, après plusieurs itérations de la fonction f, nous trouverons un cycle qui nous donnera une collision possible, *i.e.* $h(\text{DES}_{k_i}(m)) = h(\text{DES}_{k_j}(m))$. Néanmoins, comme h modifie la sortie du DES, la collision trouvée n'est pas forcement une vraie collision mais plutôt une pseudo-collision. Plus précisément, la probabilité que tous les bits des deux sorties soient égaux et donc d'avoir trouvé une vraie collision est $1/256$. Donc, après la détection de chaque pseudo-collision, il faut vérifier s'il s'agit d'une vraie collision ou non. Dans le cas où cela n'en est pas une, il faut recommencer les calculs.

Finalement, ils proposent un algorithme plus efficace qui est basé sur la méthode précédente mais qui peut être parallélisé en utilisant la méthode des points distingués. L'algorithme commence par l'initialisation d'un nouveau message m qui sera utilisé comme point de départ pour le calcul de la chaine. Ensuite, nous construisons la chaine

en itérant la fonction f. Si une des valeurs $f(m)$ générée est un point distingué, elle est stockée dans un tableau T et l'algorithme vérifie si la même valeur est déjà stockée dans T. Si oui, nous considérons que nous avons trouvé une pseudo-collision. Si aucun point distingué n'est détecté, la construction de la chaine est arrêtée après un nombre prédéfini d'itérations $limit\_r$ de la fonction f. Après avoir calculé plusieurs chaines, l'algorithme récupère toutes les collisions détectées sur les points distingués et vérifie s'il s'agit d'une vraie collision ou non. Le pseudo-code de l'algorithme 8 décrit ce fonctionnement.

Pour cette attaque, Quisquater et Delescaille ont utilisé plusieurs machines qui ont travaillé en parallèle mais en utilisant des clés et des points de départs différents. Au total, ils ont réussi à trouver 21 collisions sur DES.

2.4.3 Algorithme de recherche parallélisable des collisions de van Oorschot et Wiener

La méthode de points distingués a été utilisée par van Oorschot et Wiener [vOW99] pour présenter un algorithme de recherche parallélisable des collisions pour résoudre le problème du logarithme discret dans un groupe cyclique. Soit $\mathcal{G}$ un groupe cyclique d'ordre n (n est un nombre premier) et soit g un générateur du groupe. En outre, supposons que le groupe $\mathcal{G}$ possède une partition en trois sous-ensembles de tailles équivalentes que nous noterons $\mathcal{G}_1, \mathcal{G}_2$ et $\mathcal{G}_3$. Si y est un élément du groupe pour lequel nous avons que $y = g^x$, le but de l'algorithme est de trouver x.

Cette méthode fonctionne de façon similaire aux algorithmes vus précédemment. Il faut tout d'abord définir la fonction itérée f qui sera utilisée pour la construction de chaines, *i.e.* $x_{i+1} = f(x_i)$. Van Oorschot et Wiener proposent d'utiliser la même fonction que Pollard dans son algorithme de base :

$$f(x_i) = \begin{cases} x_i^2, & \text{si } x_i \in \mathcal{G}_1, \\ gx_i, & \text{si } x_i \in \mathcal{G}_2, \\ hx_i, & \text{si } x_i \in \mathcal{G}_3. \end{cases} \tag{2.4}$$

Comme pour la méthode de points distingués, une machine joue le rôle du serveur central et les autres machines jouent les rôles des clients. Chaque client choisit, de manière indépendante des autres, son point de départ x_0 pour construire une chaine en utilisant la fonction f. Pour cela, il génère deux exposants aléatoires a_0, b_0 avec $0 \leq a_0 < n$ et $0 \leq b_0 < n$ et définit le point de départ $x_0 = g^{a_0} y^{b_0}$. Dès qu'une chaine rencontre un point distingué d_p, le client en informe le serveur central en lui donnant les valeurs correspondantes de a_i et b_i telles que $d_p = x_i = g^{a_i} y^{b_i}$ et le point distingué d_p. Ensuite, le client redémarre l'opération à partir d'un nouveau point de départ. Pour détecter une collision, il faut que le serveur central récupère deux chaines qui arrivent au même point distingué. Soit deux chaines pour lesquelles nous avons une collision, *i.e.*

$x_i = d_p = y_j$ et soit les exposants correspondants (a_i, b_i) et (c_j, d_j). Nous en déduisons :

$$\left.\begin{array}{l} x_i = g^{a_i} y^{b_i} \\ z_j = g^{c_j} y^{b_j} \end{array}\right\} \Rightarrow g^{a_i} y^{b_i} = g^{c_j} y^{b_j} \Rightarrow g^{a_i - c_j} = y^{(d_j - b_i)}$$

Algorithme 9 Algorithme de van Oorschot et Wiener pour le logarithme discret : client.

ENTRÉES :

- Un générateur g d'un groupe cyclique $\mathcal{G}$ d'ordre n
- Un élément y de $\mathcal{G}$
- Les suites (2.2) et (2.3)

tant que Connexion avec le serveur active **faire**
  Choisir aléatoirement a, b tels que $0 \leq a < n$ et $0 \leq b < n$.
  $x \leftarrow g^a y^b$
  tant que $x \notin S_0$ **faire**
    $x \leftarrow f(x)$
    Calculer a (2.2)
    Calculer b (2.3)
  fin tant que
  Envoyer au serveur (x, a, b)
fin tant que

Algorithme 10 Algorithme de van Oorschot et Wiener pour le logarithme discret : serveur.

ENTRÉES :

- Un générateur g d'un groupe cyclique $\mathcal{G}$ d'ordre n
- Un élément y de $\mathcal{G}$

SORTIES :

- Un élément α tel que $\alpha = \log_g y$

Activer la connexion avec les clients
tant que DLP pas résolu **faire**
  Recevoir les triplets (x, a, b) des clients
  Stocker les triplets (x, a, b) dans un tableau T
  Trier le tableau T
  si $x = x^{'}$ pour deux triplets (x, a, b) et $(x^{'}, a^{'}, b^{'})$ **alors**
    si $b^{'} \not\equiv b \mod n$ **alors**
      retourne $(a - a^{'})(b^{'} - b)^{(-1)} \mod n$
    fin si
  fin si
fin tant que
Terminer la connexion avec les clients

Si $b_i \not\equiv d_j \mod n$, le serveur central peut donc calculer le logarithme discret qui est égal à :

$$\log_g y = (a_i - c_j) \cdot (d_j - b_i)^{-1} \mod n.$$

Les algorithmes 10 et 9 présentent en pseudo-code le fonctionnement de cet algorithme. La fonction f est la fonction (2.4) et $\mathcal{S}_0$ désigne l'ensemble de points distingués.

2.4.4 Le compromis temps-mémoire de Hellman

Hellman a introduit en 1980 [Hel80] le principe des attaques par compromis temps-mémoire contre les algorithmes de chiffrement par blocs. Ce type d'attaque est présenté ici parce que son fonctionnement est très similaire aux algorithmes utilisés pour l'analyse de fonctions aléatoires vus précédemment.

L'algorithme d'Hellman a pour objet d'inverser une fonction à sens unique, *i.e.* inverser une fonction $f : E \to F$ pour laquelle il est facile de calculer $f(x) = y$, mais très difficile de trouver x à partir de y. Dans le cas d'un algorithme de chiffrement par blocs, la fonction f est de la forme $f(K) = E_K(0^n)$ où E représente l'algorithme de chiffrement par blocs que nous souhaitons attaquer et K la clé recherchée Évidemment, la recherche exhaustive et l'attaque par dictionnaire peuvent être utilisées pour résoudre ce type de problèmes. Néanmoins, la première est très coûteuse en temps de calcul et la deuxième utilise une très grande taille de mémoire. Le but des attaques par compromis

temps-mémoire est donc de trouver un compromis entre ces deux méthodes. Pour faire cela plus efficacement, Hellman utilise une étape de précalcul et chercher à trouver un bon compromis entre la taille de la mémoire utilisée et le temps nécessaire de calcul.

Le compromis temps-mémoire de Hellman figure dans l'article d'origine pour attaquer DES. Cependant, il a été aussi utilisé pour attaquer plusieurs cryptosystèmes. Babbage [Bab95] et Golic [Gol97] ont adapté ce type d'attaques pour proposer un compromis temps-mémoire-données contre les algorithmes de chiffrement à flot.

L'étape du précalcul. Le précalcul consiste à calculer et stocker en mémoire un certain nombre de valeurs qui seront utilisées pendant la phase en-ligne de l'attaque. Dans le compris de Hellman, le précalcul se fonde sur la construction de chaînes de valeurs en itérant la fonction $f : \{0, 1, \ldots, 2^k - 1\} \to \{0, 1, \ldots, 2^k - 1\}$, où k est la taille de la clé utilisée par la fonction f que l'attaquant veut inverser. Pendant cette phase du précalcul l'attaquant choisit m points de départ aléatoires $(SP_1, SP_2, \ldots, SP_m)$ et, à partir de chaque point de départ, il construit une chaine en itérant t fois la fonction f comme le montre la figure 2.8.

Après avoir calculé les chaines, l'attaquant stocke dans un tableau T les points de départs SP et les points d'arrivées EP de chaque chaine.

L'étape de calcul en-ligne. L'étape de calcul en-ligne consiste à inverser la fonction de chiffrement de l'algorithme de chiffrement par bloc de façon plus rapide que la recherche exhaustive à l'aide des tableaux calculés pendant l'étape du précalcul. Ici, l'attaquant possède une valeur y pour laquelle il sait que $f(x) = y$ et il cherche à trouver x. Pour cela, il recalcule à nouveau les chaînes et il cherche à trouver la chaîne dont le point d'arrivée est égale à $f(y)$. Ensuite, il recalcule cette chaîne à partir de son point de départ et il arrête le calcul dès qu'il trouve la valeur y. La valeur d'avant lui donne x.

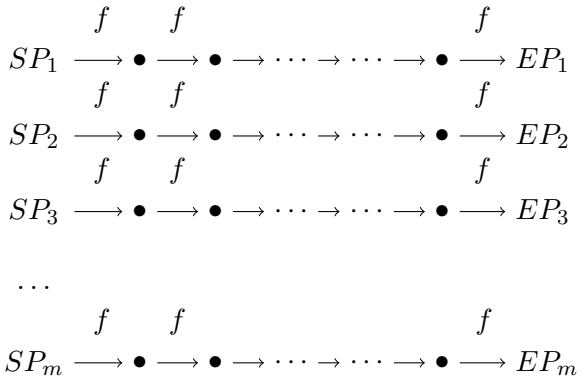

FIGURE 2.8 – Construction des tableaux de Hellman.

Analyse de l'attaque. La probabilité de succès de l'attaque est de l'ordre de $mt/2k$, où k est la taille de la clé, tant que $mt^2 < 2^k$. Si nous allons au-delà de cette limite, le nombre des collisions entre les chaînes devient très grand et le compromis ne présente plus d'intérêt. Pour $m = t = 2^{k/3}$, la probabilité de succès est $2^{-k/3}$ avec une complexité de $\mathcal{O}(2^{k/3})$ opérations en mémoire et $\mathcal{O}(k2^{k/3})$ opérations en temps. Pour améliorer cette probabilité, Hellman propose d'utiliser $2^{k/3}$ versions différentes de la fonction f. En supposant que les tableaux générés pour chaque version de la fonction sont indépendents, la probabilité de succès augmente à une fraction constante de l'espace des clés et la complexité en temps et en mémoire devient $\mathcal{O}(2^{k/3})$ opérations.

3. Attaques génériques sur le schéma Even-Mansour

Dans ce chapitre, nous présentons les attaques sur le schéma Even-Mansour que nous avons développées pendant cette thèse. Il s'agit d'attaques par collision visant à récupérer la clé de l'algorithme utilisé. Elles sont issues d'un article publié à Asiacrypt

2014 en collaboration avec Pierre-Alain Fouque et Antoine Joux [FJM14]. Ce type d'attaque utilise les techniques algorithmiques qui ont été introduites au chapitre 2. Cela nécessite de remanier à profondeur la méthode pour lui permettre de travailler avec deux fonctions distinctes plutôt qu'une seule grâce à la méthode des chaînes !parallèles que nous avons introduite dans cet article.

Nous considérons aussi ce nouveau type d'attaque contre le schéma Even-Mansour dans un modèle multi-utilisateurs. Dans ce modèle, l'attaquant ne s'intéresse pas uniquement à l'attaque d'un unique utilisateur mais à tout un groupe. Il souhaite donc récupérer un maximum de clés d'utilisateurs de ce groupe pour un coût total aussi faible que possible. Il espère en particulier réaliser une économie d'échelle par rapport à l'approche classique qui consiste à attaquer les différents utilisateurs séparément. Ces attaques sont d'autant plus efficaces que le nombre d'utilisateurs sera grand. Bien sûr, il faudra s'assurer que ce nombre reste réaliste.

Nous commençons ce chapitre en présentant le schèma Even-Mansour et les attaques existante sur ce schèma. Ensuite, nous présentons nos nouvelles techniques et nos attaques génériques sur ce cryptosystème.

3.1 Le schéma Even-Mansour

La conception et l'analyse des constructions minimales a toujours été un sujet important de la recherche cryptographique. Déjà en 1991, S. Even et Y. Mansour [EM91, EM97] ont abordé le problème d'une construction simple d'algorithme de chiffrement par blocs ayant une preuve formelle de sécurité. Leur schéma a été inspiré par DESX proposé par R. Rivest en 1984.

3.1.1 À l'origine DESX : une simple extension du DES

Pour trouver une solution au problème de la recherche exhaustive sur la clé du DES, plusieurs approches ont été suggérées. Une solution est de construire un algorithme de chiffrement par blocs basé sur DES mais qui utilise une clé plus longue, comme par exemple le triple DES (3DES). Cependant, cette solution s'avère assez coûteuse, car pour un chiffrement ou déchiffrement du 3DES, plusieurs chiffrements ou déchiffrements du DES sont nécessaires.

En 1984, Rivest (non publié) propose une construction d'un nouvel algorithme de chiffrement par blocs qui utilise des clés supplémentaires sans impacter la structure interne du DES ni faire plusieurs appels à l'algorithme. Pour cela, il utilise un bloc de 64 bits, une clé DES K de 56 bits et deux clés de blanchiment de 64 bits chacune qui sont XORées en entrée et en sortie du DES. DESX est donc défini par :

$$\text{DESX}_{K||K_1||K_2}(m) = K_2 \oplus DES_K(K_1 \oplus m).$$

Il est évident que pour $K_1 = K_2 = 0$, DESX devient équivalent au DES.

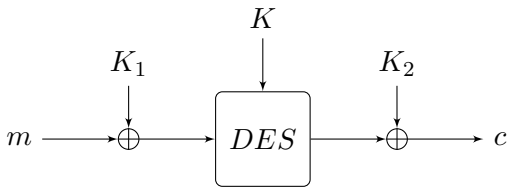

FIGURE 3.1 – Le schéma DESX.

3.1.2 La construction de Even et Mansour

Inspirés par DESX, Even et Mansour [EM91, EM97] proposent à Asiacrypt en 1991 une construction minimaliste d'un algorithme de chiffrement par blocs en utilisant une permutation aléatoire. Étant donné une permutation publique π qui opère sur des valeurs de n bits et deux clés de blanchiment K_1 et K_2 de n bits XORées en entrée et en sortie de π, ils construisent une permutation Π avec clé :

$$\Pi_{K_1,K_2}(m) = \pi(m \oplus K_1) \oplus K_2.$$

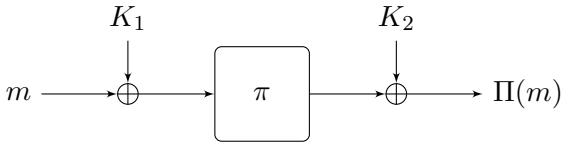

FIGURE 3.2 – Le schéma Even-Mansour.

Pour souligner que leur construction ne peut pas se simplifier plus, Even et Mansour notent que la suppression d'une de clés détériorera la sécurité du système. Plus précisément, si la clé K_2 n'est pas utilisée, nous aurons $\Pi_{K_1}(m) = \pi(m \oplus K_1)$ et donc l'adversaire qui a accès à la permutation publique π peut calculer $\pi^{-1}(\Pi(m)) = m \oplus K_1$. Si la clé K_1 n'est pas utilisée, le schéma revient à $\Pi_{K_2}(m) = \pi(m) \oplus K_2$. L'adversaire pourrait, avec une seule paire connue de messages clair et chiffré, calculer $\pi(m)$ et ensuite récupérer la clé $K_2 = \Pi_{K_2}(m) \oplus \pi(m)$.

La construction FX

Suite à la publication des schémas Even-Mansour et DESX, Kilian et Rogaway [KR96, KR01] ont généralisé DESX en définissant la construction FX. Ils montrent qu'à partir d'un algorithme de chiffrement par blocs F de n bits avec une clé K de k bits et deux clés de blanchiment K_1, K_2 de n bits, nous pouvons définir un algorithme de chiffrement par blocs FX :

$$\mathrm{FX}_{K||K_1||K_2}(m) = K_2 \oplus F_K(K_1 \oplus m).$$

Ils montrent également que la recherche exhaustive sur FX est plus coûteuse que celle sur F.

Étant donné que F est un algorithme de chiffrement par blocs idéal et, en admettant, que l'adversaire puisse récupérer 2^d paires clair/chiffré, ils ont prouvé que cette construction atteint $(k + n - 1 - d)$ bits de sécurité.

En outre, ils montrent que les schémas $\mathrm{FX}_{K||K_1}(m) = F_K(K_1 \oplus m)$ et $\mathrm{FX}_{K||K_1}(m) = K_1 \oplus F_K(m)$ n'améliorent pas la sécurité de F. Néanmoins, le schéma :

$$\mathrm{FX}_{K||K_1}(m) = K_1 \oplus F_K(K_1 \oplus m)$$

offre la même sécurité que la version avec deux clés distinctes.

Variantes du schéma Even-Mansour

En 2012, deux variantes du schéma Even-Mansour ont été proposées indépendamment à Eurocrypt. Dunkelman, Keller et Shamir [DKS12] ont proposé une variante de Even-Mansour avec une seule clé (*Single Key Even-Mansour - SEM*) comme le montre la figure 3.3.

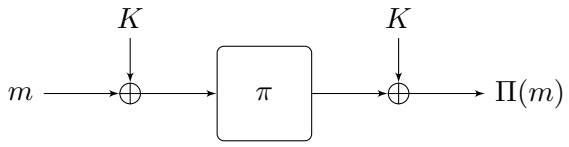

FIGURE 3.3 – Le schéma Even-Mansour avec une seule clé.

Bogdanov *et al.* [BKL+12] ont généralisé le schéma Even-Mansour à plusieurs tours et ils ont proposé le schéma Even-Mansour itéré. Étant donné r permutations aléatoires publiques $\{\pi_1, \pi_2, \ldots, \pi_r\}$ et $(r + 1)$ clés $\{K_0, K_1, \ldots, K_r\}$ (voir figure 3.4), le schéma Even-Mansour itéré est défini par :

$$\Pi(m) = \pi_r(\ldots(\pi_2(\pi_1(m \oplus K_0) \oplus K_1)\ldots) \oplus K_r.$$

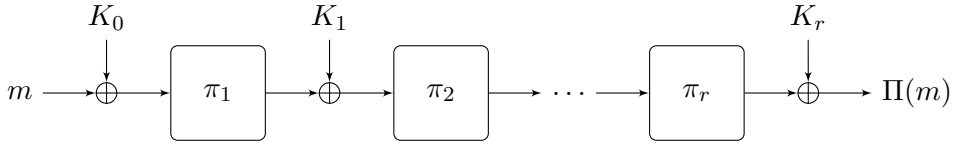

FIGURE 3.4 – Le schéma Even-Mansour itéré.

Récemment, plusieurs schémas Even-Mansour adaptables (*Tweakable Even-Mansour - TEM*) ont été proposés. Sasaki *et al.* ont introduit ce type de constructions en propo-

sant l'algorithme Minalpher [STA+] à la compétition CAESAR[1]. Minalpher est basé sur la construction XEX [Rog04] de Rogaway. Dans ce cas, la construction Even-Mansour adaptable est définie comme :

$$TEM_K^P((\alpha_1, \ldots, \alpha_\ell, N), m) = P(m \oplus \Delta) \oplus \Delta,$$

où $\Delta = x_1^{\alpha_1}, \ldots, x_\ell^{\alpha_\ell}(K \parallel N \oplus P(K \parallel N))$, $x_1, \ldots, x_\ell$ sont des générateurs dans $GF(2^n)$, P est la permutation publique utilisée, N le nonce et m le message. Mennink [Men15] a généralisé le schéma de Minalpher en proposant XPX.

En outre, Cogliati, Lampe et Seurin [CLS15] proposent à Crypto 2015 le schéma Even-Mansour adaptable (TEM). Il est basé sur la construction CLRW de Landecker *et al.* [LST12]. TEM est construit à partir d'une permutation P de n bits et d'une famille de fonctions de hachage universelles H_k pour un espace des *tweaks* T dans $\{0,1\}^n$:

$$TEM_K^P(t, m) = H_K(t) \oplus P(H_K(t) \oplus m)$$

où K est la clé, $t \in T$ est le *tweak* et m est le message. Cette construction peut être généralisée pour plusieurs tours comme nous pouvons voir en figure 3.5 :

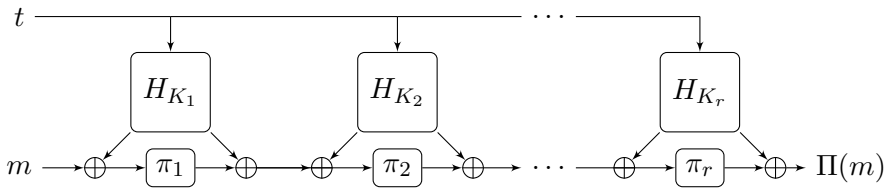

FIGURE 3.5 – Le schéma Even-Mansour adaptable basé sur CLRW avec r tours.

3.1.3 Les preuves de sécurité du schéma Even-Mansour

Dans leur article, Even et Mansour se placent dans le modèle de la permutation aléatoire de $N = 2^n$ éléments et prouvent que pour π, K_1 et K_2 uniformément aléatoires, si $\mathcal{T}$ est le nombre de requêtes de l'attaquant à la permutation publique π ou à son inverse π^{-1} et $\mathcal{D}$ le nombre de requêtes à la permutation avec clé Π ou son inverse Π^{-1} (attaque CCA), l'attaquant peut déchiffrer un chiffré avec probabilité $\mathcal{O}(\frac{\mathcal{DT}}{N})$. Cela veut dire que leur schéma est sûr jusqu'à environ $2^{n/2}$ requêtes de l'attaquant à la permutation π et à la permutation Π.

Dunkelman *et al.* ont prouvé dans [DKS12] que la variante simplifiée de Even-Mansour avec une seule clé a exactement la même sécurité que la version avec clé et donc un attaquant a encore besoin de faire $2^{n/2}$ requêtes. Cette borne de sécurité pour le schéma avec une seule clé est potentiellement surprenante mais pas inattendue car la majorité des attaques existantes cherchent à récupérer les deux clés indépendamment.

[1] http ://competitions.cr.yp.to/caesar.html

En effet, une fois que la clé K_1 est retrouvée, il est très simple de récupérer ensuite la clé K_2 comme nous l'avons expliqué précédemment.

En ce qui concerne la version itérée, Bogdanov *et al.* ont montré dans leur article principal [BKL+12] que, pour $r = 2$ tours, la construction est sûre jusqu'à $2^{2n/3}$ requêtes de l'attaquant à la permutation π et à la permutation Π. Plusieurs papiers [Ste12, LPS12], qui améliorent la borne de sécurité du schéma en augmentant le nombre de tours, ont été présentés par la suite. Finalement, Chen et Steinberger [CS14] ont prouvé que le schéma Even-Mansour itéré avec r tours et r clés indépendamment choisies est sûr jusqu'à $2^{\frac{rn}{r+1}}$ requêtes de la part de l'adversaire.

En plus des preuves de sécurité concernant les attaques CCA, la sécurité du schéma Even-Mansour a été prouvée pour des attaques à clés reliées [CS15, FP15], à clé choisie [CS15] et à clé connue [ABM13].

3.1.4 Attaques connues sur Even-Mansour

Nous présenterons ici les attaques connues sur le schéma Even-Mansour et qui nous seront utiles pour la compréhension de nos attaques expliquées au chapitre 4.

L'attaque à clair connu de Daemen

L'attaque à clair connu de Daemen [Dae91] utilise deux paires clair/chiffré connues (m_1, c_1) et (m_2, c_2). Soit $v_i = m_i \oplus K_1$ et $w_i = c_i \oplus K_2$ avec $i \in \{1, 2\}$ et π la permutation publique utilisée. En faisant une recherche exhaustive sur la clé K_1, l'attaquant trouve toutes les valeurs possibles des v_i. Ensuite, pour chaque valeur de v_i trouvée, il calcule :

$$c_1 \oplus c_2 = w_1 \oplus w_2 = \pi(v_i) \oplus \pi(v_i \oplus (m_1 \oplus m_2)).$$

Si $v_i = m_1 \oplus K_1$ ou $v_i = m_2 \oplus K_1$, l'équation ci-dessus est vraie et donc l'attaquant en déduit un candidat pour la clé K_1. Cette attaque a une complexité en temps de 2^{n+1} opérations et nécessite d'avoir deux paires clair/chiffré.

L'attaque à clair choisi de Daemen

L'attaque à clair choisi de Daemen [Dae91] est un compromis temps/mémoire de l'attaque précédente. L'adversaire choisit $\mathcal{D}$ paires de clairs $(m_0^{(i)}, m_1^{(i)})$, avec $0 \leq i \leq \mathcal{D} - 1$, telles que $m_0^{(i)} \oplus m_1^{(i)} = \delta$ où δ est une constante différente de zéro. Nous remarquons que cette différence δ entre les messages clairs sera conservée après le XOR de la première clé de blanchiment K_1. Ensuite, l'attaquant demande le chiffrement de toutes les paires et il récupère et stocke les chiffrés correspondants $(c_0^{(i)}, c_1^{(i)})$. Comme pour les clairs, la différence entre les messages chiffrés sera conservée par le XOR de la deuxième clé de blanchiment K_2. Soit $v^{(i)} = m_0^{(i)} \oplus K_1$. Pour chaque valeur $v^{(i)}$ possible, il calcule $\pi(v^{(i)}) \oplus \pi(v^{(i)} \oplus \delta)$. S'il trouve une collision $\pi(v^{(i)}) \oplus \pi(v^{(i)} \oplus \delta) = c_0^{(i)} \oplus c_1^{(i)}$, il déduit que $v^{(i)} = m_0^{(i)} \oplus K_1$ ou $v^{(i)} = m_0^{(i)} \oplus K_1 \oplus \delta$ et donc il récupère K_1.

D'après le paradoxe des anniversaires, nous attendons d'avoir une collision avec grande probabilité entre les valeurs $\pi(v^{(i)}) \oplus \pi(v^{(i)} \oplus \delta)$ et $c_0^{(i)} \oplus c_1^{(i)}$ quand $DT = \mathcal{O}(N)$. Cette attaque vérifie donc la borne de sécurité de l'article d'origine de Even et Mansour.

Attaque *slide* de Biryukov-Wagner

Presque dix ans après les attaques de Daemen, Biryukov et Wagner [BW99] définissent les attaques *slide*. Ils proposent ensuite [BW00] une attaque *slide* avec un twist (*slide with a twist*) sur le schéma Even-Mansour qui est en pratique une attaque à clair connu.

Pour expliquer leur attaque, il faut tout d'abord définir une paire *slid* : une paire de messages clairs (m, m') tel que $m \oplus m' = K_1$.

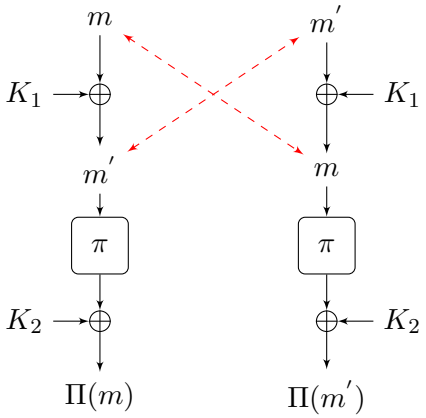

FIGURE 3.6 – Une paire *slid*.

Pour ces messages clairs, nous observons que :

$$\Pi(m) \oplus \pi(m') = K_2 \text{ et } \Pi(m') \oplus \pi(m) = K_2, \text{ donc}$$

$$\Pi(m) \oplus \pi(m) = \Pi(m') \oplus \pi(m').$$

Il suffit donc de trouver deux messages clairs qui satisfont cette égalité. Par le paradoxe des anniversaires, nous avons que dans un ensemble de $2^{n/2}$ messages clairs, nous pouvons trouver au moins une paire *slid* (m, m') qui nous donnera les deux clés candidates $K_1 = m \oplus m'$ et $K_2 = \Pi(m) \oplus \Pi(m')$. L'attaque nécessite d'avoir $D = 2^{(n+1)/2}$ messages clairs connus et faire $T = 2^{(n+1)/2}$ requêtes à π. Nous avons donc $DT = 2^{n+1}$ qui correspond à la borne de sécurité du schéma Even-Mansour à un facteur 2 constant prés. Avec cette attaque, nous obtenons la même complexité que l'attaque de Daemen, mais avec des messages clairs connus au lieu des messages clairs choisis.

Application de l'attaque *slide* sur DESX Comme nous l'avons vu précédemment, la construction FX de DESX est similaire au schéma Even-Mansour. De ce fait, certaines attaques connues sur Even-Mansour peuvent être adaptées pour attaquer DESX. Cependant, les détails de ces attaques sont différentes. Dans le cas de DESX, la permutation publique n'existe pas. À la place, la fonction DES paramétrée par la clé de l'utilisateur est utilisée.

Par exemple, l'attaque *slide* de Biryukov et Wagner peut être appliquée sur DESX. Nous rappelons la fonction de DESX :

$$\text{DESX}_{K||K_1||K_2}(m) = K_2 \oplus DES_K(K_1 \oplus m).$$

Dans ce cas, une paire *slid* est définie comme suit : deux paires clair/chiffré (m, c) et $(m^{'}, c^{'})$ forment une paire *slid* si et seulement si $c \oplus c^{'} = K_2$. Nous observons que pour une paire *slid* :

$$m^{'} = K_1 \oplus DES_K^{-1}(c^{'} \oplus K_2) = K_1 \oplus DES_K^{-1}(c) \text{ et}$$

$$m = K_1 \oplus DES_K^{-1}(c \oplus K_2) = K_1 \oplus DES_K^{-1}(c^{'}).$$

Si nous combinons ces deux résultats il en découle que :

$$K_1 = m^{'} \oplus DES_K^{-1}(c) = m \oplus DES_K^{-1}(c^{'}).$$

De ce fait, pour détecter une paire *slid*, il nous suffit de détecter une collision :

$$m \oplus DES_K^{-1}(c) = m^{'} \oplus DES_K^{-1}(c^{'})$$

qui nous donnera un candidat potentiel pour la clé K_1.

Pour obtenir donc une paire *slid*, il faut avoir un ensemble de $2^{32.5}$ paires clair/chiffré et chercher une collision entre elles. L'attaque nécessite donc d'avoir $\mathcal{D} = 2^{32.5}$ paires de messages clair/chiffré et faire $\mathcal{T} = 2^{87.5}$ appels aux fonctions de chiffrement et déchiffrement de DES.

Attaque *slidex* de Dunkelman *et al.*

Dunkelman *et al.* [DKS12] ont proposé une attaque *slide* avancée en utilisant un degré de liberté supplémentaire δ. Leur attaque est appelée attaque *slidex* et fonctionne de la manière suivante pour le schéma avec deux clés.

Nous supposons que nous avons une paire *slidex* de messages clairs $(m, m^{'})$ telle que $m \oplus m^{'} = K_1 \oplus \delta$ où $\delta \in \{0,1\}^n$.

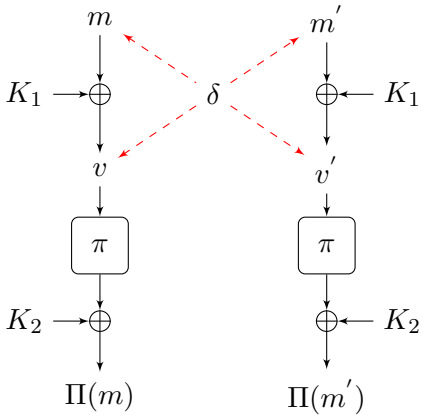

FIGURE 3.7 – Une paire *slidex*.

Nous définissons :

$$F(m) = \Pi(m) \oplus \Pi(m \oplus \delta) \text{ et } f(m) = \pi(m) \oplus \pi(m \oplus \delta).$$

Nous observons que :

$$\begin{aligned} F(m') &= \Pi(m') \oplus \Pi(m' \oplus \delta) \\ &= \cancel{K_2} \oplus \pi(m' \oplus K_1) \oplus \cancel{K_2} \oplus \pi(m' \oplus \delta \oplus K_1) \\ &= \pi(m \oplus \delta) \oplus \pi(m) = f(m). \end{aligned}$$

Il suffit donc juste de trouver des collisions :

$$\Pi(m') \oplus \Pi(m' \oplus \delta) = \pi(m) \oplus \pi(m \oplus \delta).$$

Pour chaque collision trouvée, $m \oplus m' \oplus \delta$ est un bon candidat pour la clé K_1. Une fois la K_1 identifiée, il est facile de récupérer aussi $K_2 = \Pi(m) \oplus \pi(m' \oplus \delta)$. Le nombre de requêtes à la permutation avec clé Π est égal à $D = 2^{(d+1)/2}$ et le nombre des requêtes à la permutation publique π est égal à $T = 2^{n-(d-1)/2}$. Par conséquent, nous avons donc que $DT = 2^{n+1}$ et que la borne de sécurité du schéma est atteinte à un facteur 2 constant près.

Dans leur article, Dunkelman *et al.* décrivent aussi une attaque similaire appliquée sur le schéma Even-Mansour avec une seule clé. Pour cela, nous appliquons la construction Davies-Meyer aux permutations π et Π et nous définissons les fonctions :

$$F(m) = \Pi(m) \oplus m \text{ et } f(m) = \pi(m) \oplus m.$$

Pour $m^{'} = m \oplus K$, nous observons que :

$$\begin{aligned} F(m^{'}) &= \Pi(m^{'}) \oplus m^{'} = \Pi(m \oplus K) \oplus m \oplus K \\ &= \pi(m \oplus \not{K} \oplus \not{K}) \oplus \not{K} \oplus m \oplus \not{K} \\ &= \pi(m) \oplus m = f(m) \end{aligned}$$

Il suffit donc de trouver une paire $(m, m^{'})$ pour laquelle nous avons une collision $\Pi(m^{'}) \oplus m^{'} = \pi(m) \oplus m$. La complexité de cette attaque atteint aussi la borne de sécurité du schéma Even-Mansour.

Il faut noter ici que la principale différence entre l'attaque *slide* de Biryukov-Wagner et l'attaque *slidex* de Dunkelman *et al.* se situe dans les appels à la permutation publique π et à la permutation avec clé Π. Dans la première attaque, ils sont mélangés. Au contraire, dans la deuxième, ils sont séparés. Cela veut dire que nous pouvons effectuer les appels à π en temps de précalcul.

Attaque sans mémoire. Toutes les attaques sur Even-Mansour étudient seulement les complexités en temps et en données. Le coût de l'attaque en mémoire n'est pas considéré. Dans leur article, Dunkelman *et al.* proposent également une variante de l'attaque *slidex* sur Even-Mansour qui a une complexité en temps et en données égale à $\mathcal{O}(2^{n/2})$ opérations et une complexité en mémoire constante. Comme leur attaque *slidex*, ici aussi ils cherchent à détecter des collisions de la forme :

$$\Pi(m^{'}) \oplus \Pi(m^{'} \oplus \delta) = \pi(m) \oplus \pi(m \oplus \delta),$$

et donc sans mélanger les appels à la permutation avec clé et les appels à la permutation publique. Pour cela, ils utilisent les algorithmes classiques de recherche de collisions qui ont été présentés au chapitre 2, comme par exemple l'algorithme de détection de cycle de Floyd. Cependant, leur attaque est une attaque à clair adaptatif choisi et non plus une attaque à clair connu comme il est le cas de l'attaque *slidex*.

Pour conclure leur article, Dunkelman *et al.* se demandent s'il existe une attaque sans mémoire sur le schéma Even-Mansour qui fait D requêtes à la permutation Π et N/D à la permutation publique π avec $D \ll N^{1/2}$. Le but des attaques que nous présentons dans ce chapitre est de donner une réponse effective à cette question.

3.2 La technique de détection de chaînes parallèles

Nous présentons ici la technique de détection de chaînes parallèles [FJM14]. Cette technique modifie l'idée de base de van Oorschot et Wiener. Comme présenté en section 2.4.3, dans leur algorithme, van Oorschot et Wiener construisent des chaines en itérant une fonction aléatoire et utilisent la méthode des points distingués pour détecter celles qui fusionnent. Nous montrons ici que nous pouvons appliquer des techniques similaires sur Even-Mansour. En effet, en utilisant quelques fonctions spécifiques au schéma de

chiffrement d'Even-Mansour, nous sommes capables de récupérer le XOR entre les clés des utilisateurs. Cependant, dans notre cas, nous avons un problème : les deux fonctions que nous itérons pour construire nos chaines ne sont pas les mêmes. Par conséquent, les chaînes ne peuvent pas fusionner et donc nous ne pouvons pas utiliser la méthode de points distingués. Pour résoudre ce problème, nous modifions les deux fonctions et nous définissons deux nouvelles fonctions similaires qui ne peuvent plus fusionner mais qui peuvent devenir parallèles, *i.e.* maintenant nous cherchons des chaînes qui ont une différence constante entre elles. Grâce à nos attaques, nous montrons que notre méthode de chaînes parallèles est aussi efficace que la méthode de base.

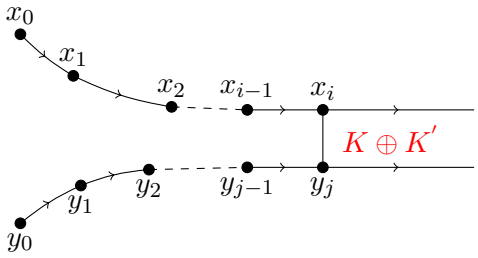

FIGURE 3.8 – Schéma de chaînes parallèles.

3.2.1 Le principe des chaînes parallèles.

Comme l'algorithme de van Oorschot et Wiener, le but de notre technique est de détecter une collision entre deux chaînes qui appartiennent à deux ensembles distincts. Le premier ensemble contient des chaines construites à partir de la permutation publique π et le deuxième des chaînes construites à partir de la permutation avec clé Π.

Dans l'algorithme de van Oorschot et Wiener, afin de détecter une collision, nous construisons des chaînes et nous souhaitons que ces chaînes fusionnent. Le point de fusion nous donne la collision recherchée. Nous souhaitons suivre la même procédure. Nous aurons donc besoin de définir deux fonctions, basées sur les permutations publiques et secrètes d'Even-Mansour, que nous utiliserons pour construire nos chaînes et chercher les points distingués. Une première idée serait d'utiliser les fonctions :

$$F(m) = \Pi(m) \oplus \Pi(m \oplus \delta) \text{ et } f(m) = \pi(m) \oplus \pi(m \oplus \delta)$$

comme nous l'avons vu dans l'attaque de Dunkelman *et al.* en section 3.1.4. Cependant, nous remarquons que deux chaînes construites à partir de fonctions F et f ci-dessus peuvent éventuellement se croiser mais pas fusionner puisque les fonctions sont distinctes.

Une autre option serait de définir une nouvelle fonction qui mélange les appels aux permutations publiques et privées. Mais, notre but est de présenter une attaque sur la permutation avec clé d'Even-Mansour qui est basée sur les algorithmes classiques de recherche des collisions et qui fait des requêtes sur π et Π sans avoir de dépendance

entre elles. En effet, en utilisant des fonctions qui ne mélangent pas les appels aux deux types de permutations, il est possible de précalculer les requêtes à la permutation publique afin d'améliorer le temps nécessaire en-ligne de l'attaque.

Nous introduisons ici une nouvelle idée qui résout ce problème. Nous définissons deux nouvelles fonctions F et f telles quelles :

$$F_\Pi(x) = x \oplus \Pi(x) \oplus \Pi(x \oplus \delta) \text{ et } f_\pi(x) = x \oplus \pi(x) \oplus \pi(x \oplus \delta).$$

Il est évident que deux chaînes construites en utilisant ces fonctions ne peuvent pas non plus fusionner car nous avons toujours deux fonctions différentes. Néanmoins, ces deux fonctions ont une autre propriété. Nous utiliserons ici une paire *slid* de messages clairs, *i.e.* une paire de messages clairs $(m, m^{'})$ tel que $m \oplus m^{'} = K_1$ ou $m \oplus m^{'} = K_1 \oplus \delta$.

Pour $m \oplus m^{'} = K_1$ et en utilisant les nouvelles fonctions F et f ci-dessus nous avons :

$$\begin{aligned} F_\Pi(m^{'} \oplus K_1) &= m^{'} \oplus K_1 \oplus \Pi(m^{'} \oplus K_1) \oplus \Pi(m^{'} \oplus K_1 \oplus \delta) \\ &= m^{'} \oplus K_1 \oplus K_2 \oplus \pi(m^{'} \oplus \cancel{K_1} \oplus \cancel{K_1}) \oplus \pi(m^{'} \oplus \cancel{K_1} \oplus \cancel{K_1} \oplus \delta) \\ &= m^{'} \oplus K_1 \oplus \pi(m^{'}) \oplus \pi(m^{'} \oplus \delta) \\ &= f_\pi(m^{'}) \oplus K_1. \end{aligned}$$

De même, pour $m \oplus m^{'} = K_1 \oplus \delta$:

$$F_\Pi(m^{'} \oplus K_1) = f_\pi(m^{'}) \oplus K_1 \oplus \delta.$$

Nous remarquons donc que, pour une paire *slid* de messages clairs, les fonctions F et f ne peuvent pas fusionner mais qu'elles deviennent parallèles, *i.e.* qu'elles ont une différence constante entre elles. En outre, cette différence entre les deux chaînes nous donne la clé K_1. En effet, elle vaut soit K_1 soit $K_1 \oplus \delta$.

Effectivement, soit deux points x et $x^{'}$ qui appartiennent aux chaines construites avec F_Π et f_π respectivement. En outre, soit x et $x^{'}$ une paire *slid* : $x \oplus x^{'} = K_1$ (ou $x \oplus x^{'} = K_1 \oplus \delta$). Quand $x \oplus x^{'} = K_1$, pour les points suivants $y = F_\Pi(x)$ et $y^{'} = f_\pi(x^{'})$, nous avons :

$$y = F_\Pi(x) = F_\Pi(x^{'} \oplus K_1) = f_\pi(x^{'}) \oplus K_1 = y^{'} \oplus K_1.$$

Nous en déduisons que $y = y^{'} \oplus K_1$, ce qui signifie que y et $y^{'}$ satisfont la même relation que x et $x^{'}$. Par conséquent, dès que deux points des chaînes F_Π et f_π forment par hasard une paire *slid*, les points suivants satisferont la même équation. De manière équivalente, pour $x \oplus x^{'} = K_1 \oplus \delta$, nous obtenons $y = y^{'} \oplus K_1 \oplus \delta$. La figure 3.9 illustre ce fonctionnement.

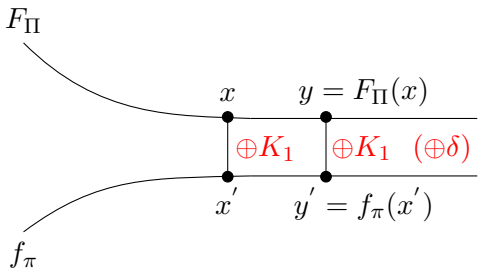

FIGURE 3.9 – Détection de chaînes parallèles.

3.2.2 La méthode des points distingués pour les chaînes parallèles

La question que nous nous posons naturellement est : comment pouvons-nous détecter les chaînes parallèles ? Nous montrons ici que, dans notre cas, la détection de chaînes parallèles est compatible avec la méthode de points distingués.

En effet, nous définissons qu'un point x de la chaîne F_Π est un point distingué si $\Pi(x) \oplus \Pi(x \oplus \delta)$ appartient à l'ensemble de points distingués $\mathcal{S}_0$. De même, un point $x^{'}$ de la chaîne f_π est un point distingué si $\pi(x^{'}) \oplus \pi(x^{'} \oplus \delta)$ appartient à $\mathcal{S}_0$. En outre, si $x \oplus x^{'} = K_1$, ou si $x \oplus x^{'} = K_1 \oplus \delta$, et si $x^{'}$ est un point distingué pour une chaîne f_π nous remarquons que :

$$\begin{aligned} \Pi(x) \oplus \Pi(x \oplus \delta) &= \cancel{K_2} \oplus \pi(x^{'} \oplus \cancel{K_1} \oplus \cancel{K_1}) \oplus \cancel{K_2} \oplus \pi(x^{'} \oplus \cancel{K_1} \oplus \delta \oplus \cancel{K_1}) \\ &= \pi(x^{'}) \oplus \pi(x^{'} \oplus \delta) \end{aligned}$$

Cela veut dire que le point x est un point distingué pour la chaîne F_Π. Pour détecter donc deux chaînes parallèles, il suffit de tester si $\Pi(x) \oplus \Pi(x \oplus \delta) = \pi(x^{'}) \oplus \pi(x^{'} \oplus \delta)$ pour deux points distingués x et $x^{'}$. Si nous détectons deux chaînes parallèles, nous saurons que $x \oplus x^{'}$ est un bon candidat pour la clé K_1 ou pour $K_1 \oplus \delta$.

Un des avantages de cette méthode est que comme les valeurs $\Pi(x) \oplus \Pi(x \oplus \delta)$ et $\pi(x^{'}) \oplus \pi(x^{'} \oplus \delta)$ sont nécessaires pour calculer l'élément suivant de chaque chaîne, en utilisant cette méthode nous n'ajoutons pas de coût supplémentaire. En outre, une différence importante par rapport à la recherche classique des collisions est que nous n'avons plus besoin de recalculer les chaînes depuis leur point de départ pour identifier leur point de fusion. En effet, il suffit de détecter les points distingués parallèles pour obtenir des candidats potentiels pour la clé K_1.

3.3 Attaque sur Even-Mansour en utilisant les chaînes parallèles

Après avoir présenté la méthode des chaînes parallèles et vu comment utiliser les points distingués pour les détecter, nous présentons maintenant une attaque générique sur Even-Mansour qui utilise cette nouvelle technique.

Comme nous l'avons déjà expliqué, notre but est de proposer une attaque sur Even-Mansour sans mémoire, qui ne mélange pas les appels aux deux types de permutations et qui fait $\mathcal{D}$ requêtes à la permutation avec clé avec $\mathcal{D} \ll N^{1/2}$.

3.3.1 Fonctionnement de l'attaque générique sur Even-Mansour

La première étape de l'attaque consiste à définir l'ensemble des points distingués $\mathcal{S}_0$ et construire les chaînes. En commençant par un point aléatoire x_0, nous construisons une chaîne en utilisant :

$$x_i = F_\Pi(x_{i-1}) = x_{i-1} \oplus \Pi(x_{i-1}) \oplus \Pi(x_{i-1} \oplus \delta) \text{ avec } i \geq 1, \ldots.$$

Si $\Pi(x_{i-1}) \oplus \Pi(x_{i-1} \oplus \delta)$ est un point distingué, nous arrêtons la construction de la chaîne et nous stockons le point distingué trouvé. De façon similaire, en commençant par un point aléatoire $x_0^{'}$, nous construisons des chaînes pour l'utilisateur public en utilisant :

$$x_j^{'} = f_\pi(x_{j-1}^{'}) = x_{j-1}^{'} \oplus \pi(x_{j-1}^{'}) \oplus \pi(x_{j-1}^{'} \oplus \delta) \text{ avec } j \geq 1, \ldots.$$

Si $\pi(x_{j-1}^{'}) \oplus \pi(x_{j-1}^{'} \oplus \delta)$ est un point distingué, nous arrêtons la construction de la chaîne et nous stockons le point distingué trouvé. Finalement, si $F_\Pi(x_i) = f_\pi(x_j^{'})$ nous avons deux chaînes parallèles et nous en déduisons que $x_i \oplus x_j^{'}$ est un bon candidat pour la clé K_1 (ou $x_i \oplus x_j^{'} = K_1 \oplus \delta$). L'algorithme 11 décrit le fonctionnement de cette attaque en pseudocode.

3.3.2 Analyse de la complexité

Nous devons maintenant évaluer le coût de cette attaque. Le tableau 3.1 résume la signification des notations utilisées.

Notation	Signification
$\mathcal{T}$	Requêtes à l'utilisateur public
$\mathcal{D}$	Requêtes aux utilisateurs avec clés
$\mathcal{M}$	Mémoire utilisée
$\mathcal{B}_\mathcal{T}$	Nombre de chaînes de l'utilisateur public
ℓ	Longueur de la chaîne construite

Tableau 3.1 – Notations utilisées.

Algorithme 11 Nouvelle attaque générique sur Even-Mansour.

ENTRÉES :

- La permutation publique π
- La permutation avec clé Π
- Une constante arbitraire δ
- L'ensemble de points distingués $\mathcal{S}_0$
- Le nombre $\mathcal{B}_{\mathcal{T}}$ des chaînes à construire pour l'utilisateur public

SORTIES :

- La clé K_1

{*Étape* 1 *: Calcul des chaînes pour l'utilisateur public.*}
pour $0 \leq k < \mathcal{B}_{\mathcal{T}}$ **faire**
  Choisir aléatoirement x_0 (différent pour chaque k)
  pour $0 \leq i < N^{1/2}$ **faire**
    $dp \leftarrow \pi(x_i) \oplus \pi(x_i \oplus \delta)$
    $x_{i+1} \leftarrow x_i \oplus dp$
    si $dp \in \mathcal{S}_0$ **alors**
      Stocker $\{x_{i+1}, dp\}$
      Sortie de la boucle
    fin si
  fin pour
fin pour

{*Étape* 2 *: Calcul de la chaîne pour l'utilisateur avec clé.*}
Choisir aléatoirement y_0
pour $0 \leq j < N^{1/2}$ **faire**
  $dp^{'} \leftarrow \Pi(y_j) \oplus \Pi(y_j \oplus \delta)$
  $y_{j+1} \leftarrow y_j \oplus dp^{'}$
  si $dp^{'} \in \mathcal{S}_0$ **alors**
    Stocker $\{y_0, dp^{'}\}$
    Sortie de la boucle
  fin si
fin pour

{*Étape* 3 *: Détection de la collision.*}
pour $\{x_{i+1}, dp\}$ stockés à l'étape 1 **faire**
  si $dp == dp^{'}$ **alors**
    retourne $K_1 \leftarrow x_{i+1} \oplus y_{j+1}$ et $K_1 \leftarrow x_{i+1} \oplus y_{j+1} \oplus \delta$
  fin si
fin pour

Nous supposons que le nombre de requêtes à l'utilisateur public $\mathcal{T}$ est plus grand que le nombre des requêtes $\mathcal{D}$. Effectivement, il s'agit du scénario le plus réaliste car en pratique il est plus facile de construire des chaînes pour l'utilisateur public.

Nous commençons l'analyse par le calcul de chaînes de l'utilisateur public. Dans ce cas, la longueur ℓ de nos chaînes doit être choisie pour satisfaire :

$$\mathcal{T} = \ell \cdot \mathcal{B}_{\mathcal{T}} \text{ et } N = \mathcal{B}_{\mathcal{T}} \cdot \ell^2.$$

Nous en déduisons que $\ell = N/\mathcal{T}$ et $\mathcal{B}_{\mathcal{T}} = \mathcal{T}^2/N$. En ce qui concerne la mémoire demandée, l'attaque utilise une mémoire de taille $\mathcal{O}(\mathcal{B}_{\mathcal{T}})$ pour stocker les chaînes de l'utilisateur public.

Concernant le calcul des chaînes avec clé, nous souhaitons faire $\mathcal{D}$ requêtes à l'utilisateur avec clé, c'est-à-dire $\mathcal{D}$ évaluations de la fonction F_{Π}. Par définition du schéma Even-Mansour nous avons $\mathcal{D} = N/\mathcal{T}$. En outre, le nombre de requêtes à l'utilisateur avec clé doit être égal à la longueur des chaînes construites, *i.e.* $\mathcal{D} = \ell$. Cela veut dire qu'en construisant une seule chaîne pour l'utilisateur avec clé, nous attendons qu'elle collisionne avec une des chaînes de l'utilisateur public.

Notre but est de faire une attaque sur Even-Mansour qui n'utilise pas beaucoup de mémoire. Nous souhaitons donc que $\mathcal{M} < \mathcal{D}$. De plus, nous savons que :

$$\mathcal{M} = \mathcal{T}/\mathcal{D} = \mathcal{T}/(N/\mathcal{T}) = \mathcal{T}^2/N = N/\mathcal{D}^2.$$

Pour que $\mathcal{M} < \mathcal{D}$, il nous faut donc $N < \mathcal{D}^3$. Le tableau 3.2 résume le coût de cette attaque.

Utilisateur public	
Longueur de chaînes	$\ell = N/\mathcal{T}$
Nombre de chaînes à construire	$\mathcal{B}_{\mathcal{T}} = \mathcal{T}^2/N$
Utilisateur avec clé	
Longueur de chaînes	$\ell = \mathcal{D} = N/\mathcal{T}$
Mémoire utilisée	$\mathcal{M} = \mathcal{T}^2/N = N/\mathcal{D}^2$

Tableau 3.2 – Résumé de l'analyse de l'attaque.

Montrons maintenant un cas typique. Soit $N^{1/3} < \mathcal{D} < N^{1/2}$. Si $\mathcal{D} = N^{2/5}$ et donc $\mathcal{T} = N^{3/5}$, nous en déduisons que $\mathcal{M} = N^{1/5}$ ce qui est plus petit que $\mathcal{D}$. Il en découle que cette attaque utilise $\mathcal{D} \ll N^{1/2}$ requêtes aux utilisateurs avec clés et donc, même si elle n'est pas complètement sans mémoire, la mémoire utilisée est plus petite que la taille des données.

En outre, il faut mentionner ici, que, quand nous construisons des chaînes, nous souhaitons éviter de calculer plusieurs fois la même chose. En effet, il ne faut pas con-

struire des chaînes dont la taille est supérieure à $N^{1/2}$ parce que ce genre de chaînes appartiennent normalement à un cycle. Néanmoins, il ne faut pas que les chaînes soient trop courtes non plus. Effectivement, quand nous construisons $\mathcal{B}_\mathcal{T}$ chaînes de taille ℓ chacune, il faut s'assurer que $\mathcal{B}_\mathcal{T} \cdot \ell^2$ ne sera pas plus grand que N [Jou09]. Autrement, plusieurs chaînes vont fusionner avec leurs prédécesseurs.[2]

Amélioration par rapport à Dunkelman *et al.*. Comme nous l'avons vu précédemment, la variation de l'attaque *slidex* sans mémoire de Dunkelman *et al.* a une complexité en temps et en données égale à $\mathcal{O}(2^{n/2})$. Dunkelman *et al.* souhaitent trouver une attaque également sans mémoire qui fait $\mathcal{D}$ requêtes à la permutation avec clé et $\mathcal{T} = N/\mathcal{D}$ requêtes à la permutation publique avec $\mathcal{D} \ll N^{1/2}$.

Avec notre attaque, nous répondons partiellement à ce problème. Notre attaque fait effectivement $\mathcal{D} = N^{2/5}$ requêtes à la permutation avec clé et $\mathcal{T} = N^{3/5}$ requêtes à la permutation publique. Par contre, notre attaque n'est pas complètement sans mémoire. Néanmoins, en faisant une phase de pré-calcul où nous calculons les chaînes de l'utilisateur public, la mémoire utilisée est $\mathcal{M} = N^{1/5}$ et donc plus petite que le $min(\mathcal{T}, \mathcal{D})$.

3.3.3 Différent compromis de l'attaque générique sur Even-Mansour

La même attaque peut aussi être appliquée avec un compromis légèrement différent[3]. Dans ce cas, les opérations de base de l'attaque restent identiques, cependant, en modifiant l'ordre d'exécution des opérations nous pouvons proposer une attaque qui utilise moins de mémoire que l'attaque précédente.

En effet, pour l'attaque générique nous utilisons une phase de pré-calcul. Pendant cette première phase de pré-calcul, nous construisons les chaînes pour l'utilisateur public. Toutes les chaînes construites sont stockés. Ensuite, nous commençons la phase de l'attaque en-ligne. La première étape de cette phase consiste à construire une chaîne pour l'utilisateur avec clé. Comme nous l'avons montré, nous attendons que cette chaîne collisionnera avec grande probabilité avec une des chaînes de l'utilisateur public stockée en mémoire pendant la phase du pré-calcul. Une fois qu'une telle collision est détectée, en utilisant les points d'arrivée de nos chaînes, nous arrivons à récupérer la première clé de blanchiment.

Nous montrons maintenant comment en modifiant légèrement les opérations de cette attaque, nous obtenons le même résultat mais en utilisant moins de mémoire que précédemment. Dans ce cas, nous ne commençons pas avec une phase de pré-calcul. La première étape consiste donc à construire une seule chaîne pour l'utilisateur que nous stockons en mémoire.

[2]Pour les collisions entre deux ensembles nous savons que : pour deux sous-ensembles de taille N_1 et N_2 respectivement d'un ensemble de taille N, le nombre des collisions attendu est égal à $\frac{N_1 \cdot N_2}{N}$. Donc, pour $\mathcal{B}_\mathcal{T}$ chaînes de taille ℓ chacune, il faut que $\mathcal{B}_\mathcal{T} \cdot \ell^2 < N$.

[3]Ce compromis nous a été indiqué par un relecteur anonyme à Asiacrypt 2014.

Algorithme 12 Différent compromis de l'attaque générique sur Even-Mansour.

ENTRÉES :

- La permutation publique π
- La permutation avec clé Π
- Une constante arbitraire δ
- L'ensemble de points distingués $\mathcal{S}_0$
- Le nombre $\mathcal{B}_{\mathcal{T}}$ des chaînes à construire pour l'utilisateur public

SORTIES :

- La clé K_1

{*Étape* 1 *: Calcul de la chaîne pour l'utilisateur avec clé.*}
Choisir aléatoirement x_0
pour $0 \leq i < N^{1/2}$ **faire**
  $dp \leftarrow \Pi(x_i) \oplus \Pi(x_j \oplus \delta)$
  $x_{i+1} \leftarrow x_i \oplus dp$
  si $dp \in \mathcal{S}_0$ **alors**
    Stocker $\{x_{i+1}, dp\}$
    Sortie de la boucle
  fin si
fin pour

{*Étape* 2 *: Calcul des chaînes pour l'utilisateur public.*}
pour $0 \leq k < \mathcal{B}_{\mathcal{T}}$ **faire**
  Choisir aléatoirement y_0 (différent pour chaque k)
  pour $0 \leq j < N^{1/2}$ **faire**
    $dp^{'} \leftarrow \pi(y_i) \oplus \pi(y_i \oplus \delta)$
    $y_{i+1} \leftarrow y_i \oplus dp^{'}$
    si $dp^{'} \in \mathcal{S}_0$ **alors**
      si $dp == dp^{'}$ **alors**
        retourne $K_1 \leftarrow x_{i+1} \oplus y_{j+1}$ et $K_1 \leftarrow x_{i+1} \oplus y_{j+1} \oplus \delta$
      fin si
      Sortie de la boucle
    fin si
  fin pour
fin pour

Pendant la deuxième phase, nous construisons les chaînes de l'utilisateur public. Après le calcul de chaque chaîne, nous vérifions si la chaîne construite collisionne avec la chaîne de l'utilisateur stockée. Si les chaînes ne collisionnent pas, nous construisons encore une chaîne pour l'utilisateur sans clé. Au cas contraire, si les chaînes collisionnent, nous arrêtons la phase de calcul des chaînes. Nous utilisons donc la collision trouvée pour récupérer la clé de l'utilisateur. À l'inverse, si les chaînes ne collisionnent pas, nous construisons encore une chaîne pour l'utilisateur public et, comme précédemment, nous vérifions si la nouvelle chaîne collisionnent avec la chaîne de l'utilisateur. Nous continuons de cette manière jusqu'à la détection de la collision cherchée.

L'avantage de ce compromis est que nous utilisons moins de mémoire que précédemment car nous stockons juste une chaîne. Cependant, pour ce compromis nous n'avons pas de phase de pré-calcul mais toutes nos étapes sont en-ligne. L'inconvénient est donc que la complexité en temps de cet algorithme est plus important que l'algorithme de base.

Pour décrire ce compromis nous pouvons se baser sur le pseudocode de l'algorithme 11 décrit ci-dessus. Néanmoins, il faut faire la deuxième étape en premier et ensuite comparer chaque chaîne de l'utilisateur public avec la chaîne stockée. L'algorithme 12 décrit ce compromis.

3.4 Attaques dans le modèle multi-utilisateurs

Nous montrons ici comment adapter l'attaque générique présentée précédemment dans un modèle multi-utilisateurs. Avant d'expliquer le fonctionnement de cette attaque, nous présentons le modèle multi-utilisateurs et son intérêt.

3.4.1 Description du modèle multi-utilisateurs

Souvent négligé en cryptographie, le modèle multi-utilisateurs est en réalité très intéressant et reflète bien certains aspects du monde réel. En effet, la sécurité de la plupart des systèmes cryptographiques est généralement étudiée lorsque nous avons un seul utilisateur, à l'exception de certains cas, comme les systèmes d'échange de clés, les algorithmes de chiffrement à clé publique et les signatures. Dans ce modèle classique avec un utilisateur unique, il existe une seule entité légitime qui peut chiffrer, déchiffrer, signer et authentifier les données. Le but de l'adversaire est donc d'attaquer cet utilisateur.

Néanmoins, le modèle classique avec un seul utilisateur ne tient pas compte d'un aspect important du monde réel : les systèmes cryptographiques sont conçus pour être utilisés par nombreux utilisateurs simultanément. Cela veut dire que plusieurs acteurs utilisent le même algorithme, chacun avec sa propre clé choisie indépendamment des autres. Par exemple, les algorithmes cryptographiques utilisés dans les cartes bleues pour le chiffrement de transactions sont les mêmes pour tous les utilisateurs. Cependant, chaque carte utilise sa propre clé pour le chiffrement de données.

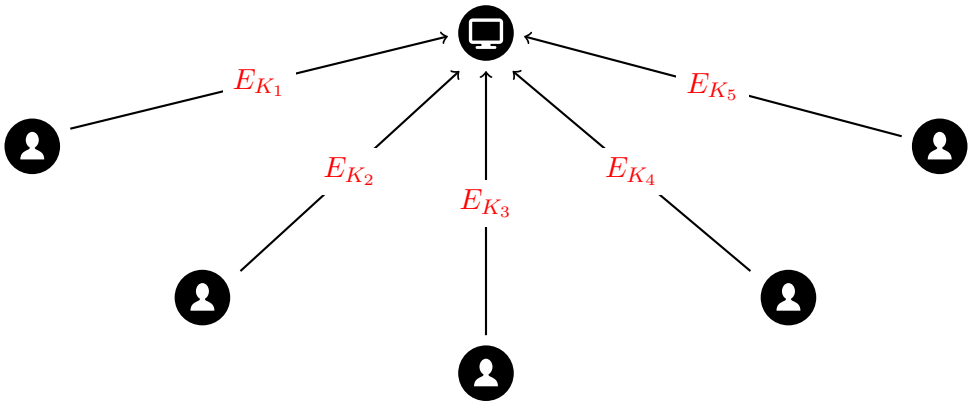

FIGURE 3.10 – Un exemple du modèle multi-utilisateurs : les cinq utilisateurs du groupe utilisent tous la fonction du chiffrement E pour communiquer avec un serveur central mais chaque utilisateur possède sa propre clé K_i ($1 \leq i \leq 5$).

Dans ce contexte, l'attaquant essaye de récupérer tout ou partie des clés plus efficacement que dans une attaque classique. Évidemment, une attaque en multi-utilisateurs ne peut pas coûter globalement moins cher que l'attaque d'un seul utilisateur. Le but de l'attaquant est ici de récupérer les clés des utilisateurs en faisant une attaque plus efficace que la complexité de l'attaque dans le modèle à un utilisateur, multipliée par le nombre d'utilisateurs. Cela peut être possible en amortissant le coût de l'attaque parmi les utilisateurs du groupe.

Pour mieux visualiser le modèle multi-utilisateurs et comprendre le but de l'attaquant, nous présentons ici le fonctionnement du scénario d'attaque contre les algorithmes MAC. Soit une famille de MAC H_K où $H_K : \{0,1\}^* \rightarrow \{0,1\}^t$ et $K \in \{0,1\}^k$ est la clé utilisée. Dans un modèle classique avec un seul utilisateur, il doit être difficile pour un attaquant, qui a accès à H_K, de générer une paire message-tag valide (m, τ) sans connaître la clé K. Dans le modéle multi-utilisateurs, nous supposons que l'attaquant veut attaquer un groupe de L utilisateurs en même temps. Chaque utilisateur possède sa propre clé $K^{(i)} \in \{0,1\}^k$, où $0 \leq i \leq (L-1)$, générée indépendamment des autres. Pour monter une attaque contre cette famille de MACs, l'attaquant doit avoir accès à l'oracle pour $H_{K^{(i)}}$. Son but est de produire un triplet (i, m, τ). Nous supposons que l'attaquant a réussi à effectuer une attaque dans le modèle à un utilisateur avec une complexité égale à E. En multi-utilisateurs, le but de l'attaquant est de récupérer la totalité des clés du groupe ou les clés d'une proportion θ de L utilisateurs en temps plus petit que $E \times L\theta$. En particulier, pour $\theta = 1/L$, cela permet de retrouver la clé d'un utilisateur parmi les l en un temps plus petit que le temps de l'attaque classique.

À EUROCRYPT 2012, Menezes a voulu souligner l'importance du modèle multi-utilisateurs. Pendant sa conférence invitée [Men12], il a souligné l'écart entre les preuves de sécurité pour le modèle à un utilisateur et le modèle multi-utilisateurs. Il a mon-

tré qu'il existe une réduction directe entre la preuve de sécurité pour un seul utilisateur et la preuve de sécurité pour L utilisateurs avec une probabilité de réussite divisé par L.

De tout cela, il en découle que le modèle multi-utilisateurs est un scénario qui correspond au contexte du monde réel et qui doit être étudié pendant la conception et l'analyse de cryptosystèmes. Certainement, ce modèle présente aussi des inconvénients. Par exemple, les attaques présentées par Menezes nécessitent en général un très grand nombre d'utilisateurs. Pour plus de résultats sur les attaques en multi-utilisateurs, le lecteur peut également se référer aux travaux de Bellare et Rogaway [BR94], qui ont été les premiers à donner quelques définitions pour ce modèle, mais aussi aux travaux de Biryukov, Mukhopadhyay et Sarkar [BMS05] et de Chatterjee, Menezes et Sarkar [CMS11].

3.4.2 Principe de l'algorithme

Dans cette section, nous montrons que l'attaque générique sur Even-Mansour présentée précédemment peut aussi s'appliquer dans un modèle multi-utilisateurs. En outre, nous présentons une nouvelle idée qui rend l'attaque plus efficace dans ce modèle.

Dans le cas du modèle multi-utilisateurs, nous avons L utilisateurs qui utilisent tous le schéma Even-Mansour basé sur la même permutation publique π. Chaque utilisateur $U^{(i)}$, avec $0 \leq i < L$, possède sa propre clé ($K^{(i)}$ dans le cas d'un Even-Mansour avec une seule clé ou $\{K_1^{(i)}, K_2^{(i)}\}$ dans le cas du schéma classique) choisie aléatoirement et indépendamment des clés des autres utilisateurs.

Évidemment, l'attaque précédente peut s'appliquer dans ce contexte. Néanmoins, l'objectif de l'attaquant peut varier. Un scénario possible est que l'adversaire souhaite récupérer la clé de tous les utilisateurs. Pour cela, chaque utilisateur effectue les requêtes nécessaires en utilisant sa propre permutation Π. En outre, les requêtes à la permutation publique sont réparties parmi les utilisateurs pour que chacun en effectue une partie. De cette façon, chaque utilisateur effectue quelques requêtes à la permutation publique et donc leur coût est amorti parmi les utilisateurs. Dans ce cas, en se basant sur l'analyse effectuée précédemment, pour un groupe de $L = N^{1/3}$ utilisateurs, nous faisons $\mathcal{T} = N^{2/3}$ requêtes à la permutation publique ($N^{1/3}$ chaînes de $N^{1/3}$ requêtes chacune, $\mathcal{M} = N^{1/3}$). Pour chaque utilisateur il faut faire $N^{1/3}$ requêtes à la permutation avec clé. Donc, le coût amorti pour chaque utilisateur est $N^{1/3}$ et, également, la mémoire utilisée est égale à $N^{1/3}$.

En outre, le but de l'adversaire pourrait être la récupération d'au moins une clé du groupe des utilisateurs. Pour cela, il suffit juste d'amortir les $\mathcal{D}$ requêtes à la permutation avec clé entre les utilisateurs.

Cependant, dans cette section, nous présentons un compromis différent dans le cas d'un modèle multi-utilisateurs. Comme pour l'attaque générique, pour chaque utilisateur nous construisons des chaînes en utilisant la fonction $F_\Pi(x) = x \oplus \Pi(x) \oplus \Pi(x \oplus \delta)$ comme elle a été définie précédemment. Nous arrêtons la construction des chaînes dès

qu'elle arrivent à un point distingué. Nous procédons de facon similaire pour l'utilisateur sans clé mais en utilisant la fonction $f_\pi(x) = x \oplus \pi(x) \oplus \pi(x \oplus \delta)$. Chaque collision trouvé nous donne le XOR des clés. Ensuite, nous construisons un graphe qui représente les utilisateurs du groupe et les collisions détectées entre eux.

Construction d'un graphe. Le but de la construction de graphe est de modéliser les relations entre les différentes clés des utilisateurs. Les utilisateurs du groupe sont représentés par les nœuds du graphe. Dès qu'une relation entre les clés de deux utilisateurs est détectée, nous mettons une arête entre les nœuds correspondants. En outre, nous ajoutons un utilisateur public sans clé, c'est-à-dire un utilisateur qui utilise seulement la permutation publique d'Even-Mansour.

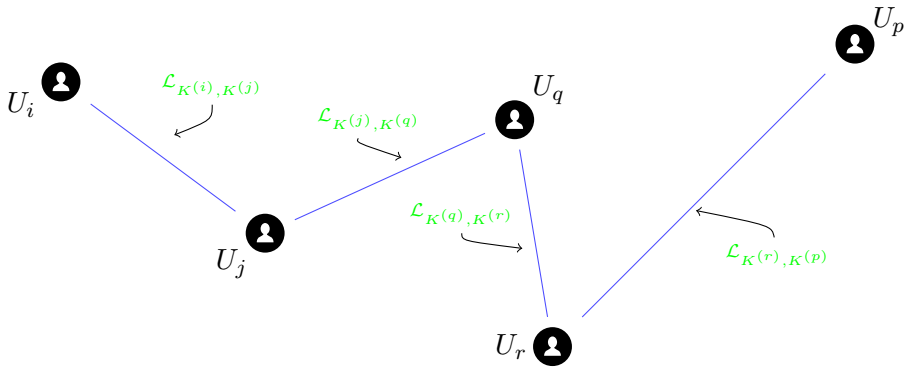

FIGURE 3.11 – Construction d'un graphe sans et avec l'utilisateur public.

Comme nous le verrons par la suite, quand le nombre d'arêtes est légèrement plus grand que le nombre d'utilisateurs, une composante géante apparaît et nous récupérons donc une grande partie des clés.

De plus, si une relation entre l'utilisateur public et un autre utilisateur du groupe est détectée, nous arrivons à récupérer les clés de tous les utilisateurs de la composante géante. L'avantage de notre proposition est que nous pouvons faire plusieurs calculs en parallèle. Cependant, dans notre cas, l'inconvénient est que nous récupérons les clés juste à la fin, quand une composante géante apparait dans le graphe.

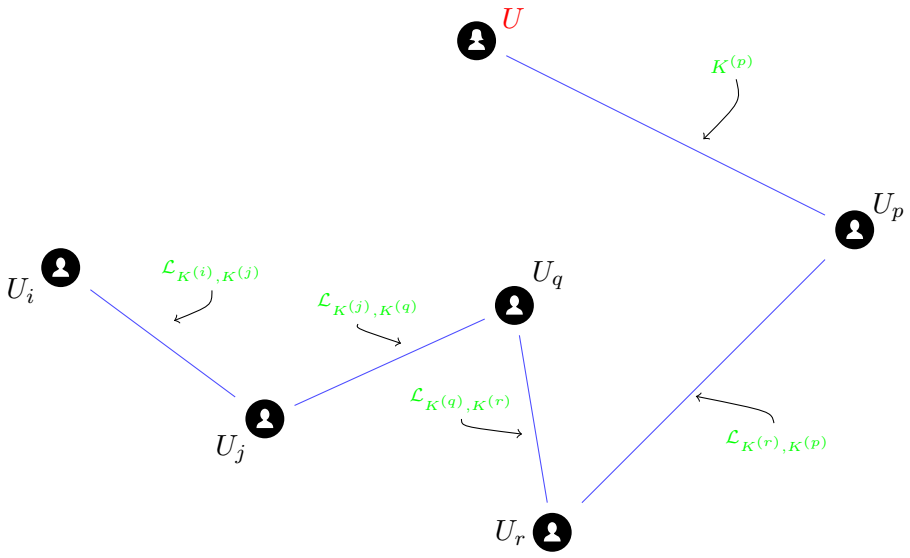

FIGURE 3.12 – Construction d'un graphe sans et avec l'utilisateur public.

3.4.3 Analyse par des propriétés des graphes aléatoires

Nous expliquons ici pourquoi l'idée du graphe aléatoire nous permet de récupérer les clés de la majorité des utilisateurs. Notre analyse utilise des propriétés des graphes aléatoires tirées de [Bol85, JLR00]. Nous ne présentons ici que les résultats qui nous intéressent.

Pour qu'un adversaire arrive à récupérer les clés de la majorité des utilisateurs il faut qu'une composante géante apparaisse avec grande probabilité dans notre graphe. Pour prouver cela, nous utilisons un résultat de la théorie des graphes aléatoires qui a déjà été utilisé pour des attaques similaires [CKM00, JPS03]. En effet, nous affirmons que ce graphe se comporte comme un graphe aléatoire selon le modèle Erdös-Rényi de graphes aléatoires.

Définition 3.1. *Dans le modèle de grapheErdös-Rényi $G\{n, Pr(edge) = p\}$ $(0 < p < 1)$, tous les graphes ont un ensemble de nœuds $V = \{1, 2, \ldots, n\}$, dans lequel les arêtes sont choisies indépendamment avec probabilité p. En d'autres termes, si G_0 est un graphe avec un ensemble de nœuds V et qu'il a m arêtes, alors :*

$$Pr(\{G_0\}) = Pr(G = G_0) = p^m q^{N-m},$$

où $q = 1 - p$ et $N = \binom{n}{2}$.

Définition 3.2. *Un processus de graphe aléatoire sur $V = \{1, 2, \ldots, n\}$ est une chaîne de Markov $\tilde{G} = (G_t)_0^\infty$, dont les états sont des graphes sur V. Le processus commence avec un graphe vide et, pour $1 \leq t \leq N$, le graphe G_t est obtenu à partir du graphe G_{t-1} en ajoutant*

une arête, tous les nouvelles arêtes étant équiprobables ; la seule restriction étant qu'une arête déjà choisie ne peut pas l'être de nouveau.

Tout d'abord, nous donnons deux théorèmes tirés de [Bol85, JLR00] sur la composante géante du graphe.

Théorème 3.3. *Soit $c > 1$ une constante, $t = \lfloor cn/2 \rfloor$ et $\omega(n) \to \infty$. Alors, presque tous les graphes G_t sont l'union de la composante géante, de petits composants uni-cycliques et de petits composants en forme d'arbre. L'ordre de la composante géante $L_1(G_t)$ satisfait :*

$$|L_1(G_t) - (1 - t(c))n| \leq \omega(n)n^{1/2},$$

où

$$t(c) = \frac{1}{c}\sum_{k=1}^{\infty} \frac{k^{k-1}}{k!}(ce^{-c})^k,$$

et pour tout $i \geq 2$

$$\left|L_i(G_t) - (1/\alpha)\Big(\log n - \frac{5}{2}\log\log n\Big)\right| \leq \omega(n),$$

où $\alpha = c - 1 - \log c$.

Si $t = \mathcal{O}(n \log n)$, alors le graphe est connecté de façon presque certaine.

Théorème 3.4. *Pour $t = \lfloor 2n \log n \rfloor$, nous avons :*

$$\Pr[G_t \text{ n'est pas connecté}] < n^{-n/4}.$$

Donc, dès que le nombre d'arêtes est plus grand que le nombre des nœuds, une composante géante, dont la taille est linéaire en le nombre d'arêtes, apparaît. Plus précisément, dans un graphe avec L nœuds et $cL/2$ arêtes placés de façon aléatoire, avec $c > 1$, une composante géante apparaît dès que la taille est presque égale à $(1 - t(c))L$, avec :

$$t(c) = \frac{1}{c}\sum_{k=1}^{\infty} \frac{k^{k-1}(ce^{-c})^k}{k!}.$$

Par exemple, si nous avons L nœuds et $cL/2$ arêtes aléatoires, avec $c = 4$, notre graphe contient une composante géante dont la taille fait 98% des points. D'après le thèoréme 3.4, pour $2L \log L$ arêtes aléatoires, tous les points du graphe appartiennent à la composante géante avec une très grande probabilité. Par conséquence, dès que le nombre d'arêtes atteint ce seuil, nous obtiendrons un graphe connexe. Si l'utilisateur public apparaît dans cette composante géante, nous récupérons les clés de tous les utilisateurs.

3.5 Attaque en multi-utilisateurs sur Even-Mansour

Nous présentons maintenant le fonctionnement de l'attaque sur le schéma Even-Mansour dans un modèle multi-utilisateurs et l'analyse de sa complexité.

3.5.1 Fonctionnement de l'attaque

Nous pouvons résumer le fonctionnement de l'attaque en cinq phases. Pendant la première étape de l'attaque, nous créons un nombre constant de chaînes pour chaque utilisateur en utilisant la fonction F_Π. La construction de chaque chaîne s'arrête quand elle arrive sur un point distingué. En outre, nous construisons aussi des chaînes pour l'utilisateur public en utilisant la fonction f_π et nous les arrêtons dès qu'elles arrivent sur un point distingué.

Chaque point distingué trouvé est stocké. Les deux phases suivantes consistent alors à trier tous les points distingués et à les réunir en sous-ensembles. Une collision trouvée entre deux points distingués nous donne le XOR de deux premières clés de blanchiment des utilisateurs.

Pendant la quatrième étape, nous construisons un graphe. Tous les utilisateurs du groupe, ainsi que l'utilisateur public, sont représentés par les nœuds du graphe. Chaque fois que nous obtenons une collision entre $F_\Pi^{(i)}(x)$ et $F_\Pi^{(j)}(y)$, pour deux utilisateurs $U^{(i)}, U^{(j)}$ et deux points x et y, nous ajoutons une arête entre les nœuds correspondants marquée par :

$$F_\Pi^{(i)}(x) \oplus F_\Pi^{(j)}(y) = K_1^{(i)} \oplus K_1^{(j)}$$

ou $F_\Pi^{(i)}(x) \oplus F_\Pi^{(j)}(y) = K^{(i)} \oplus K^{(j)}$ dans le cas du schéma avec une seule clé. Comme nous allons le voir lors de l'analyse de notre attaque, ce graphe deviendra connexe avec une grande probabilité.

Pour conclure l'attaque, il suffit de trouver une seule collision entre l'utilisateur public et un des utilisateurs avec clés, qui se trouve dans la composante géante, pour récupérer les clés de tous les utilisateurs du graphe.

Les algorithmes 13 et 14 expliquent en pseudocode le fonctionnement de cette attaque.

3.5.2 Analyse de l'attaque

Tout d'abord, il faut montrer que nous construisons un graphe qui peut être vu comme un graphe aléatoire qui suit le modèle Erdös-Rényi des graphes aléatoires. Comme nous l'avons expliqué précédemment, pour ce type des graphes, quand le nombre d'arêtes devient plus grand que le nombre d'utilisateurs, une composante géante apparaît. Il faut donc montrer que nous construisons un graphe aléatoire et que chaque arête de ce graphe est ajoutée indépendamment des autres. Nous définissons une version idéalisée de notre algorithme et nous montrons que l'attaque fonctionne aussi avec cette

version. Ensuite, nous prouvons que la version idéalisée et l'attaque sont équivalentes en utilisant des arguments de simulation.

Dans le modèle idéalisé, le simulateur choisit aléatoirement L clés $K_1, K_2, \ldots, K_L$. Nous définissons $\mathcal{S}_0$ l'ensemble qui contient des paires formées de points distingués d_i et de leur identifiant $id(d_i)$, *i.e.* $(d_i, id(d_i))$. Chaque identifiant est unique. Le simulateur itère la fonction $F_{\Pi}^{(i)}(x) = K_i \oplus f_\pi(x \oplus K_i)$ jusqu'à ce que $x_\ell \oplus K_i \in \mathcal{S}_0$ où ℓ est le nombre d'itérations de la fonction F_Π. Une fois qu'un point distingué est détecté, le simulateur retourne l'identifiant du point distingué trouvé et le point x_ℓ. Il est évident que K_i ne peut pas être récupérée à partir de l'information retournée par le simulateur. Pour prouver que l'attaque fonctionne dans ce modèle idéalisé, il suffit de voir que si deux utilisateurs ont le même identifiant, alors $x_\ell \oplus K_i = x_{\ell'} \oplus K_j$ et donc $x_\ell \oplus x_{\ell'} = K_i \oplus K_j$. Nous pouvons donc récupérer la même information qu'avec la véritable attaque.

Finalement, il faut prouver que le simulateur n'a pas besoin de connaître $F_{\Pi}^{(i)}$ mais qu'il peut retourner l'information demandée en utilisant uniquement la fonction publique f_π. En outre, nous prouvons que les sorties du simulateur ne peuvent pas être distinguées des sorties du modèle idéalisé. Le simulateur commence par générer aléatoirement L clés pour le schéma Even-Mansour. Nous montrons que pour chaque clé K_i générée, la paire $(d_i, id(d_i))$ peut être trouvée en utilisant seulement la fonction publique f_π.

En effet, soit x_ℓ la ℓ^{e} itération de la fonction $F_{\Pi}^{(i)}$, à partir d'un point de départ x_0, en utilisant la clé K_i. Le simulateur peut calculer la valeur $x_\ell \oplus K_i$ en itérant la fonction f_π à partir d'un point de départ $x_0 \oplus K_i$. Il en découle que pour générer les paires points distingués et leurs identifiants, le simulateur peut calculer $(x_\ell \oplus K_i, id(x_\ell))$ sans avoir besoin de la fonction F_Π des utilisateurs. Les paires sont donc générées aléatoirement sans connaître la fonction F_π. En outre, comme la fonction f_π est une fonction aléatoire, les arêtes du graphe sont ajoutées aléatoirement et indépendamment les unes des autres. Par conséquent, le graphe construit est un graphe qui suit le modèle de graphe aléatoire de Erdös-Rényi.

Finalement, nous calculons la complexité de cette attaque. Dans un groupe de L utilisateurs, si nous construisons $c/2$ chaînes de longueur ℓ pour chaque utilisateur, la première étape exige $cL/2$ en mémoire avec une complexité en temps égale à $c\ell$ opérations par utilisateur. Toutes les étapes suivantes peuvent être effectuées en temps linéaire en le nombre des utilisateurs L.

Typiquement, pour un groupe de $N^{1/3}$ utilisateurs, avec c une constate petite et arbitraire, il nous faut $c \cdot N^{1/3}$ requêtes par utilisateur et $N^{1/3}$ requêtes à l'utilisateur public pour récupérer presque toutes les $N^{1/3}$ clés avec une grande possibilité. Si vous souhaitons récupérer les clés de tous les utilisateurs, il nous faut $N^{1/3+\epsilon}$ arêtes pour connecter tous les utilisateurs.

Algorithme 13 Attaque sur Even-Mansour dans un modèle multi-utilisateurs : client.

ENTRÉES :

- La permutation avec clé Π
- L'ensemble de points distingués $\mathcal{S}_0$
- Une constante arbitraire δ
- Le nombre $\mathcal{B}$ des chaînes à construire pour l'utilisateur

{*Étape* 1 *: Calcul des chaînes pour l'utilisateur avec clé.*}
pour $0 \leq l < \mathcal{B}$ **faire**
  Choisir aléatoirement x_0
  pour $0 \leq i < N^{1/3}$ **faire**
    $A \leftarrow \Pi(x_i) \oplus \Pi(x_i \oplus \delta)$
    $x_{i+1} \leftarrow x_i \oplus A$
    si $A \in \mathcal{S}_0$ **alors**
      Stocker $\{x_{i+1}, A\}$
      Sortie de la boucle
    fin si
  fin pour
fin pour

{*Étape* 2 *: Envoyer les données au serveur.*}
Activer la connexion avec le serveur
Envoyer tous les $\{x_{i+1}, A\}$ stockés au serveur
Terminer la connexion avec le serveur

Algorithme 14 Attaque sur Even-Mansour dans un modèle multi-utilisateurs : serveur.

ENTRÉES :

- Les données envoyées par les clients
- La permutation sans clé π
- L'ensemble de points distingués $\mathcal{S}_0$
- Une constante arbitraire δ
- Le nombre $\mathcal{B}$ des chaînes à construire pour l'utilisateur public

SORTIES :

- La clé K_1

{*Étape* 1 *: Calcul des chaînes pour l'utilisateur public.*}
pour $0 \leq l < \mathcal{B}$ **faire**
  Choisir aléatoirement y_0
  pour $0 \leq j < N^{1/3}$ **faire**
    $C \leftarrow \pi(y_j) \oplus \pi(y_j \oplus \delta)$
    $y_{j+1} \leftarrow y_j \oplus C$
    si $C \in \mathcal{S}_0$ **alors**
      Stocker $\{y_{j+1}, C\}$
      Sortie de la boucle
    fin si
  fin pour
fin pour

{*Étape* 2 *: Recevoir, stocker et trier les données des clients.*}
pour chaque $\{x_{i+1}, A\}$ reçu **faire**
  Stocker dans un tableau $T[\,][\,]$ le couple $\{x_{i+1}, A\}$
fin pour

Trier le tableau selon une relation d'ordre sur les points distingués

Regrouper les éléments correspondants au même point distingué en un sous-ensemble

{*Étape* 3 *: Construire le graphe.*}
Créer l'ensemble V_0 des utilisateurs atteignables par l'utilisateur public
tant que $V_{n-1}! = V_n$ **faire**
  $V_n = V_{n-1} \cup \{$utilisateurs atteignables par les utilisateurs de $V_{n-1}\}$
fin tant que

{*Étape* 4 *: Récupérer les clés.*}
Commençant par l'utilisateur public, parcourir la composante géante et récupérer les clés

3.5.3 Résultats d'implémentation

Nous souhaitons prouver l'exactitude de notre analyse en implémentant l'attaque. Pour cela, nous avons implémenté le schéma Even-Mansour qui utilise comme permutation publique l'algorithme DES avec une clé fixe et $n = 64$ bits. Pour un groupe de 2^{22} utilisateurs nous avons créé 8 chaînes pour chaque utilisateur et 80 chaînes pour l'utilisateur public. Les points distingués utilisés contiennent 21 zéros et nous fixons la longueur maximale de nos chaînes à 2^{24} itérations. Cela veut dire, que nous abandonnons la construction des chaînes qui n'arrivent pas à un point distingué après 2^{24} évaluations. Au total, nous avons généré $33\,543\,077$ chaînes et le nombre d'ensembles trouvés qui contiennent au moins deux chaînes parallèles est $4\,109\,961$. La composante géante trouvée contient $3\,788\,059$ utilisateurs parmi les $4\,194\,304$ du groupe. Cela veut dire que nous déduisons les clés de 90% d'utilisateurs de notre groupe.

Effectivement, nous pouvons remarquer que nous ne récupérons pas le 98% des clés comme cela a été montré lors de l'analyse de notre attaque. Cependant, pour notre implémentation, le nombre d'arêtes utilisées est plus petit que le nombre de nœuds. Pour obtenir ce pourcentage de clés, il aurait fallu avoir deux fois plus d'arêtes. Les résultats expérimentaux sont donc cohérents avec l'analyse théorique de notre attaque.

Finalement, pour générer ce nombre des chaînes il a fallu 1600 secondes en utilisant 4096 cœurs en parallèle et l'analyse du graphe a été fait en quelques minutes en utilisant un ordinateur standard.

4. Applications à diverses primitives symétriques

Dans ce chapitre, nous montrons que les attaques génériques sur Even-Mansour présentées au chapitre 3 peuvent avoir des applications sur d'autres algorithmes. Nous présentons des attaques de récuperation de clé contre les algorithmes de chiffrement par blocs DESX et PRINCE [FJM14] mais également contre le code d'authentification de messages Chaskey [Mav15]. Comme nous allons le voir par la suite, ces algorithmes

utilisent la structure du schéma Even-Mansour. Évidemment, ces attaques ont des similarités avec l'attaque de base sur le schéma Even-Mansour. Néanmoins, DESX, PRINCE et Chaskey utilisent une clé interne supplémentaire qui n'existe pas sur Even-Mansour et qui influence le modèle d'attaque final.

4.1 Attaques sur l'algorithme de chiffrement par blocs à bas coût PRINCE

Dans le cadre de la cryptographie à bas coût, de nombreux algorithmes de chiffrement par blocs ont été proposés. Parmi eux, PRINCE [BCG$^+$12], a attiré l'intérêt de plusieurs chercheurs. Il a aussi été étudié pendant cette thèse. Il s'agit d'un algorithme de chiffrement par blocs à bas coût et il est basé sur la construction FX. La motivation principale derrière la proposition de cet algorithme était de construire un algorithme de chiffrement par blocs à faible latence qui peut chiffrer un message instantanément et dont les procédures de chiffrement et de déchiffrement sont à bas coût. Cela n'est pas évident en utilisant les cryptosystèmes actuels. Les auteurs comparent leur algorithme avec AES mais aussi avec d'autres algorithmes de chiffrement par blocs à bas coût, comme par exemple PRESENT. Ils justifient leur construction en montrant que PRINCE utilise entre 6 et 7 fois moins de place que PRESENT-80 et entre 14 et 15 fois moins que l'AES-128.

Nous proposons ici deux attaques génériques sur PRINCE qui opèrent sur le nombre total de tours. Plusieurs attaques (Tableau 4.1) ont été proposées sur cet algorithme et qui attaquent certains ou la totalité des tours de l'algorithme. Cependant, nos nouvelles attaques ont une complexité en temps particulièrement faible.

Elles sont basées sur l'attaque générique présentée au Chapitre 3 sur Even-Mansour mais, dans le cas de PRINCE, il fallait prendre en compte la clé secrète utilisée par la permutation interne. Cela veut dire que la permutation interne n'est plus connue mais elle diffère pour chaque utilisateur comme la clé de chaque utilisateur est différente. Cependant, nous avons pu contourner ce problème en utilisant la propriété de α-*reflection* de PRINCE et la spécificité du cadencement de clé de l'algorithme, *i.e.* la relation entre les deux clés de blanchiment.

Notre première attaque s'applique à un modèle multi-utilisateurs et récupère la clé de deux utilisateurs parmi un ensemble de 2^{32} utilisateurs en temps 2^{65}. Ensuite, nous présentons une autre attaque générique dans le modèle classique avec un seul utilisateur et, après un pré-calcul de complexité en temps 2^{96} et en mémoire 2^{64}, nous récupérons la clé de l'utilisateur en 2^{32} opérations. Néanmoins, il faut mentionner que notre attaque dans le modèle classique ne contredit pas la sécurité prouvée dans le papier original de PRINCE. Notre but est de montrer que des compromis différents peuvent être envisagés.

4.1.1 Description du fonctionnement de PRINCE

PRINCE a été proposé à Asiacrypt 2012 par Borghoff *et al.*. Il utilise un bloc de 64 bits et une clé de 128 bits divisée en deux clés équivalentes de 64 bits :

$$K = K_0 \| K_1$$

qui est étendue à 192 bits grâce à une troisième clé $K_0^{'}$:

$$K = (K_0 \| K_1) \rightarrow (K_0 \| K_0^{'} \| K_1).$$

La clé $K_0^{'}$ est dérivée de la clé K_0 en utilisant la fonction linéaire $L^{'}$:

$$L'(K_0) = (K_0 \ggg 1) \oplus (K_0 \gg 63),$$

où $\gg$ indique le décalage à droite et $\ggg$ la rotation d'un mot de 64 bits. Les clés K_0 et $K_0^{'}$ sont utilisées comme clés de blanchiment. La clé K_1 est utilisée par la fonction interne de PRINCE qui fait 12 tours et qui est appelée PRINCE$_{core}$. Pour simplifier la lecture, la fonction interne PRINCE$_{core}$ sera appelée par la suite la fonction interne de PRINCE et nous la noterons Pcore.

Pour chiffrer chaque message clair m, PRINCE utilise la fonction de chiffrement :

$$E_K(P) = K_0^{'} \oplus \text{Pcore}_{K_1}(P \oplus K_0)$$

où Pcore utilise la clé K_1. La figure 4.1 décrit ce fonctionnement.

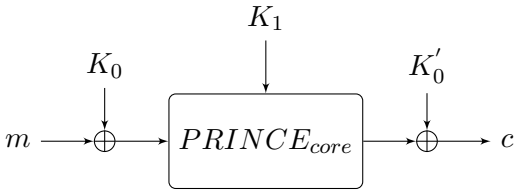

FIGURE 4.1 – L'algorithme de chiffrement par blocs PRINCE

La fonction interne de PRINCE (voir figure 4.2) contient cinq principales étapes :

1. une addition de la clé K_1 et une addition de la constante du tour RC_0 ;
2. cinq fonctions de tour R ;
3. un tour du milieu ;
4. cinq fonctions de tour inverse R^{-1} ;

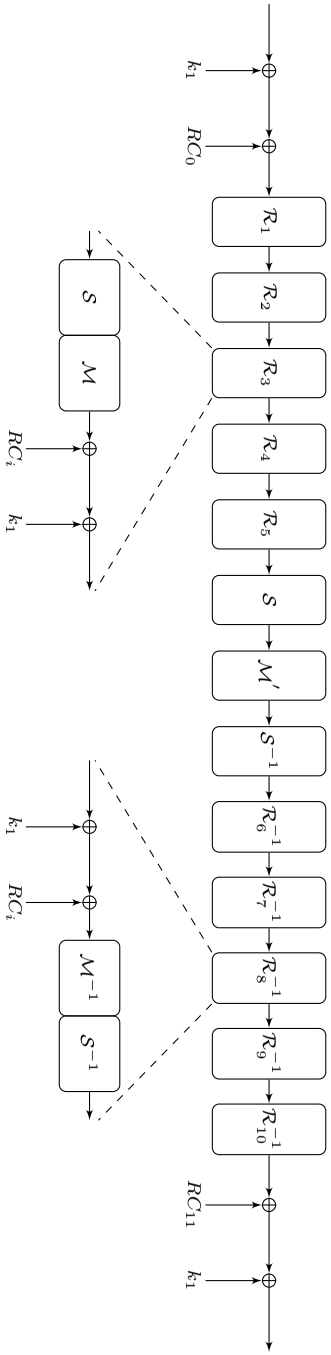

FIGURE 4.2 – Structure de PRINCE$_{core}$

5. l'addition de la constante du tour RC_{11} et de la clé K_1.

Chaque fonction de tour R contient une couche non-linéaire, une couche linéaire et finalement les XORs de la constante du tour RC_i avec $1 \leq i \leq 5$ et de la clé K_1.

La couche non-linéaire est l'application de la boite S de 4 bits :

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4

La couche linéaire correspondant à l'application de la matrice 64×64 M où $M = SR \circ M^{'}$ et SR est la permutation suivante :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\longrightarrow$	0	5	10	15	4	9	14	3	8	13	2	7	12	1	6	11

Les cinq dernières fonctions de tour R^{-1} font les opérations inverses de la procédure décrite précédemment. Donc, l'application de la couche $M^{'}$ au tour du milieu peut être vue comme un miroir.

Pour construire la matrice 64×64 diagonale par blocs $M^{'}$, il faut d'abord définir les matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

et

$$\hat{M}_{(0)} = \begin{pmatrix} M_0 & M_1 & M_2 & M_3 \\ M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \end{pmatrix} \quad \hat{M}_{(1)} = \begin{pmatrix} M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \\ M_0 & M_1 & M_2 & M_3 \end{pmatrix}.$$

La matrice $M^{'}$ est alors une matrice diagonale où $(\hat{M}^{(0)}, \hat{M}^{(1)}, \hat{M}^{(1)}, \hat{M}^{(0)})$ est sa diagonale principale. En outre, $M^{'}$ est une involution avec 2^{32} points fixes.

À chaque tour, une constante RC_i de 64 bits est XORée à l'état. Les constantes en hexadécimal utilisées sont les suivantes :

RC_0	0000000000000000
RC_1	13198a2e03707344
RC_2	a4093822299f31d0
RC_3	082efa98ec4e6c89
RC_4	452821e638d01377
RC_5	be5466cf34e90c6c
RC_6	7ef84f78fd955cb1
RC_7	85840851f1ac43aa
RC_8	c882d32f25323c54
RC_9	64a51195e0e3610d
RC_{10}	d3b5a399ca0c2399
RC_{11}	c0ac29b7c97c50dd

Par construction, pour chaque i entre 0 et 11, $RC_i \oplus RC_{11-i}$ est égal à la constante $\alpha =$ 0xc0ac29b7c97c50dd. De cette propriété et du fait que la matrice $M^{'}$ est une involution, on déduit que la fonction de déchiffrement de PRINCE est la même que la fonction de chiffrement en inversant l'ordre des clés k_0 et $k_0^{'}$ et en utilisant la clé $k_1 \oplus \alpha$ à la place de k_1. Cela signifie que pour chaque clé $(k_0 \| k_0^{'} \| k_1)$, nous avons :

$$D_{(k_0 \| k_0' \| k_1)}(\cdot) = E_{(k_0' \| k_0 \| k_1 \oplus \alpha)}(\cdot).$$

Cette propriété est primordiale au fonctionnement de PRINCE et est appelée propriété de α-*reflection* de PRINCE. Les auteurs montrent que la propriété de α-*reflection* n'introduit pas d'attaques générique avec une complexité nettement inférieure à celle des attaques génériques déjà connues sur la construction FX.

Les *reflection ciphers* et les variantes de PRINCE.

La propriété de α-*reflection* de PRINCE a motivé la définition d'une nouvelle catégorie d'algorithmes de chiffrement appelés *reflection ciphers*.

Définition 4.1. *Un algorithme de chiffrement par blocs E est un reflection cipher s'il existe une permutation P de l'espace de clés telle que, pour chaque clé K :*

$$(E_K)^{-1} = E_{P(K)}.$$

La permutation P est appelée permutation de couplage.

La catégorie des *reflection ciphers* contient toutes les constructions dont la fonction de tour est une fonction involutive, comme par exemple les réseaux de Feistel. Les propriétés de cette catégorie d'algorithmes et surtout la construction de la permutation de couplage ont été étudiées récemment [BCKL15]. Plus précisément, la permutation de couplage doit être une involution et les points fixes doivent être évités. En outre, dans le modèle classique avec un seul utilisateur, les distingueurs à clé reliée en général peuvent avoir un impact à la sécurité de ce type d'algorithmes.

De plus, les auteurs proposent trois variantes de la permutation de couplage de PRINCE$_{core}$. Comme expliqué précédemment, sur la version initiale de PRINCE$_{core}$ la permutation de couplage est de la forme $P(k) = k \oplus \alpha$ où k représente la clé k_1 qui est utilisée dans la fonction interne. Pour toutes les variantes, la clé k de n bits est remplacée par une clé de $2n$ bits. Dans le cas de PRINCE, cela veut dire que la clé k de 64 bits est remplacée par une clé de 128 bits. En outre, la clé k est séparée en deux parties égales : $k = (k^{'}, k^{''})$.

Pour la première variante, la permutation de couplage est une permutation non-linéaire de la forme :

$$P(k) = P(k^{'}, k^{''}) = (P_0(k^{'}), P_0(k^{''}))$$

où P_0 est une permutation non-linéaire.

Pour la deuxième variante, la permutation de couplage est :

$$P(k) = P(k^{'}, k^{''}) = (P_0(k^{'}), P_1(k^{''}))$$

où P_0 et P_1 sont deux permutations affines.

Finalement, dans la troisième variante, la correspondance du couplage est basée sur une permutation de bits.

4.1.2 Analyse de la sécurité de PRINCE

Les auteurs de PRINCE montrent que la sécurité de leur algorithme est assurée jusqu'à 2^{127-n} appels à la fonction de chiffrement ou de déchiffrement pour un attaquant qui arrive à récupérer 2^n paires clair/chiffré. Cette borne est valable uniquement pour le modèle avec une seule clé. En outre, les auteurs ne donnent aucune information concernant la sécurité des attaques à clés reliées et à clé connue ou choisie.

Mis à part la cryptanalyse du papier initial de PRINCE, plusieurs attaques ont été proposées sur cet algorithme. Le tableau suivant 4.1 résume les principales attaques existantes. Tout d'abord, en 2012, Abed, List and Lucks [ALL12] ont décrit une attaque sur la fonction interne Pcore qui utilise des techniques de cryptanalyse biclique. Il s'agit d'une attaque sur les 12 tours de PRINCE dont la complexité est légèrement meilleure que la complexité de la recherche exhaustive sur la clé utilisée. À FSE 2013, Jean *et al.* [JNP+13] ont réduit la borne de sécurité à 2^{126} avec une seule paire clair/chiffré. En outre, ils ont présenté deux attaques intégrales sur 6 tours (une sur la fonction interne Pcore et une sur la fonction complète de PRINCE) et une attaque *boomerang* sur 12 tours mais qui se porte sur une variante de PRINCE avec une autre valeur de α choisie par les auteurs. Pendant la même conférence, Soleimany *et al.* [SBY+13] ont présenté des attaques sur 12 tours en utilisant des valeurs choisies de α et une attaque à 6 tours pour sa valeur originale. À Crypto 2013, Canteaut, Naya-Plasencia et Vayssière [CNV13] ont mis en évidence une attaque *sieve-in-the-middle* sur 8 tours. En utilisant des techniques des attaques *meet-in-the-middle*, Li, Jia et Wang [LJW13] ont présenté une attaque sur 9 tours. Finalement, Canteaut *et al.* [CFG+14] ont amélioré les attaques sur PRINCE en

proposant une attaque sur 10 tours basée sur les différentielles multiples. La complexité temps-mémoire de l'attaque est $2^{118.56}$.

Les concepteurs de PRINCE ont aussi créé un challenge[1] dans le but de trouver des attaques pratiques contre leur algorithme. Plus précisément, au lieu d'avoir des attaques qui ne peuvent pas être appliquées dans un vrai contexte mais qui opèrent à un grand nombre de tours, les concepteurs de PRINCE cherchent à avoir plutôt des implémentation d'attaques pratiques. Derbez et Perrin [DP15] ont gagné le premier et deuxième tour de la compétition. En utilisant des attaques *meet-in-the-middle*, ils attaquent 6 tours en temps en ligne $2^{33.7}$ et 8 tours en temps en ligne $2^{65.7}$. Ils présentent aussi une attaque sur les 10 tours de PRINCE en temps en ligne 2^{68}.

4.1.3 Généralités et notations utilisées

Comme précédemment, l'idée de base de l'attaque est d'utiliser la méthode de points distingués. Plus précisément, nous cherchons à construire des chaînes et détecter éventuelles collisions en utilisant la méthode des points distingués. Une collision entre les points distingués nous montre que les chaînes correspondantes sont devenues parallèles et nous donne le XOR des clés.

Pour construire nos chaînes nous avons besoin de définir une fonction qui utilise PRINCE. Pour cela, pour chaque utilisateur $U^{(i)}$, nous définissons la fonction :

$$F_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x) = x \oplus \text{ PRINCE}_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x) \oplus \text{ PRINCE}_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x \oplus \delta)$$

où δ est une constante arbitraire non nulle.

Nous observons que la clé $K_0^{'(i)}$ peut disparaitre de l'équation F :

$$\begin{aligned} F_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x) &= x \oplus \text{ PRINCE}_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x) \oplus \text{ PRINCE}_{K_0^{(i)},K_0'^{(i)},K_1^{(i)}}(x \oplus \delta) \\ &= x \oplus \cancel{K_0^{'(i)}} \oplus \text{ Pcore}_{K_1^{(i)}}(x \oplus K_0^{(i)}) \oplus \cancel{K_0^{'(i)}} \oplus \text{ Pcore}_{K_1^{(i)}}(x \oplus K_0^{(i)} \oplus \delta) \\ &= x \oplus \text{ Pcore}_{K_1^{(i)}}(x \oplus K_0^{(i)}) \oplus \text{ Pcore}_{K_1^{(i)}}(x \oplus K_0^{(i)} \oplus \delta) \end{aligned}$$

Pour construire nos chaînes, nous utiliserons donc la suite :

$$x_{j+1} = F_{K_0^{(i)},K_1^{(i)}}(x_j) = x_j \oplus \text{ Pcore}_{K_1^{(i)}}(x_j \oplus K_0^{(i)}) \oplus \text{ Pcore}_{K_1^{(i)}}(x_j \oplus K_0^{(i)} \oplus \delta)$$

pour $j = 0, 1, 2, \ldots$ et x_0 un point de départ aléatoire.

À partir de cette fonction F, pour chaque utilisateur $U^{(i)}$, nous distinguons deux types différents de chaines : une chaîne de chiffrement $\mathcal{E}$ et une chaîne de déchiffrement

[1] https ://www.emsec.rub.de/research/research_startseite/prince-challenge/

Type d'attaque	**Référence**	**Tours**	**Données $\mathcal{D}$**	**Temps $\mathcal{T}$**	**$\mathcal{D} \times \mathcal{T}$**	**Mémoire**
intégrale	[JNP+13]	6	2^{16}	2^{64}	2^{80}	2^{16}
intégrale (PRINCE$_{core}$)	[JNP+13]	6	2^{16}	2^{30}	2^{46}	2^{16}
meet-in-the-middle	[DP15]	6	2^{16}	$2^{33.7}$	$2^{49.7}$	$2^{31.9}$
meet-in-the-middle	[DP15]	8	2^{16}	$2^{50.7}$ (en-ligne)	$2^{66.7}$ ($\mathcal{T}$ en-ligne)	$2^{84.9}$
meet-in-the-middle	[DP15]	8	2^{16}	$2^{65.7}$ (en-ligne)	$2^{81.7}$ ($\mathcal{T}$ en-ligne)	$2^{68.9}$
sieve-in-the-middle	[CNV13]	8	1	2^{123}	2^{123}	2^{20}
meet-in-the-middle	[LJW13]	9	2^{57}	2^{64}	2^{121}	$2^{57.3}$
différentielle multiple	[CFG+14]	9	$2^{46.89}$	$2^{51.21}$	$2^{98.10}$	$2^{52.21}$
différentielle multiple	[CFG+14]	10	$2^{57.94}$	$2^{60.62}$	$2^{118.56}$	$2^{61.52}$
meet-in-the-middle	[DP15]	10	2^{57}	2^{68} (en-ligne)	2^{125} ($\mathcal{T}$ en-ligne)	2^{41}
boomerang (α choisi)	[JNP+13]	12	2^{41}	2^{41}	2^{82}	-
biclique (PRINCE$_{core}$)	[ALL12]	12	2^{40}	$2^{62.72}$	$2^{102.72}$	2^{8}

Tableau 4.1 – Sommaire des attaques existantes sur PRINCE.

$\mathcal{D}$. Pour $\mathcal{E}$ nous utilisons la fonction de chiffrement de PRINCE tandis que pour $\mathcal{D}$ nous utilisons la fonction de déchiffrement. Pour chaque utilisateur $U^{(i)}$, nous définissons donc sa chaîne de chiffrement et sa chaîne de déchiffrement comme suit :

$$\mathcal{E}_{K_0^{(i)},K_1^{(i)}}(x_j^{(i)}) = x_{j+1}^{(i)} = x_j^{(i)} \oplus \text{Pcore}_{K_1^{(i)}}(x_j^{(i)} \oplus K_0^{(i)}) \oplus \text{Pcore}_{K_1^{(i)}}(x_j^{(i)} \oplus K_0^{(i)} \oplus \delta)$$

$$\begin{aligned}\mathcal{D}_{K_0'^{(i)},K_1^{(i)}\oplus\alpha}(y_j^{(i)}) &= y_{j+1}^{(i)} \\ &= y_j^{(i)} \oplus \text{Pcore}_{K_1^{(i)}\oplus\alpha}(y_j^{(i)} \oplus K_0'^{(i)}) \oplus \text{Pcore}_{K_1^{(i)}\oplus\alpha}(y_j^{(i)} \oplus K_0'^{(i)} \oplus \delta).\end{aligned}$$

Pour faciliter la lecture, nous définissons aussi les fonctions :

$$f^{\mathcal{E}} = \text{Pcore}_{K_1^{(i)}}(x_j^{(i)} \oplus K_0^{(i)}) \oplus \text{Pcore}_{K_1^{(i)}}(x_j^{(i)} \oplus K_0^{(i)} \oplus \delta) \text{ et}$$

$$f^{\mathcal{D}} = \text{Pcore}_{K_1^{(i)}\oplus\alpha}(y_j^{(i)} \oplus K_0'^{(i)}) \oplus \text{Pcore}_{K_1^{(i)}\oplus\alpha}(y_j^{(i)} \oplus K_0'^{(i)} \oplus \delta).$$

4.1.4 Attaque dans un modèle multi-utilisateurs

Pour notre première attaque nous nous plaçons dans un contexte multi-utilisateur. L'idée de base est l'attaque générique sur Even-Mansour dans un modèle multi-utilisateurs. Nous avons donc L utilisateurs distincts qui utilisent tous l'algorithme PRINCE. Chaque utilisateur $U^{(i)}$, avec $0 \leq i < L$, choisit sa clé $K^{(i)} = K_0^{(i)} \| K_1^{(i)}$ indépendamment des autres. Cependant, ici, nous n'avons pas d'utilisateur public car la fonction interne de PRINCE n'est pas publique.

Avant d'expliquer le fonctionnement de l'attaque, nous donnons les définitions des fonctions et des termes qui nous seront utiles pour la suite.

Fonctionnement de l'attaque.

Nous considérons notre groupe de L utilisateurs. Pour chaque utilisateur, nous construisons une chaîne de chiffrement $\mathcal{E}$ et une chaîne de déchiffrement $\mathcal{D}$ en commençant par deux points de départs aléatoires et distincts. Nous arrêtons la construction de la chaîne $\mathcal{E}$ une fois que $f^{\mathcal{E}}$ est arrivée à un point distingué. De même, pour la chaîne $\mathcal{D}$, nous arrêtons quand $f^{\mathcal{D}}$ arrive sur un point distingué.

Afin d'utiliser nos chaînes pour détecter des collisions, nous stockons, pour chaque chaîne construite, les triplets $(x_{\ell-1}, d, x_{\ell+1})$ où d est le point distingué trouvé ($x_\ell = d$), $x_{\ell-1}$ est le point de la chaîne avant qu'elle arrive à un point distingué et $x_{\ell+1}$ est le point suivant d'un point distingué. Ces triplets sont indispensables pour la suite. En effet, il faut stocker le point $x_{\ell-1}$ pour tester si la collision détectée est utilisable. Plus précisément, il faut s'assurer que les chaînes qui collisionnent ne sont pas de sous-chaînes les unes des autres et pour cela nous avons besoin de vérifier les points avant la

détection d'un point distingué. En outre, il faut vérifier que la collision détectée est une vraie collision. Pour cela nous stockons le point $x_{\ell+1}$ et nous vérifions que nous avons toujours une égalité.

Une fois nos chaînes construites, nous cherchons des collisions entre les points d'arrivée pour les différentes chaînes de chiffrement et de déchiffrement. Soit deux utilisateurs $U^{(1)}$ et $U^{(2)}$. Si leurs chaînes $\mathcal{E}_{K_0^{(1)},K_1^{(1)}}(x^{(1)})$ et $\mathcal{D}_{K_0^{'(2)},K_1^{(2)}\oplus\alpha}(y^{(2)})$ arrivent au même point distingué, nous suspectons que les deux chaînes sont devenues parallèles.

La fonctions interne Pcore est seulement paramétrée par la clé K_1. Cela veut dire que si $f^{\mathcal{E}_{K_1^{(1)}}}$ collisionne avec $f^{\mathcal{D}_{K_1^{(2)}\oplus\alpha}}$, nous obtenons, très probablement, une collision entre les clés $K_1^{(1)}$ et $K_1^{(2)} \oplus \alpha$. Cependant, nous devrons vérifier qu'il s'agit d'une vraie collision et non pas seulement d'un incident aléatoire. Pour cela, il faut continuer le calcul de $f^{\mathcal{E}}$ et $f^{\mathcal{D}}$ pour un point après un point distingué et vérifier que ces deux points continuent rester égaux. Si la collision obtenue est une vraie collision, nous savons que :

$$K_1^{(1)} = K_1^{(2)} \oplus \alpha.$$

En outre, le fait que les chaînes $\mathcal{E}_{K_0^{(1)},K_1^{(1)}}(x^{(1)})$ et $\mathcal{D}_{K_0^{'(2)},K_1^{(2)}\oplus\alpha}(y^{(2)})$ soient devenues parallèles nous indique que :

$$\mathcal{E}_{K_0^{(1)},K_1^{(1)}}(x^{(1)}) = \mathcal{D}_{K_0^{'(2)},K_1^{(2)}\oplus\alpha}(y^{(2)}) \oplus K_0^{(1)} \oplus K_0^{'(2)}.$$

Nous en déduisons donc que nous avons une paire $x^{(1)}$ et $y^{(2)}$ telle quelle :

$$x^{(1)} \oplus y^{(2)} = K_0^{(1)} \oplus K_0^{'(2)} \ \ (\text{ou} \ \ K_0^{(1)} \oplus K_0^{'(2)} \oplus \delta).$$

Il est évident que de $K_1^{(1)} = K_1^{(2)} \oplus \alpha$, il résulte que $K_1^{(1)} \oplus \alpha = K_1^{(2)}$. De la même collision, nous pouvons donc aussi récupérer :

$$K_0^{'(1)} \oplus K_0^{(2)} \ \ (\text{ou} \ \ K_0^{'(1)} \oplus K_0^{(2)} \oplus \delta).$$

De tout cela, nous déduisons que, si pour deux utilisateurs nous détectons que leurs chaînes, de types différents (chaîne de chiffrement pour l'un et chaîne de déchiffrement pour l'autre), sont parallèles, il en découle que :

$$K_0^{(1)} \oplus K_0^{'(2)} = A \ \text{ et } \ K_0^{'(1)} \oplus K_0^{(2)} = B \ \ (*).$$

Il suffit donc de résoudre ce système et récupérer $K_0^{(1)}$ et $K_0^{(2)}$. Soit $\{a_{63}, \ldots, a_0\}$ la représentation en bits de $K_0^{(1)}$ et $\{b_{63}, \ldots, b_0\}$ la représentation en bits de $K_0^{(2)}$. Nous

allons maintenant utiliser le lien entre les deux clés de blanchiment de PRINCE. De la définition de la clé $K_0^{'}$, nous avons que $K_0^{'} = (K_0 \ggg 1) \oplus (K_0 \gg 63)$. Nous pouvons donc récupérer la représentation binaire de $K_0^{'(1)}$ et $K_0^{'(2)}$:

$$K_0^{'(1)} = \{a_0, a_{63}, \ldots, a_2, a_1 \oplus a_{63}\} \text{ et } K_0^{'(2)} = \{b_0, b_{63}, \ldots, b_2, b_1 \oplus b_{63}\}.$$

En utilisant $(*)$, nous construisons le système :

$$\begin{aligned} \{a_{63}, \ldots, a_0\} \oplus \{b_0, b_{63}, \ldots, b_2, b_1 \oplus b_{63}\} &= \{A_{63}, \ldots, A_0\} \\ \{b_{63}, \ldots, b_0\} \oplus \{a_0, a_{63}, \ldots, a_2, a_1 \oplus a_{63}\} &= \{B_{63}, \ldots, B_0\} \end{aligned}$$

Comme celui-ci est un système linéaire inversible, nous pouvons le résoudre facilement et récupérer $K_0^{(1)}$ et $K_0^{(2)}$.

Pour conclure l'attaque, une fois que $K_0^{(i)}$ est détectée, il faut aussi récupérer la clé $K_1^{(i)}$. Étant donné que la clé $K_1^{(i)}$ fait seulement 64 bits, nous pouvons la récupérer en faisant une recherche exhaustive dont la complexité en temps est 2^{64}.

Les algorithmes ci-dessous décrivent en pseudocode le fonctionnement de cette attaque. Le client calcule les chaînes, stocke et envoie au serveur les données nécessaires pour la détection des collisions (algorithme 15). Le serveur, de son côté, récupère les données des clients, cherche entre eux des collisions potentielles et, dans le cas où il en détecte une, il calcule les valeurs de leurs clés (algorithme 16).

Utilisation simultanée des chaînes de chiffrement et de déchiffrement. La question que nous pouvons nous poser naturellement ici est pourquoi nous avons besoin de construire pour chaque utilisateur une chaîne de chiffrement $\mathcal{E}$ et une chaîne de déchiffrement $\mathcal{D}$ et donc ajouter le coût du calcul d'une chaîne supplémentaire. Effectivement, nous pouvons pour chaque utilisateur construire juste une chaîne de chiffrement[2]. Soit deux utilisateurs $U^{(1)}$ et $U^{(2)}$ pour lesquels nous détectons une collision entre $f^{\mathcal{E}^{(1)}}$ et $f^{\mathcal{E}^{(2)}}$. Si la collision obtenue est une vraie collision[3], nous en déduisons que $K_1^{(1)} = K_1^{(2)}$. En outre, le fait que les chaînes $\mathcal{E}_{K_0^{(1)}, K_1^{(1)}}$ et $\mathcal{E}_{K_0^{(2)}, K_1^{(2)}}$ sont parallèles, nous donne que :

$$K_0^{(1)} \oplus K_0^{(2)} = A \text{ et } K_0^{'(1)} \oplus K_0^{'(2)} = B.$$

Cependant, comme la clé $K_0^{'}$ est dérivée de la clé K_0 en utilisant une transformation linéaire, B sera forcement l'image de A par cette transformation et les deux équations seront identiques. Ce système, ne donnera donc pas de solution unique pour les clés K_0.

[2]Nous pouvons aussi ne construire que des chaînes de déchiffrement $\mathcal{D}$ et procéder de façon similaire.

[3]Nous vérifions qu'il s'agit d'une vraie collision comme précédemment, en vérifiant que les points suivants collisionnent aussi.

Algorithme 15 Attaque sur PRINCE dans un modèle multi-utilisateurs : client.

ENTRÉES :

- La fonction de chiffrement $\mathcal{E}$
- La fonction de déchiffrement $\mathcal{D}$
- La fonction $f^{\mathcal{E}}$
- La fonction $f^{\mathcal{D}}$
- L'ensemble de points distingués $\mathcal{S}_0$

Choisir aléatoirement deux points de départ x_0 et y_0 avec $x_0 \neq y_0$

{*Construire et stocker la chaîne de chiffrement.*}
pour $1 \leq \ell \leq 2^{32}$ **faire**
  $d^{\mathcal{E}} \leftarrow f^{\mathcal{E}}(x_{\ell-1})$
  $x_\ell \leftarrow x_{\ell-1} \oplus d^{\mathcal{E}}$
  si $d^{\mathcal{E}} \in \mathcal{S}_0$ **alors**
    $x_{\ell+1} \leftarrow f^{\mathcal{E}}(x_\ell)$
    Stocker $\{x_{\ell-1}, d^{\mathcal{E}}, x_{\ell+1}\}$
  fin si
fin pour

{*Construire et stocker la chaîne de déchiffrement.*}
pour $1 \leq k \leq 2^{32}$ **faire**
  $d^{\mathcal{D}} \leftarrow f^{\mathcal{D}}(y_{k-1})$
  $y_k \leftarrow y_{k-1} \oplus d^{\mathcal{D}}$
  si $d^{\mathcal{D}} \in \mathcal{S}_0$ **alors**
    $y_{k+1} \leftarrow f^{\mathcal{D}}(y_k)$
    Stocker $\{y_{k-1}, d^{\mathcal{D}}, y_{k+1}\}$
  fin si
fin pour

{*Envoyer les données au serveur.*}
Activer la connexion avec le serveur
Envoyer $\{x_{\ell-1}, d^{\mathcal{E}}, x_{\ell+1}\}$ et $\{y_{k-1}, d^{\mathcal{D}}, y_{k+1}\}$ au serveur
Terminer la connexion avec le serveur

Algorithme 16 Attaque sur PRINCE dans un modèle multi-utilisateurs : serveur.

ENTRÉES :
- Les données envoyées par les clients

SORTIES :
- Les clés K_0 des clients

{*Recevoir, stocker et trier les données des clients.*}
pour chaque $\{x_{\ell-1}, d^{\mathcal{E}}, x_{\ell+1}\}$ et $\{y_{k-1}, d^{\mathcal{D}}, y_{k+1}\}$ reçu **faire**
  Stocker dans un tableau $T^{\mathcal{E}}[\,][\,][\,]$ le triplet $\{x_{\ell-1}, d^{\mathcal{E}}, x_{\ell+1}\}$
  Stocker dans un tableau $T^{\mathcal{D}}[\,][\,][\,]$ le triplet $\{y_{k-1}, d^{\mathcal{D}}, y_{k+1}\}$
fin pour

Trier le tableau selon une relation d'ordre sur les points distingués

Regrouper les éléments correspondants au même point distingué en un sous-ensemble

{*Récupérer les clés.*}
pour chaque sous-ensemble **faire**
  si $T^{\mathcal{E}^{(i)}}[\,][\cdot][\,] == T^{\mathcal{D}^{(j)}}[\,][\cdot][\,]$ **alors**
    si $T^{\mathcal{E}^{(i)}}[\,][\,][\cdot] == T^{\mathcal{D}^{(j)}}[\,][\,][\cdot]$ **alors**
      Une collision est détectée
      si $T^{\mathcal{E}^{(i)}}[\cdot][\,][\,] == T^{\mathcal{D}^{(j)}}[\cdot][\,][\,]$ **alors**
        La collision n'est pas utilisable : détection de deux sous-chaînes
      fin si
    sinon
      La collision détectée n'est pas une vraie collision
    fin si
    $A \leftarrow T^{\mathcal{E}^{(i)}}[\cdot][\,][\,] \oplus T^{\mathcal{D}^{(j)}}[\cdot][\,][\,]$
    $B \leftarrow T^{\mathcal{D}^{(i)}}[\cdot][\,][\,] \oplus T^{\mathcal{E}^{(j)}}[\cdot][\,][\,]$
    De A et B déduire les clés K_0 des utilisateurs $U^{(i)}$ et $U^{(j)}$
  fin si
fin pour

Analyse et complexité de l'attaque.

Comme déjà expliqué, PRINCE utilise une clé K de 128 bits qui est séparée en deux parties de 64 bits *i.e.* $K = K_0 \| K_1$. Pour une paire d'utilisateurs i et j pour lesquels nous avons détecté que $K_1^{(i)} = K_1^{(j)} \oplus \alpha$, l'attaque consiste à identifier et récupérer tous les éléments de leurs clés. Par le paradoxe des anniversaires, nous avons qu'une telle collision peut apparaître, avec grande probabilité, entre deux utilisateurs d'un groupe d'au moins 2^{32} utilisateurs.

Pour que notre attaque soit possible, nous avons donc besoin d'utiliser un groupe de 2^{32} utilisateurs et pour chacun d'entre eux, nous construisons 2 chaînes : une chaîne de chiffrement et une chaîne de déchiffrement. La longueur de nos chaînes est égale à 2^{32} et donc le coût pour chaque utilisateur est égal à 2^{32} opérations. Il en découle que le coût total pour un groupe de 2^{32} dans le but de récupérer les clés de deux utilisateurs est de 2^{64} opérations. Ensuite, pour récupérer K_1, il faut appliquer une recherche exhaustive dont la complexité est 2^{64}. Finalement, le coût total pour récupérer les clés K_0 et K_1 de deux utilisateurs dans un groupe de 2^{32} utilisateurs est égal à 2^{65} opérations.

4.1.5 Attaque dans un modèle classique

La deuxième attaque effectuée sur PRINCE se place dans un modèle classique, *i.e.* nous attaquons un unique utilisateur. Nous montrons que, dans le cas de cet algorithme, la méthode des points distingués et les chaînes parallèles peuvent être utilisées en dehors du contexte multi-utilisateurs. Pour cette attaque, nous n'avons pas un groupe d'utilisateurs mais un seul utilisateur U pour lequel nous construisons des chaînes en utilisant la fonction de chiffrement $\mathcal{E}$ définie précédemment. Notre but ici est de construire plusieurs chaînes pour le même utilisateur et chercher des collisions entre elles.

La spécificité de notre attaque est une phase de pré-calcul. Pendant cette phase, nous construisons des chaînes d'une façon générique. Ces chaînes peuvent effectivement être utilisées pour attaquer un autre utilisateur. Néanmoins, cette phase de pré-calcul est importante car elle diminue la complexité de temps en-ligne de l'attaque.

Fonctionnement de l'attaque.

L'attaque présentée ici utilise les chaînes de chiffrement $\mathcal{E}$. Cependant, la même attaque peut être effectuée en utilisant les chaînes de déchiffrement $\mathcal{D}$.

Phase de pré-calcul. La première phase de notre attaque est une phase de pré-calcul. Pendant cette phase, nous souhaitons construire des chaînes pour chaque clé possible K_1^i avec $0 \leq i < 2^{64}$. Plus précisément, nous effectuons tout d'abord une recherche exhaustive sur K_1 pour examiner les 2^{64} clés possibles. Ensuite, pour chaque K_1^i, nous fixons $K_0^i = 0$ et nous construisons les différentes clés $K^i = K_0^i||K_1^i$. Finalement, pour chaque i, en utilisant la fonction de chiffrement $\mathcal{E}$, nous construisons une chaîne $\mathcal{S}_i$ de longueur 2^{32}. Pour chaque $\mathcal{S}_i$, nous stockons le point de départ de la chaîne, le point d'arrivée, c'est-à-dire le point distingué trouvé, et la longueur de la chaîne, c'est-à-dire le nombre d'itérations de la fonction $\mathcal{E}$.

Phase de l'attaque en ligne. Le but de cette deuxième phase de l'attaque est de trouver une collision entre une chaîne construite par notre utilisateur et une des chaînes construites en pré-calcul. Pour cela, nous commençons par un point de départ aléatoire x_0 et nous calculons une chaîne de chiffrement $\mathcal{R}$ pour notre utilisateur en utilisant la fonction $\mathcal{E}$. Notre but est de récupérer ses clés K_0 et K_1. La chaîne $\mathcal{R}$ construite colli-

sionne avec grande probabilité avec une des chaînes $\mathcal{S}_i$. Une fois qu'une collision entre les points distingués est détectée, nous savons que nos deux chaînes $\mathcal{S}_i$ et $\mathcal{R}$ sont parallèles. De ce fait, nous récupérons :

$$K_0^i \oplus K_0.$$

Comme la clé K_0^i est fixée à zéro, il est facile d'en déduire la clé K_0 de l'utilisateur. Grâce à la taille faible de la clé K_1, en faisant une recherche exhaustive nous arrivons à récupérer aussi K_1 facilement. Les algorithmes 17 et 18 décrivent en pseudocode la procédure de cette attaque.

Algorithme 17 Attaque sur PRINCE dans un modèle classique. - 1re partie

ENTRÉES :

- La fonction $f^{\mathcal{E}}$
- La fonction $\mathcal{E}$
- L'ensemble de points distingués $\mathcal{S}_0$

SORTIES :

- La clé K_0

{*Étape 1 : Examiner toutes les clés* $K^i = 0||K_1^i$}
Construire toutes les clés K_1^i avec $0 \leq i < 2^{64}$

{*Étape 2 : Construire et stocker les chaînes.*}
pour chaque clé K_1^i, avec $0 \leq i < 2^{64}$, calculée pendant la première étape **faire**
  Choisir aléatoirement un point de départ aléatoire x_0^i
  pour $1 \leq \ell \leq 2^{32}$ **faire**
    $dp^{\mathcal{E}} \leftarrow f^{\mathcal{E}}_{K_1^i}(x_{\ell-1}^i)$
    $x_\ell^i \leftarrow x_{\ell-1}^i \oplus dp^{\mathcal{E}}$
    si $dp^{\mathcal{E}} \in \mathcal{S}_0$ **alors**
      $x_{\ell+1}^i \leftarrow \mathcal{E}_{K_1^i}(x_\ell^i)$
      Stocker les i triplets $\{x_{\ell-1}^i, dp^{\mathcal{E}}, x_{\ell+1}^i\}$ dans un tableau $\mathcal{T}[\,][\,][\,]$
    fin si
  fin pour
fin pour

Algorithme 18 Attaque sur PRINCE dans un modèle classique. - 2e partie

{*Étape 3 : Construction de la chaîne de l'utilisateur.*}
Choisir aléatoirement un point de départ y_0
pour $1 \leq \ell \leq 2^{32}$ **faire**
  $d^{\mathcal{E}} \leftarrow f^{\mathcal{E}}(y_{\ell-1})$
  $y_\ell \leftarrow y_{\ell-1} \oplus d^{\mathcal{E}}$
  si $d^{\mathcal{E}} \in \mathcal{S}_0$ **alors**
    $y_{\ell+1} \leftarrow \mathcal{E}(y_\ell)$
    Stocker $\{y_{\ell-1}, d^{\mathcal{E}}, y_{\ell+1}\}$
  fin si
fin pour

{*Étape 4 : Chercher les collisions et récupérer la clé.*}
pour $0 \leq i \leq 2^{32}$ **faire**
  si $\mathcal{T}[\,][i][\,] == d^{\mathcal{E}}$ **alors**
    si $T[\,][\,][i] == y_{\ell+1}$ **alors**
      Une collision est détectée
      si $T[i][\,][\,] == y_{\ell-1}$ **alors**
        La collision n'est pas utilisable : détection de deux sous-chaînes
      fin si
    sinon
      La collision détectée n'est pas une vraie collision
    fin si
    $K_0 \leftarrow T^{\mathcal{E}}[i][\,][\,]$
  fin si
fin pour
retourne K_0

Analyse et complexité de l'attaque.

Pendant la phase du pré-calcul, nous calculons une chaîne de longueur 2^{32} pour chaque clé K_1. Nous avons 2^{64} clés possibles et donc la complexité de cette phase en temps est égale à 2^{96} opérations. En outre, chaque chaîne doit être stockée et donc le coût du pré-calcul en mémoire est égal à 2^{32}. Une fois la phase du pré-calcul terminée, l'adversaire doit calculer juste une chaîne pour récupérer la clé de l'utilisateur. Cette phase, coûte donc en temps 2^{32} opérations. De tout cela, il découle que le coût total de l'attaque est égal à 2^{96} opérations en temps et 2^{32} en mémoire.

Nous remarquons que cette attaque satisfait le compromis temps-mémoire $\mathcal{D} \times \mathcal{T} = 2^{32} \times 2^{96} = 2^{128}$. Cela veut dire que notre attaque n'améliore pas les attaques [BCG+12] et [JNP+13] et ne contredit pas la preuve de sécurité du papier d'origine de PRINCE. Néanmoins, grâce à la phase du pré-calcul, notre attaque propose un meilleur compromis mémoire-temps en-ligne $\mathcal{D} \times \mathcal{T}_{on}$ qui est égal à $\mathcal{D} \times \mathcal{T}_{on} = 2^{32} \times 2^{32} = 2^{64}$.

4.2 Applications sur l'algorithme de chiffrement par blocs DESX

Dans cette section nous montrons que les attaques vues précédemment sont assez génériques et donc elles peuvent s'adapter dans d'autres contextes. Plus précisément, l'attaque dans un modèle classique sur PRINCE peut aussi s'adapter au cas du DESX. Néanmoins, pour DESX, les deux clés de blanchiment sont indépendantes, ce qui n'est pas le cas de PRINCE. Notre attaque s'applique donc quand DESX suit le modèle Even-Mansour avec une seule clé, *i.e.* les deux clés de blanchiment K_1 et K_2 sont égales.

Pour conclure cette section, nous montrons comment appliquer l'attaque générique sur Even-Mansour dans le cas de DESX. Cependant, les résultats ne sont pas les mêmes car pour DESX nous ne pouvons pas construire de chaînes pour l'utilisateur public et, au contraire de PRINCE, les fonctions de chiffrement et de déchiffrement ne sont pas liées.

Préliminaires. Le fonctionnement de l'algorithme DESX a été présenté au chapitre 3. En effet, DESX suit le même modèle de construction qu'Even-Mansour :

$$DESX_{K||K_1||K_2}(m) = K_2 \oplus DES_K(K_1 \oplus m).$$

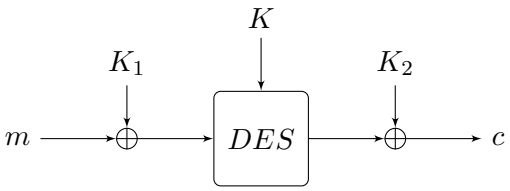

FIGURE 4.3 – Le schéma DESX.

Comme précédemment, nous avons besoin de définir les fonctions basées sur DESX qui seront utilisées pour la construction de chaînes. De façon similaire, nous définissons :

$$F_{K||K_1||K_2}(m) = m \oplus DESX_{K||K_1||K_2}(m) \oplus DESX_{K||K_1||K_2}(m \oplus \delta),$$

où m est un message clair et δ une constante arbitraire différente de zéro. La clé K_2 peut s'éliminer :

$$\begin{aligned} F_{K||K_1||K_2}(m) &= m \oplus DESX_{K||K_1||K_2}(m) \oplus DESX_{K||K_1||K_2}(m \oplus \delta) \\ &= m \oplus \cancel{K_2} \oplus DES_K(m \oplus K_1) \oplus \cancel{K_2} \oplus DES_K(m \oplus \delta \oplus K_1) \\ &= m \oplus DES_K(m \oplus K_1) \oplus DES_K(m \oplus \delta \oplus K_1). \end{aligned}$$

En outre, nous définissons la fonction f comme suit :

$$f_{K_1}(m) = DES_K(m \oplus K_1) \oplus DES_K(m \oplus \delta \oplus K_1).$$

4.2.1 Attaque sur DESX dans un modèle classique

L'attaque sur PRINCE dans un modèle classique peut également être appliquée sur DESX. Cependant, dans le cas de PRINCE la deuxième clé de blanchiment est déduite de la première ce qui n'est pas le cas de DESX, *i.e.* les clés K_1 et K_2 sont indépendantes. Nous montrons ici comment appliquer l'attaque dans le cas où $K_1 = K_2$:

$$DESX_{K||K_1}(m) = K^{'} \oplus DES_K(K_1 \oplus m).$$

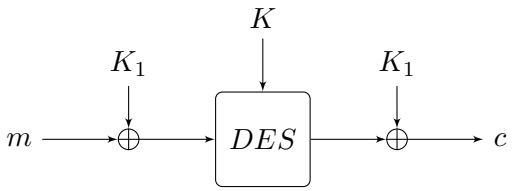

FIGURE 4.4 – Le schéma DESX avec $K_1 = K_2$.

Même si nous utilisons qu'une clé de blanchiment, la fonction utilisée pour la construction de chaînes sont définies de façon similaire :

$$F_{K||K_1}(m) = m \oplus DESX_{K||K_1}(m) \oplus DESX_{K||K_1}(m \oplus \delta),$$

où m est un message clair et δ une constante arbitraire différente de zéro.

$$\begin{aligned} F_{K||K_1}(m) &= m \oplus DESX_{K||K_1}(m) \oplus DESX_{K||K_1}(m \oplus \delta) \\ &= m \oplus \cancel{K_1} \oplus DES_K(m \oplus K_1) \oplus \cancel{K_1} \oplus DES_K(m \oplus \delta \oplus K_1) \\ &= m \oplus DES_K(m \oplus K_1) \oplus DES_K(m \oplus \delta \oplus K_1). \end{aligned}$$

Également, nous utilisons la fonction f :

$$f_{K_1}(m) = DES_K(m \oplus K_1) \oplus DES_K(m \oplus \delta \oplus K_1).$$

Fonctionnement de l'attaque.

L'attaque fonctionne de manière similaire à l'attaque sur PRINCE : une phase de pré-calcul suivie d'une phase d'attaque en ligne.

Phase de pré-calcul. Pendant la phase du pré-calcul, pour chaque i avec $0 \leq i < 2^{56}$ nous construisons les clés $K^i||K_1^i$ de la manière suivante : nous fixons $K_1^i = 0$ et nous faisons une recherche exhaustive sur K^i. Ensuite, pour chaque clé créée, nous construisons une chaîne $\mathcal{R}_i$ de longueur 2^{28} en commençant par un point de départ aléatoire et en utilisant la fonction $F_{K||0}$. Toutes les chaînes sont stockées.

Phase de l'attaque en ligne. Maintenant, pour l'utilisateur, nous commençons par un point de départ aléatoire et nous construisons une chaîne $\mathcal{S}$ en utilisant la fonction $F_{K||K_1}$ où $K||K_1$ sont les clés utilisées par l'utilisateur avec $K_1 = 0$. Le but est de trouver une collision entre les chaînes $\mathcal{R}_i$ construites précédemment et la chaîne $\mathcal{S}$. Ces chaînes collisionnent avec grande possibilité.

Une fois que la collision entre les point distingués de ces chaînes est détectée, nous en déduisons que les chaînes sont devenues parallèles et, donc, nous récupérons :

$$K_1^i \oplus K_1.$$

De cela, il est facile de récupérer la clé K_1 de l'utilisateur car $K_1^i = 0$. En outre, étant donné que la clé K est de petite taille, il est facile de la récupérer en faisant une recherche exhaustive.

Algorithme 19 Attaque sur DESX dans un modèle classique avec deux clés de blanchiment égales. - 1re partie

ENTRÉES :

- La fonction f
- La fonction F
- L'ensemble de points distingués $\mathcal{S}_0$

SORTIES :

- La clé K_1

{*Étape 1 : Récupérer toutes les clés* K^i}
Construire toutes les clés K^i avec $0 \leq i < 2^{56}$

{*Étape 2 : Construire et stocker les chaînes.*}
Choisir aléatoirement i points de départ aléatoires x_0^i
pour chaque clé K^i calculée pendant la première étape **faire**
  pour $1 \leq l \leq 2^{28}$ **faire**
    $dp \leftarrow f_{K^i}(x_{l-1})$
    $x_l \leftarrow x_{l-1} \oplus dp$
    si $dp \in \mathcal{S}_0$ **alors**
      $x_{\ell+1} \leftarrow F_{K^i||0}(x_l)$
      Stocker les i triplets $\{x_{l-1}, dp, x_{\ell+1}\}$ dans un tableau $\mathcal{T}[\,][\,][\,]$
    fin si
  fin pour
fin pour

Algorithme 20 Attaque sur DESX dans un modèle classique avec deux clés de blanchiment égales. - 2e partie

{*Étape 3 : Construction de la chaîne de l'utilisateur.*}
Choisir aléatoirement un point de départ y_0
pour $1 \leq l \leq 2^{28}$ **faire**
  $d \leftarrow f^(y_{l-1})$
  $y_l \leftarrow y_{l-1} \oplus d$
  si $d \in \mathcal{S}_0$ **alors**
    $y_{\ell+1} \leftarrow F(y_l)$
    Stocker $\{y_{l-1}, d, y_{\ell+1}\}$
  fin si
fin pour

{*Étape 4 : Chercher les collisions et récupérer la clé.*}
pour $0 \leq i \leq 2^{28}$ **faire**
  si $\mathcal{T}[\,][i][\,] == d$ **alors**
    si $T[\,][\,][i] == y_{\ell+1}$ **alors**
      Une collision est détectée
      si $T[i][\,][\,] == y_{\ell-1}$ **alors**
        La collision n'est pas utilisable : détection de deux sous-chaînes
      fin si
    sinon
      La collision détectée n'est pas une vraie collision
    fin si
    $K_1 \leftarrow T[i][\,][\,]$
  fin si
fin pour
retourne K_1

Analyse et complexité de l'attaque.

Pendant la phase de pré-calcul, nous calculons une chaîne de longueur 2^{28} pour chaque clé K^i. Nous avons donc 2^{56} possibilités et donc la complexité en temps hors-ligne de l'attaque est égale à 2^{84} opérations. De plus, le coût du pré-calcul en mémoire est 2^{28}.

Concernant la phase en-ligne, nous calculons juste une chaîne de taille 2^{28} et donc la complexité en temps en-ligne de l'attaque est égale à 2^{28}.

Pour conclure, le compromis temps-mémoire $\mathcal{D} \times \mathcal{T}$ de l'attaque sur DESX est :

$$\mathcal{D} \times \mathcal{T} = 2^{28} \times 2^{84} = 2^{112}.$$

Cependant, le compromis avec le temps en-ligne de l'attaque est beaucoup plus intéressant :

$$\mathcal{D} \times \mathcal{T} = 2^{28} \times 2^{28} = 2^{56}.$$

Cas avec deux clés distinctes de blanchiment. La fonction $F_{K||K_1}$ utilisée ici est en effet la même fonction $F_{K||K_1||K_2}$ présentée au début de la section. Un adversaire pourra donc procéder au même type d'attaque si $K_1 \neq K_2$ et récupérer la clé K_1. Pour récupérer la totalité des clés, il suffit de faire une recherche exhaustive sur la clé K qui coûte 2^{56} opérations en temps et une recherche exhaustive sur la clé K_2 qui coûte 2^{64} opérations en temps.

4.2.2 Attaque sur DESX dans un modèle multi-utilisateurs

Finalement, nous montrons comment l'attaque sur le schéma Even-Mansour dans un contexte multi-utilisateurs s'applique sur DESX. Ici aussi, comme cela été le cas pour l'attaque sur PRINCE, nous ne pouvons pas utiliser l'utilisateur public car la fonction interne est paramétrée par la clé de l'utilisateur.

Fonctionnement de l'attaque.

En effet, nous considérons un groupe de L utilisateurs et pour chaque utilisateur $U^{(i)}$, avec $0 \leq i < L$, nous construisons quelques chaînes en utilisant la fonction $F_{K^{(i)}||K_1^{(i)}||K_2^{(i)}}(x_0)$ pour un point de départ x_0 aléatoire. La construction de chaque chaîne s'arrête une fois que le $f_{K^{(i)}}$ correspondant arrive sur un point distingué.

Pour chaque chaîne construite qui arrive sur un point distingué, afin de pouvoir l'utiliser, nous stockons les triplets $(x_{\ell-1}, d, x_{\ell+1})$, où d est le point distingué trouvé, $x_{\ell-1}$ est le point de la chaîne avant qu'elle arrive sur un point distingué et $x_{\ell+1}$ est le point juste après le point distingué. Une fois les chaînes construites et stockées, nous cherchons les collisions entre les différents points distingués. Si une collision est détectée pour deux chaînes de deux utilisateurs $U^{(1)}$ et $U^{(2)}$ nous en déduisons que :

$$F_{K||K_1||K_2}(x) = F_{K||K_1||K_2}(y) \oplus K_1^{(1)} \oplus K_1^{(2)}.$$

La prochaine étape consiste donc à construire un graphe comme celui qui a été défini au chapitre 3 : les nœuds du graphe représentent les utilisateurs du groupe et les arêtes les collisions trouvées entre eux. Chaque arête correspond au XOR des premières clés de blanchiment des utilisateurs. Ce graphe deviendra connexe avec grande probabilité et nous arriverons donc à récupérer les clés d'une grande partie des utilisateurs.

Analyse de l'attaque.

En suivant l'analyse de l'attaque présentée pour Even-Mansour, pour le cas de DESX, nous avons besoin d'avoir un groupe de $N^{1/3}$ utilisateurs et faire $N^{1/3}$ requêtes par utilisateur. Par conséquence, pour appliquer l'attaque sur DESX, il nous faut un groupe de 2^{62} utilisateurs et faire 2^{62} requêtes pour récupérer les clés d'une grande partie des utilisateurs. Par contre, comme l'utilisateur public ne se trouve pas dans le graphe, nous ne sommes pas capables de récupérer la totalité des clés.

4.3 Le code d'authentification de message Chaskey

Nous avons vu jusqu'à maintenant comment effectuer des attaques sur les algorithmes de chiffrement par bloc en utilisant les méthodes de recherche de collisions. Dans cette section, nous adaptons nos techniques de recherche de collisions pour attaquer un autre type d'algorithme : les algorithmes de code d'authentification de message. Plus précisément, nous attaquons l'algorithme de code d'authentification de message Chaskey.

Nous montrons ici comment appliquer l'attaque de base sur Even-Mansour dans le cas de Chaskey mais, aussi, nous proposons des variantes. Ces variantes sont adaptées à la construction de Chaskey et offrent de meilleurs résultats. Plus précisément, nous proposons trois attaques contre Chaskey dont le principe est de construire des chaînes basées sur Chaskey et de chercher de collisions entre elles. Ces attaques ont été présentées à la conférence SAC 2015 (*Selected Areas in Cryptography*) [Mav15].

La première attaque fonctionne dans un modèle classique. Sa complexité est égale à 2^{64} opérations et donc, elle ne contredit pas la borne de sécurité du papier d'origine de Chaskey. Cependant, l'intérêt de cette attaque est de montrer comment appliquer la technique des chaînes parallèles pour détecter des collisions dans le cas de Chaskey. La deuxième attaque proposée sur Chaskey est une application de l'attaque de base sur Even-Mansour dans un modèle multi-utilisateurs. Avec cette attaque, nous récupérons les clés de tous les utilisateurs parmi un groupe de 2^{43} en faisant 2^{43} requêtes à la permutation publique et 2^{43} requêtes par utilisateur. Finalement, nous montrons que dans un contexte multi-utilisateurs, une attaque différente est également possible. En effet, nous proposons une attaque qui nous permet de récupérer les clés de deux utilisateurs dans un groupe de 2^{32} en ayant trouvé juste une collision entre eux.

Chaskey est un nouvel algorithme de code d'authentification de message (*Message Authentication Code - MAC*) proposé par Mouha *et al.* [MMH$^+$14b, MMH$^+$14a] à SAC 2014. Actuellement, il est étudié pour être normalisé par l'Organisation internationale de normalisation (*International Organization for Standardization - ISO*)[4] et par le Secteur de la normalisation des télécommunications de l'union internationale des télécommunications (*ITU Telecommunication Standardization Sector - ITU-T*)[5]. Il a été conçu pour être utilisé par des applications qui nécessitent une sécurité de 128 bits mais qui ne peuvent pas mettre en œuvre des algorithmes MAC standards à cause de contraintes sur la vitesse, la consommation d'énergie et la taille du code. Plus précisément, l'implémentation d'un MAC sur un microcontrôleur peut avoir des conséquences sur la vitesse du microcontrôleur à cause du coût de la fonction sous-jacente (fonction de hachage ou algorithme de chiffrement par blocs utilisé).

La construction de Chaskey est rapide et petite. Par exemple, implémentée sur un processeur ARM Cortex-M4, elle fonctionne à 7 cycles par octet et utilise seulement 402 octets de ROM. À l'inverse, sur le même processeur, une implémentation de AES-128-CMAC fonctionne à $89,4$ cycles par octet et une implémentation de AES-128-ECB né-

[4] http ://www.iso.org/iso/home.html
[5] http ://www.itu.int/en/ITU-T/Pages/default.aspx

cessite dix fois plus de place que l'implémentation de Chaskey. De plus, Chaskey peut être utilisé sur des microcontrôlleurs de $8, 16$ et 32 bits, n'est couvert d'aucun brevet et il n'y a aucune restriction sur l'utilisation de son code source.

Chaskey peut être décrit de deux façons différentes : soit comme une construction à base d'une permutation soit à partir de la construction CBC-MAC. Nous décrivons ci-dessous son fonctionnement et présentons les résultats existants sur sa sécurité.

4.3.1 Description du fonctionnement de Chaskey

Tout d'abord, Chaskey peut-être vu comme un MAC construit à partir d'une permutation. Il prend en entrée un message m de taille arbitraire et une clé K de 128 bits. Le message m est divisé en ℓ blocs $m_1, m_2, \ldots, m_\ell$ chacun de 128 bits. Si le dernier bloc n'est pas complet, un padding est appliqué à celui-ci. En sortie, il donne un tag τ de t bits où $t \leq n$ qui authentifie le message m.

Deux sous-clés K_1 et K_2 sont générées à partir la clé K de la façon suivante : K_1 est le résultat de la multiplication par 2 (notation binaire) de K et K_2 est égale au résultat de la multiplication par 2 de K_1 (voir algorithmes 21 et 22). D'une façon générale, nous définissons $K_1 = \alpha K$ et $K_2 = \alpha^2 K$. Pour définir la multiplication dans $GF(2^n)$, il faut spécifier le polynôme irréductible $f(x)$ de degré n qui définit la représentation du corps. Les concepteurs de Chaskey ont choisi le polynôme irréductible : $f(x) = x^{128} + x^7 + x^2 + x + 1$.

Algorithme 21 Algorithme de multiplication par 2 de Chaskey : TimesTwo(x)

ENTRÉES :
- Valeur x

si $x[127] = 0$ **alors**
  retourne $(x \gg 1) \oplus 0^{128}$
sinon
  retourne $(x \gg 1) \oplus 0^{120}10000111$
fin si

Algorithme 22 Algorithme de génération des sous-clés de Chaskey : SubKeys(K)

ENTRÉES :
- La clé K

SORTIES :
- Les sous-clés (K_1, K_2)

$K_1 \leftarrow$ TimesTwo(K)
$K_2 \leftarrow$ TimesTwo(K_1)
retourne (K_1, K_2)

Chaskey opère en $\ell + 3$ étapes où ℓ est le nombre de blocs du message :

- Étape 1 : Le premier bloc m_1 est XORé avec la clé K.
- Étape 2 : Le résultat de l'étape précédente passe dans la permutation π et la sortie est XORée au bloc suivant.
- Étape 3 à $\ell - 1$: L'étape 2 est répétée pour $\ell - 1$ fois pour que tous les blocs avant le dernier bloc m_ℓ soit traités.
- Étape ℓ : Si le dernier bloc m_ℓ est un bloc complet, aucun changement est effectué et l'étape précédente est répétée. Sinon, le padding $10^{n-|m_\ell|-1}$ est ajouté à m_ℓ et ensuite l'étape précédente est répétée.
- Étape $\ell + 1$: Si le dernier bloc m_ℓ est un bloc complet, la clé K_1 est XORée au résultat de l'étape ℓ. Si non, la clé K_2 est utilisée.
- Étape $\ell + 2$: Le résultat de l'étape ℓ passe dans la permutation π et les clés K_1 (pour m_ℓ complet) ou K_2 (pour m_ℓ non complet) sont XORées à la sortie.
- Étape $\ell+3$: Finalement, les t derniers bits sont sélectionnés et sont utilisés comme le tag τ.

L'algorithme 23 et la figure 4.5 décrivent toute la procédure.

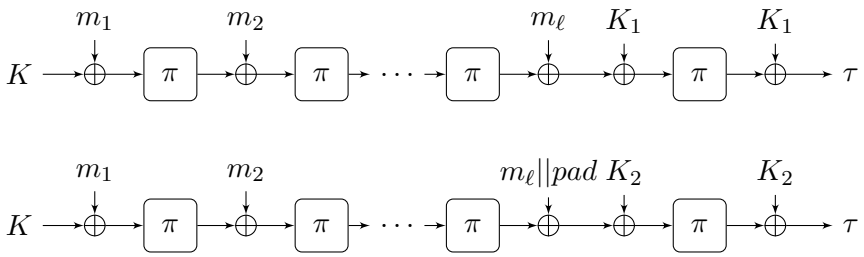

FIGURE 4.5 – L'algorithme MAC Chaskey. La première ligne montre la procédure de MAC quand $|m_\ell| = n$ et la deuxième ligne quand $0 \leq |m_\ell| < n$ où $pad = 10^{n-|m_\ell|-1}$.

Algorithme 23 Algorithme Chaskey

ENTRÉES :
- La clé K
- Le message m

SORTIES :
- Le tag τ

$(K_1, K_2) \leftarrow \text{SubKeys}(K)$
$m_1, \ldots, m_\ell \leftarrow m$
$h_1 = K$
pour tout i de 1 à $\ell - 1$ **faire**
  $h_{i+1} = \pi(h_i \oplus m_i)$
fin pour
si $|m_\ell| = n$ **alors**
  $L \leftarrow K_1$
sinon
  $m_\ell \leftarrow m_\ell \parallel 10^{n-|m_\ell|-1}$
  $L \leftarrow K_2$
fin si
$h_{\ell+1} = \pi(h_\ell \oplus m_\ell \oplus L) \oplus L$
retourne $\tau \leftarrow \text{droite}_t(h_{\ell+1})$

La permutation sous-jacente de Chaskey est construite sur le modèle ARX[6] et, plus précisément, elle est basée sur la permutation interne du MAC SipHash [AB12]. Cependant, pour Chaskey, un état de 128 bits (au lieu de 256 bits pour SipHash) est utilisé. Il est décomposé en quatre mots de 32 bits (au lieu de 64 bits pour SipHash). En outre, les constantes de rotation utilisées sont différentes. La fonction de la permutation interne de Chaskey est décrite à la figure 4.6.

La permutation de la version de base de Chaskey fait 8 tours de la fonction décrite à la figure 4.6. Les auteurs notent que les 8 tours sont suffisants pour avoir une construction sécurisée mais ils proposent que la variante Chaskey-LTS (*Long Term Security*) qui fait 16 tours soit aussi implémentée pour des raisons de sécurité. Cependant, ils soulignent que Chaskey-LTS utilise deux fois plus d'énergie et comprend deux fois plus de cycles que la version standard.

[6]La famille ARX (**A**ddition-**R**otation-**X**OR) contient les primitives qui utilisent seulement des opérations de XOR, d'addition et de rotation.

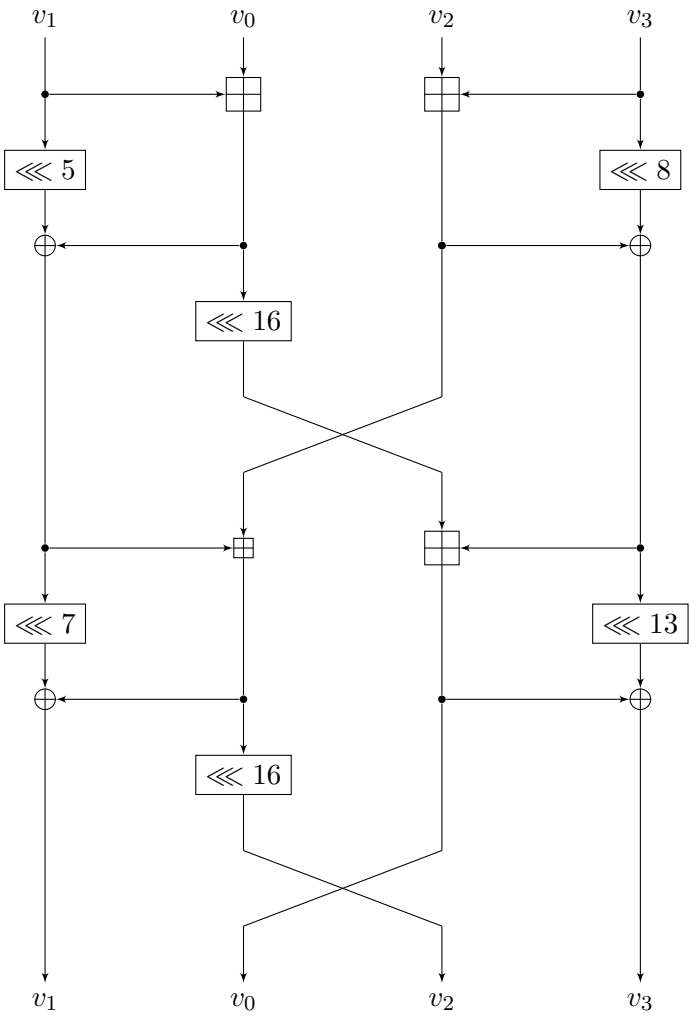

FIGURE 4.6 – La permutation interne du MAC Chaskey.

Il existe une description alternative de Chaskey, appelée Chaskey-B, où Chaskey est défini comme un algorithme MAC basé sur la construction CBC-MAC et où l'algorithme de chiffrement par blocs sous-jacent est une construction Even-Mansour. Par contre, comme CBC-MAC n'offre pas un bon niveau de sécurité quand il est utilisé pour des messages de taille arbitraire, la construction de Chaskey-B est basée sur des variantes simples de CBC-MAC proposées par Black et Rogaway [BR05].

Algorithme 24 Algorithme Chaskey-B

ENTRÉES :

- La clé K
- Le message m

SORTIES :

- Le tag τ

$(K_1, K_2) \leftarrow \text{SubKeys}(K)$
$m_1, \ldots, m_\ell \leftarrow m$
$h_1 = 0^n$
pour tout i de 1 à $\ell - 1$ **faire**
  $h_{i+1} = E_{K||K}(h_i \oplus m_i)$
fin pour
si $|m_\ell| = n$ **alors**
  $L \leftarrow K_1$
sinon
  $m_\ell \leftarrow m_\ell \parallel 10^{n-|m_\ell|-1}$
  $L \leftarrow K_2$
fin si
$h_{\ell+1} = E_{K \oplus L \| L}(h_\ell \oplus m_\ell)$
retourne $\tau \leftarrow \text{droite}_t(h_{\ell+1})$

Pour cette version de Chaskey, les auteurs définissent l'algorithme de chiffrement par blocs utilisé E comme $E_{X||Y}(m) = \pi(m \oplus X) \oplus Y$. Il suffit donc d'évaluer de façon récursive la fonction $h_{i+1} = E_{K||K}(h_i \oplus m_i)$ pour $i = 1, ..., \ell - 1$ et $h_1 = 0^n$. Si le dernier bloc est un bloc entier, nous calculons $h_\ell = E_{K \oplus K_1 || K_1}(h_\ell \oplus m_\ell)$. Si non, nous calculons $h_\ell = E_{K \oplus K_2 || K_2}(h_\ell \oplus m_\ell || 10^{n-|m_\ell|-1})$. Finalement, le tag τ est égal au t bits de poids faible. L'algorithme 24 et la figure 3.10 décrivent ce fonctionnement.

Cependant, dans ce cas, nous pouvons observer facilement que la clé K, qui est XORée à la sortie de chaque application de la permutation π, s'élimine pendant l'itération suivante (XOR de la même clé en entrée de la permutation π). Il en résulte que la clé K intervient seulement pendant la première itération (XOR de la clé K en entrée de la première application de la permutation π) et pendant la dernière itération sous la forme

des sous-clés K_1 ou K_2. Le tag τ de la sortie de Chaskey devient donc :

$$\tau = \pi(\pi(\pi(\pi(m_1 \oplus K) \oplus m_2) \oplus \ldots) \oplus m_l \oplus K_s) \oplus K_s$$

où K_s, avec $s \in \{1, 2\}$, représente une des deux sous-clés. Cela correspond à la description de Chaskey comme un MAC construit à partir d'une permutation comme il était décrit précédemment (figure 4.7).

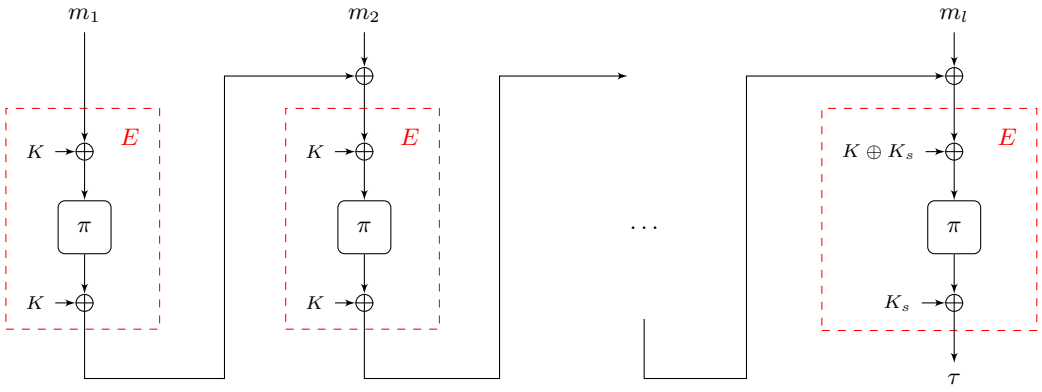

FIGURE 4.7 – Chaskey-B, la variante de Chaskey basée sur CBC-MAC où K_s avec $s \in \{1, 2\}$ est une des deux sous-clés.

En outre, il est facile d'observer que dans le cas où le message m utilisé contient un seul bloc, Chaskey devient un schéma Even-Mansour standard où la clé de blanchiment utilisée en entrée de la permutation est $K \oplus K_s$ avec $s \in \{1, 2\}$ et la clé de blanchiment utilisée en sortie est seulement la sous-clé K_s.

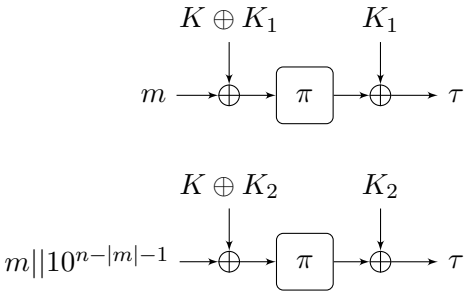

FIGURE 4.8 – Le fonctionnement de l'algorithme MAC Chaskey quand un seul bloc est utilisé.

4.3.2 Sécurité de Chaskey

Les auteurs de Chaskey prouvent la sécurité de leur schéma dans le modèle standard en se basant sur les résultats existants concernant le schéma Even-Mansour. Ils

en déduisent donc que les meilleures attaques génériques sur Chaskey ont besoin de $\mathcal{D} = 2^{n/2}$ messages clairs choisis pour une collision interne et $T = 2^n/\mathcal{D}$ requêtes à la permutation π ou π^{-1} pour faire une attaque de récupération de la clé où n est la taille du bloc.

Ils ont aussi analysé leur construction contre plusieurs types de cryptanalyse comme la cryptanalyse différentielle, la cryptanalyse différentielle tronquée, les attaques *meet-in-the-middle* et les attaques *slide* et ils en déduisent que Chaskey n'est vulnérable à aucune d'elles.

Chaskey est un MAC très récent et donc il existe actuellement peu d'attaques le concernant dans la littérature. Les premiers résultats sur Chaskey sont dûs à Bhaumik *et al.* [GJMN15]. Il s'agit de petites améliorations de la recherche exhaustive qui n'ont aucun impact pratique sur la sécurité de l'algorithme. Ensuite, Leurent [Leu15] a mis en évidence de nouvelles techniques basées sur la technique de partitionnement proposée par Biham *et al.* [BC14]. En utilisant ces techniques, il propose deux attaques de type différentielle-linéaire sur Chaskey, une sur 6 tours avec une complexité de $\mathcal{D} = 2^{25}$ et $\mathcal{T} = 2^{28.6}$ et une attaque sur 7 tours avec $\mathcal{D} = 2^{48}$ et $\mathcal{T} = 2^{67}$. Il en déduit que le schéma de base avec 8 tours de la permutation interne n'est pas suffisant pour garantir un bon niveau de sécurité.

4.3.3 Généralités et notations utilisées

Nous donnons ici les notations qui seront utilisées dans cette partie. Comme nous l'avons vu précédemment, nous cherchons à construire des chaînes et chercher des collisions entre elles. Pour construire nos chaînes, nous avons besoin de définir une fonction basée sur la fonction de Chaskey.

Cependant, la fonction de Chaskey change en fonction de la clé utilisée, *i.e.* l'utilisateur n'utilise pas la même clé si le bloc à chiffrer est complet et s'il n'en est pas. Nous définissons donc tout d'abord la fonction de Chaskey en fonction de la clé utilisée :

$$f_s(M) = K_s \oplus \pi(M \oplus (K_s \oplus K)).$$

Ensuite, nous définissons les fonctions qui seront utilisées pour la construction de nos chaînes :

$$F_{f_s}(M) = M \oplus f_s(M) \oplus f_s(M \oplus \delta)$$

$$g_{f_s}(M) = f_s(M) \oplus f_s(M \oplus \delta).$$

Pour toutes les fonctions ci-dessus K est la clé maitre de l'utilisateur, K_s avec $s \in \{1, 2\}$ représente les deux sous-clés générées comme expliqué ci-dessus, M est un message clair aléatoire et δ est une constante arbitraire non nulle.

De plus, nous définissons aussi deux fonctions pour l'utilisateur sans clé :

$$F_\pi(M) = M \oplus \pi(M) \oplus \pi(M \oplus \delta)$$

$$g_\pi = \pi(M) \oplus \pi(M \oplus \delta).$$

Nous remarquons que pour deux messages clairs m et $m^{'}$ qui forment une paire *slid*, *i.e.* $m \oplus m^{'} = K_s \oplus K$, nous avons que :

$$F_{f_s}(m^{'}) = F_\pi(m) \oplus (K_s \oplus K).$$

Nous en déduisons donc que deux chaînes basées sur les fonctions F_{f_s} et F_π définies précédemment, peuvent devenir parallèles et la différence constante entre ces deux chaînes serait égale au XOR des clés K_s et K.

En outre, nous obtenons des chaînes parallèles à chaque fois que nous détectons une collision entre les chaînes du même utilisateur ou d'utilisateurs différents. Nous distinguons trois cas possibles :

1. Une collision entre deux chaînes d'un seul utilisateur $U^{(i)}$:

   a) Collision entre deux chaînes du même type : collision entre deux chaînes basées sur F_{f_1} ou entre deux chaînes basées sur F_{f_2}. Dans ce cas, nous récupérons :
   $$K_1^{(i)} \oplus K = \alpha K \oplus K = (1+\alpha)K \text{ ou}$$
   $$K_2^{(i)} \oplus K = \alpha^2 K \oplus K = (1+\alpha^2)K.$$

   b) Collision entre deux chaînes de types différents : collision entre une chaîne basée sur F_{f_1} et une chaîne basée sur F_{f_2}. Dans ce cas, nous obtenons :
   $$K_1 \oplus \cancel{K} \oplus K_2 \oplus \cancel{K} = \alpha K \oplus \alpha^2 K = (\alpha + \alpha^2)K.$$

2. Une collision entre deux chaines du même type (F_{f_1} ou F_{f_2}) mais pour des utilisateurs différents $U^{(i)}$ et $U^{(j)}$.

   a) Collision $F_{f_1}^{(i)}(x) = F_{f_1}^{(j)}(y)$, pour deux points x et y. Dans ce cas, nous récupérons :
   $$K_1^{(i)} \oplus K^{(i)} \oplus K_1^{(j)} \oplus K^{(j)} = \alpha K^{(i)} \oplus K^{(i)} \oplus \alpha K^{(j)} \oplus K^{(j)} = (1+\alpha)(K^{(i)} \oplus K^{(j)}).$$

   b) Collision $F_{f_2}^{(i)}(x) = F_{f_2}^{(j)}(y)$, pour deux points x et y. Dans ce cas, nous obtenons :
   $$K_2^{(i)} \oplus K^{(i)} \oplus K_2^{(j)} \oplus K^{(j)} = \alpha^2 K^{(i)} \oplus K^{(i)} \oplus \alpha^2 K^{(j)} \oplus K^{(j)} = (1+\alpha^2)(K^{(i)} \oplus K^{(j)}).$$

3. *Collisions croisées* : Une collision entre deux types de chaînes différentes pour deux utilisateurs $U^{(i)}$ et $U^{(j)}$ distincts. Dans ce cas, pour deux points x et y, nous avons $F_{f_1}^{(i)}(x) = F_{f_2}^{(j)}(y)$ et nous obtenons donc :
   $$K_1^{(i)} \oplus K^{(i)} \oplus K_2^{(j)} \oplus K^{(j)} = \alpha K^{(i)} \oplus K^{(i)} \oplus \alpha^2 K^{(j)} \oplus K^{(j)} = (1+\alpha)K^{(i)} \oplus (1+\alpha^2)K^{(j)}.$$

Nous retrouverons tous ces types de collisions pendant l'explication du fonctionnement de nos attaques.

4.3.4 Attaque de récupération de clé dans un modèle classique

L'attaque que nous allons présenter maintenant est une attaque de récupération de la clé de Chaskey dans un modèle classique. La complexité de cette attaque est égale à 2^{64} opérations et donc, elle ne contredit pas la borne de sécurité de Chaskey. En outre, sa complexité est similaire d'autres attaques *slide* basés sur [BW00]. Néanmoins, cette attaque est intéressante car elle illustre une application de la technique de chaines parallèles sur Chaskey.

Pour appliquer cette attaque, il existe deux façons à procéder. Nous présentons ici les deux versions de l'attaque. Tout d'abord, nous pouvons construire des chaînes en utilisant un seul type. C'est-à-dire, utiliser des chaînes qui sont construites en utilisant F_{f_1} ou F_{f_2}. En outre, nous pouvons décrire une attaque qui cherche des collisions entre deux chaînes de type différent, *i.e.* des chaînes construites en utilisant F_{f_1} et F_{f_2}.

Fonctionnement de l'attaque.

Attaque basée sur un seul type de chaîne. La première étape de cette attaque consiste à créer pour l'utilisateur deux chaines. Il peut soit créer des chaînes en utilisant F_{f_1} et F_π soit en utilisant F_{f_2} et F_π. Pour créer ces chaînes, il faut commencer par des points de départs aléatoires, itérer les fonctions F_{f_1}, F_{f_2} et F_π plusieurs fois jusqu'au moment où un point distingué est détecté, *i.e.* jusqu'au moment où g_{f_1}, g_{f_2} et g_π arrivent sur un point distingué.

Ensuite, pendant la deuxième étape, nous stockons tous les points d'arrivée de nos chaînes, c'est-à-dire les points distingués, et nous cherchons des collisions entre eux, *i.e.* $F_{f_1}(x) = F_\pi(y)$ ou $F_{f_2}(z) = F_\pi(w)$.

Pour le premier cas, une collision $F_{f_1}(x) = F_\pi(y)$ nous donne :

$$x \oplus y = K_1 \oplus K = \alpha K \oplus K = (1 + \alpha)K.$$

Respectivement, une collision $F_{f_2}(z) = F_\pi(w)$ nous donne :

$$z \oplus w = (1 + \alpha^2)K.$$

Il est donc facile, dans les deux cas, de récupérer la clé K de l'utilisateur. L'algorithme de l'attaque pour le premier cas (utilisation de F_{f_1}) est détaillé dans l'algorithme 26.

Algorithme 25 Attaque sur Chaskey dans un modèle classique (un seul type de chaîne).

ENTRÉES :

- La constante α
- Les fonctions F_{f_1}, F_π, g_{f_1} et g_π
- L'ensemble de points distingués $\mathcal{S}_0$

SORTIES :

- La clé K

{*Étape 1 : Génération des chaînes*}
Choisir aléatoirement deux points de départ x_0 et y_0
pour $1 \leq \ell \leq 2^{64}$ **faire**
  $d_1 \leftarrow g_{f_1}(x_{\ell-1})$
  $d_2 \leftarrow g_\pi(y_{\ell-1})$
  $x_\ell \leftarrow x_{\ell-1} \oplus d_1$
  $y_\ell \leftarrow y_{\ell-1} \oplus d_2$
  si $d_1 \in \mathcal{S}_0$ **alors**
    $x_{\ell+1} \leftarrow F_{f_1}(x_\ell)$
    Stocker dans $T_1[\,][\,][\,]$ le triplet $\{x_{\ell-1}, d_1, x_{\ell+1}\}$
  fin si
  si $d_2 \in \mathcal{S}_0$ **alors**
    $y_{\ell+1} \leftarrow F_\pi(y_\ell)$
    Stocker dans $T_2[\,][\,][\,]$ le triplet $\{y_{\ell-1}, d_2, y_{\ell+1}\}$
  fin si
fin pour

{*Étape 2 : Recherche et vérification des collisions*}
pour $1 \leq i \leq 2^{64}$ **faire**
  pour $1 \leq j \leq 2^{64}$ **faire**
    si $T_1[\,][i][\,] == T_2[\,][j][\,]$ **alors**
      si $T_1[\,][\,][i] == T_2[\,][\,][j]$ **alors**
        Une collision est détectée
        si $T_1[i][\,][\,] == T_2[j][\,][\,]$ **alors**
          La collision n'est pas utilisable : détection de deux sous-chaînes
        fin si
      sinon
        La collision détectée n'est pas une vraie collision
      fin si
    fin si
  fin pour
fin pour

{*Étape 3 : Récupération de la clé K*}
retourne $K \leftarrow (T_1[i][\,][\,] \oplus T_2[j][\,][\,])/(1 \oplus \alpha)$

Attaque basée sur deux types de chaînes. Comme pour l'attaque précédente, la première étape consiste à créer pour l'utilisateur deux chaînes. Néanmoins, ici, nous construisons nos chaînes en utilisant les fonctions F_{f_1} et F_{f_2}.

Nous commençons par des points de départ aléatoires et nous itérons les fonctions F_{f_1} et F_{f_2}. Nous arrêtons la construction de nos chaînes quand g_{f_1} et g_{f_2} arrivent sur des points distingués.

Ensuite, nous stockons tous les points d'arrivée de nos chaînes, *i.e.* les point distingués trouvés. Une fois stockés, nous cherchons des collisions entre points distingués. Si une collision est trouvée, c'est-à-dire si pour deux points x et y nous avons $g_{f_1}(x) = g_{f_2}(y)$, nous en déduisons que :

$$x \oplus y = K_1 \oplus \not{K} \oplus K_2 \oplus \not{K} = \alpha K \oplus \alpha^2 K = (\alpha + \alpha^2)K.$$

De cela, nous déduisons facilement la clé K de l'utilisateur. L'algorithme de l'attaque est détaillé dans les algorithmes 26 et 27.

Algorithme 26 Attaque sur Chaskey dans un modèle classique (deux types de chaînes). - 1re partie

ENTRÉES :

- La constante α
- Les fonctions F_{f_1}, F_{f_2}, g_{f_1} et g_{f_2}
- L'ensemble de points distingués $\mathcal{S}_0$

SORTIES :

- La clé K

{Étape 1 : Génération des chaînes}
Choisir aléatoirement deux points de départ x_0 et y_0
pour $1 \leq \ell \leq 2^{64}$ **faire**
  $d_1 \leftarrow g_{f_1}(x_{\ell-1})$
  $d_2 \leftarrow g_{f_2}(y_{\ell-1})$
  $x_\ell \leftarrow x_{\ell-1} \oplus d_1$
  $y_\ell \leftarrow y_{\ell-1} \oplus d_2$
  si $d_1 \in \mathcal{S}_0$ **alors**
    $x_{\ell+1} \leftarrow F_{f_1}(x_\ell)$
    Stocker dans $T_1[\,][\,][\,]$ le triplet $\{x_{\ell-1}, d_1, x_{\ell+1}\}$
  fin si
  si $d_2 \in \mathcal{S}_0$ **alors**
    $y_{\ell+1} \leftarrow F_{f_2}(y_\ell)$
    Stocker dans $T_2[\,][\,][\,]$ le triplet $\{y_{\ell-1}, d_2, y_{\ell+1}\}$
  fin si
fin pour

Algorithme 27 Attaque sur Chaskey dans un modèle classique (deux types de chaînes). - 2e partie

{Étape 2 : Recherche et vérification des collisions}
pour $1 \leq i \leq 2^{64}$ **faire**
  pour $1 \leq j \leq 2^{64}$ **faire**
    si $T_1[\,][i][\,] == T_2[\,][j][\,]$ **alors**
      si $T_1[\,][\,][i] == T_2[\,][\,][j]$ **alors**
        Une collision est détectée
        si $T_1[i][\,][\,] == T_2[j][\,][\,]$ **alors**
          La collision n'est pas utilisable : détection de deux sous-chaînes
        fin si
      sinon
        La collision détectée n'est pas une vraie collision
      fin si
    fin si
  fin pour
fin pour

{Étape 3 : Récupération de la clé K}
retourne $K \leftarrow (T_1[i][\,][\,] \oplus T_2[j][\,][\,])/(\alpha \oplus \alpha^2)$

Analyse de la complexité de l'attaque.

Ici, il faut trouver une collision entre les deux ensembles des points distingués. Pour cela, il faut construire deux chaînes : une en utilisant F_{f_1} et l'autre en utilisant F_{f_2}. La longueur de chaque chaîne est 2^{64}. Donc, le coût total pour la construction de deux chaînes est 2^{64} opérations.

Comme expliqué au début de la section, cette attaque ne contredit pas la sécurité de Chaskey prouvée dans le papier d'origine. Le but de cette attaque est de montrer un exemple de l'utilisation de la technique des points distingués pour attaquer Chaskey dans un modèle classique avec un seul utilisateur.

4.3.5 Attaque de récupération de clé dans un modèle multi-utilisateurs

La deuxième attaque sur Chaskey se place dans un contexte multi-utilisateurs et il s'agit d'une application de l'attaque en multi-utilisateurs sur Even-Mansour. Nous présentons tout d'abord le fonctionnement de l'attaque, nous donnons sa complexité et, finalement, nous présentons une petite amélioration que nous pouvons faire sur l'attaque générique dans le cas de Chaskey.

Contexte de l'attaque. Ici, nous supposons que nous avons un groupe de L utilisateurs qui utilisent tous le MAC Chaskey. Chaque utilisateur $U^{(i)}$, avec $0 \leq i \leq L-1$,

choisit sa propre clé indépendamment des autres. À partir de la clé $K^{(i)}$, il génère les clés $K_s^{(i)}$ avec $s \in \{1, 2\}$. Les fonctions utilisées seront les fonctions f_s, F_{f_s}, f_π et F_π comme définies au début de la section.

Fonctionnement de l'attaque.

La première partie de l'attaque est la phase de la construction des chaînes. Pour chaque utilisateur $U^{(i)}$, avec $0 \leq i \leq L-1$, nous construisons un nombre fixe de chaînes F_{f_s}, en commençant par un point de départ aléatoire. Plus précisément, pour chaque $U^{(i)}$ nous itérons la fonction F_{f_1} jusqu'au moment où g_{f_1} arrive sur un point distingué. De façon similaire, nous itérons la fonction F_{f_2} jusqu'au moment où g_{f_2} arrive sur un point distingué.

Pendant la deuxième phase de l'attaque, nous construisons quelques chaînes pour l'utilisateur sans clé en itérant la fonction F_π jusqu'au moment où g_π arrive sur un point distingué.

Ensuite, nous cherchons les collisions entre les chaînes construites. Ici, nous pouvons avoir tous les types de collisions présentées à la section 4.3.3. En outre, nous pouvons avoir une collision entre un utilisateur $U^{(i)}$ et l'utilisateur sans clé. Une collision de ce type peut arriver en utilisant soit F_{f_1} soit F_{f_2}. Dans ce cas, nous récupérons :

$$K_1^{(i)} \oplus K^{(i)} = \alpha K^{(i)} \oplus K^{(i)} = (1+\alpha)K^{(i)} \text{ ou}$$

$$K_2^{(i)} \oplus K^{(i)} = \alpha^2 K^{(i)} \oplus K^{(i)} = (1+\alpha^2)K^{(i)}.$$

La quatrième étape de l'attaque consiste à construire le graphe. Les nœuds du graphe représentent les utilisateurs avec et sans clé. Cependant, pour cette attaque, étant donné que pour chaque utilisateur nous avons deux types de chaînes, chaque utilisateur est représenté par deux nœuds. À chaque fois qu'une collision est détectée entre deux utilisateurs, nous mettons une arête entre les nœuds correspondants. Cette arête est identifiée par les relations obtenues entre les clés comme indiqué précédemment. Pour mieux visualiser le graphe, nous présentons un exemple de graphe sur la figure 4.9.

Finalement, par les relations entre les différentes clés trouvées précédemment, nous pouvons résoudre le système et récupérer les clés des utilisateurs. Si nous détectons aussi une collision entre l'utilisateur public et un utilisateur avec clé, nous serons capables de trouver toutes les clés des utilisateurs qui appartiennent au graphe.

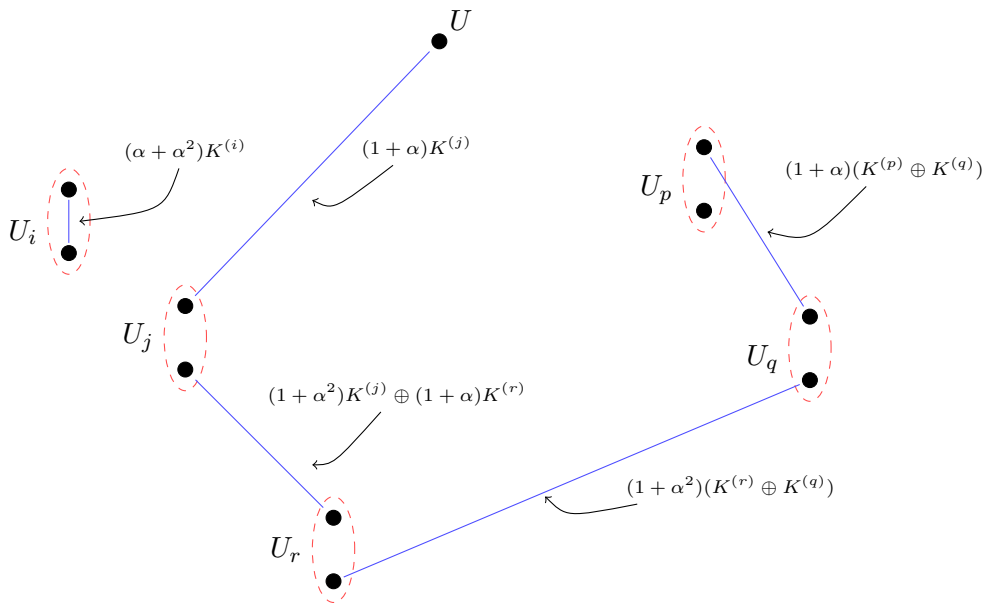

FIGURE 4.9 – Exemple d'un graphe pour Chaskey dans un contexte multi-utilisateurs.

Analyse de l'attaque.

L'évaluation du coût de cette attaque est basée sur l'analyse de la complexité de l'attaque de la section 3.3.2. Comme nous l'avons déjà montré, dans un groupe de $N^{1/3}$ utilisateurs, nous nous attendons à récupérer toutes les clés en faisant $c \cdot N^{1/3}$ requêtes par utilisateur (c est une petite constante arbitraire) et $N^{1/3}$ requêtes à l'utilisateur sans clé.

Dans le cas de Chaskey, nous avons que $N = 2^{128}$. Nous pouvons donc récupérer presque toutes les clés dans un groupe de 2^{43} utilisateurs. Pour cela, il suffit de faire 2^{43} requêtes par utilisateur et 2^{43} requêtes à l'utilisateur public.

4.3.6 Amélioration de l'attaque en multi-utilisateurs.

Maintenant, nous montrons qu'une amélioration de l'attaque précédente est possible dans le cas de Chaskey. En effet, nous montrons que dans le cas de Chaskey nous pouvons définir différemment les fonctions qui servent à la construction de nos chaînes et éliminer, de cette manière, l'utilisation du δ. Cette technique peut être utilisée pour chercher une collision entre deux chaînes du même type pour deux utilisateurs, *i.e.* $F_{f_1}^{(i)} = F_{f_1}^{(j)}$ ou $F_{f_2}^{(i)} = F_{f_2}^{(j)}$.

Dans un premier temps nous expliquons cette amélioration dans le cas où Chaskey possède un bloc complet et donc la clé K_1 est utilisée. Ensuite, nous montrons comment cette amélioration peut aussi s'appliquer dans le cas où nous utilisons des blocs incomplets.

Soit un message x, qui fait la taille entière d'un bloc, et la sortie correspondante y. En outre, soit $x^{'} = x \oplus K_1 \oplus K = x \oplus (1+\alpha)K$ et $y^{'} = \pi(x^{'}) = \alpha K \oplus y$ comme le montre la figure 4.10.

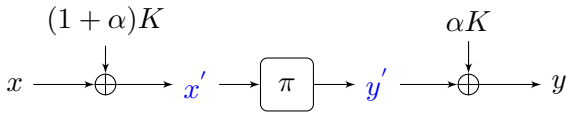

FIGURE 4.10 – L'algorithme MAC Chaskey quand des messages d'un bloc entier sont utilisés (utilisation de la sous-clé $K_1 = \alpha K$).

Nous observons que :

$$\begin{aligned}\alpha x \oplus (1+\alpha)y &= \alpha(x^{'} \oplus (1+\alpha)K) \oplus (1+\alpha)(y^{'} \oplus \alpha K) \\ &= \alpha x^{'} \oplus \cancel{\alpha(1+\alpha)K} \oplus (1+\alpha)y^{'} \oplus \cancel{\alpha(1+\alpha)K} \\ &= \alpha x^{'} \oplus (1+\alpha)y^{'}.\end{aligned}$$

Maintenant, notre but est de définir la fonction qui sera utilisée pour la construction de nos chaînes. Ici, nous définissons la fonction F_{f_1} comme suit :

$$F_{f_1}(x) = x \oplus \alpha x \oplus (1+\alpha)f_1(x) = (1+\alpha)(x \oplus f_1(x))$$

avec $f_1(x) = K_1 \oplus \pi(x \oplus (K_1 \oplus K))$.
Dans les attaques vues précédemment, nous utilisions une paire *slid* de messages clairs. Ici, nous supposons que nous avons deux messages clairs x_1 et x_2 qui satisfont :

$$x_1 \oplus x_2 = (1+\alpha)(K^{(i)} \oplus K^{(j)}).$$

Ainsi, pour deux utilisateurs $U^{(i)}$ et $U^{(j)}$, nous avons :

$$\begin{aligned}f_1^{(i)}(x_1) \oplus f_1^{(j)}(x_2) &= K_1^{(i)} \oplus \pi(x_1 \oplus (K_1^{(i)} \oplus K^{(i)})) \oplus K_1^{(j)} \oplus \pi(x_2 \oplus (K_1^{(j)} \oplus K^{(j)})) \\ &= K_1^{(i)} \oplus \pi(x_1 \oplus (1+\alpha)K^{(i)}) \oplus K_1^{(j)} \oplus \pi(x_2 \oplus (1+\alpha)K^{(j)}) \\ &= \alpha K^{(i)} \oplus \pi(x_2 \oplus \cancel{(1+\alpha)K^{(i)}} \oplus (1+\alpha)K^{(j)} \oplus \cancel{(1+\alpha)K^{(i)}}) \\ &\oplus \alpha K^{(j)} \oplus \pi(x_2 \oplus (1+\alpha)K^{(j)}) \\ &= \alpha(K^{(i)} \oplus K^{(j)})\end{aligned}$$

De cela :

$$\begin{aligned}
f_1^{(i)}(x_1) \oplus f_1^{(j)}(x_2) &= \alpha(K^{(i)} \oplus K^{(j)}) \\
x_1 \oplus x_2 \oplus f_1^{(i)}(x_1) \oplus f_1^{(j)}(x_2) &= x_1 \oplus x_2 \oplus \alpha(K^{(i)} \oplus K^{(j)}) \\
(x_1 \oplus f_1^{(i)}(x_1)) \oplus (x_2 \oplus f_1^{(j)}(x_2)) &= (1+\alpha)(K^{(i)} \oplus K^{(j)}) \oplus \alpha(K^{(i)} \oplus K^{(j)}) \\
(x_1 \oplus f_1^{(i)}(x_1)) \oplus (x_2 \oplus f_1^{(j)}(x_2)) &= K^{(i)} \oplus K^{(j)} \\
(1+\alpha)(x_1 \oplus f_1^{(i)}(x_1)) \oplus (1+\alpha)(x_2 \oplus f_1^{(j)}(x_2)) &= (1+\alpha)(K^{(i)} \oplus K^{(j)}) \\
F_{f_1}^{(i)} \oplus F_{f_1}^{(j)} &= (1+\alpha)(K^{(i)} \oplus K^{(j)}).
\end{aligned}$$

Nous en déduisons que deux chaînes construites en utilisant F_{f_1} peuvent devenir parallèles avec une différence constante égale à $(1+\alpha)(K^{(i)} \oplus K^{(j)})$. En utilisant donc la fonction $F_{f_1}(x) = (1+\alpha)(x \oplus f_1(x))$, nous pouvons procéder au même type d'attaque que précédemment.

De façon équivalente nous pouvons redéfinir la fonction F_{f_2}. En effet, pour un message clair x et sa sortie y après application de l'algorithme Chaskey, nous définissons $x^{'} = x \oplus (1+\alpha^2)K$ et $y^{'} = \pi(x) = \alpha^2 K \oplus y$.

Nous observons que :

$$\begin{aligned}
\alpha^2 x \oplus (1+\alpha^2)y &= \alpha^2(x^{'} \oplus (1+\alpha^2)K) \oplus (1+\alpha^2)(y^{'} \oplus \alpha^2 K) \\
&= \alpha^2 x^{'} \oplus \cancel{\alpha^2(1+\alpha^2)K} \oplus (1+\alpha^2)y^{'} \oplus \cancel{\alpha^2(1+\alpha^2)K} \\
&= \alpha^2 x^{'} \oplus (1+\alpha^2)y^{'}.
\end{aligned}$$

Nous définissons donc :

$$F_{f_2}(x) = x \oplus \alpha^2 x \oplus (1+\alpha^2)f_2(x) = (1+\alpha^2)(x \oplus f_2(x))$$

avec $f_2(x) = K_2 \oplus \pi(x \oplus (K_2 \oplus K))$.

Nous supposons aussi que nous avons deux messages clairs x_1 et x_2 qui satisfont :

$$x_1 \oplus x_2 = (1+\alpha^2)(K^{(i)} \oplus K^{(j)}).$$

De manière équivalente, ici, pour deux utilisateurs $U^{(i)}$ et $U^{(j)}$, nous avons :

$$f_2^{(i)}(x_1) \oplus f_2^{(j)}(x_2) = \alpha^2(K^{(i)} \oplus K^{(j)}).$$

Nous en déduisons donc que :

$$F_{f_2}^{(i)} \oplus F_{f_2}^{(i)} = (1+\alpha^2)(K^{(i)} \oplus K^{(j)}).$$

Il en découle que les chaînes construites en utilisant la fonction F_{f_2}, comme définie ci-dessus, peuvent aussi devenir parallèles avec une différence constante égale à $(1+\alpha^2)(K^{(i)} \oplus K^{(j)})$.

Nous remarquons que si nous construisons nos chaînes de cette façon, nous n'utiliserons pas la constante δ. Nous en déduisons donc que nous sommes en mesure d'avoir une petite amélioration et gagner un facteur $\sqrt{2}$ sur le calcul de nos chaînes.

4.3.7 Variante de l'attaque en multi-utilisateurs en utilisant des collisions croisées

Nous avons vu jusqu'à maintenant comment appliquer la méthode de recherche de chaînes parallèles dans le cas de Chaskey. L'attaque présentée ici est une variante de l'attaque précédente. Elle se situe dans un contexte multi-utilisateurs et utilise également les chaînes parallèles. Cependant, son application est spécifique au cas de Chaskey. Cette attaque nous permet de récupérer les clés de deux utilisateurs après avoir détecté une seule collision entre eux. Néanmoins, le type de collision qui nous permet de procéder à ce type d'attaque est seulement celui des collisions croisées entre les utilisateurs.

Pour cette attaque, les fonctions utilisées sont celles présentées au début de la section 4.3.3. Nous nous plaçons toujours dans un modèle multi-utilisateurs et nous considérons donc que nous avons L utilisateurs qui utilisent tous l'algorithme Chaskey.

Fonctionnement de l'attaque.

La première étape de l'attaque consiste à construire les chaînes qui seront utilisées par la suite. Pour deux utilisateurs $U^{(i)}$ et $U^{(j)}$ de notre groupe de L utilisateurs, nous construisons un nombre constant de chaînes en commençant par des points aléatoires et utilisant la fonction F_{f_s}, où $s \in \{1,2\}$. Pour chaque utilisateur, nous avons besoin de construire quelques chaînes basées sur F_{f_1} et quelques chaînes basées sur F_{f_2}. La construction des chaînes s'arrête quand g_{f_1} et g_{f_2} arrivent sur un point distingué.

Ensuite, nous stockons les chaînes construites et nous cherchons entre elles une *collision croisée*. Comme expliqué à la section 4.3.3, par une *collision croisée* $F_{f_1}^{(i)}(x) = F_{f_2}^{(j)}(y)$, pour deux points x, y entre deux utilisateurs $U^{(i)}$ et $U^{(j)}$ nous obtenons :

$$x \oplus y = (1+\alpha)K^{(i)} \oplus (1+\alpha^2)K^{(j)}.$$

En outre, soit c et $c^{'}$ la sortie correspondante aux entrées x et y :

$$c = K_1^{(i)} \oplus \pi(x \oplus K_1^{(i)} \oplus K^{(i)})$$

$$c^{'} = K_2^{(j)} \oplus \pi(y \oplus K_2^{(j)} \oplus K^{(j)}).$$

Donc, si x et y forment une paire *slid*, nous pouvons aussi déduire que :

$$c \oplus c^{'} = K_1^{(i)} \oplus K_2^{(j)} = \alpha K^{(i)} \oplus \alpha^2 K^{(j)}.$$

Nous récupérons donc le système linéaire :

$$\begin{aligned}(1+\alpha)K^{(i)} \oplus (1+\alpha^2)K^{(j)} &= \Delta_1 \\ \alpha K^{(i)} \oplus \alpha^2 K^{(j)} &= \Delta_2.\end{aligned}$$

Ce système est inversible car :

$$\begin{vmatrix}(1+\alpha) & (1+\alpha^2) \\ \alpha & \alpha^2\end{vmatrix} = \alpha \neq 0.$$

Il existe donc une seule solution pour ce système linéaire. Nous en déduisons donc les clés $K^{(i)}$ et $K^{(j)}$ des utilisateurs $U^{(i)}$ et $U^{(j)}$. Les algorithmes 28 et 29 décrivent en détail le fonctionnement de cette attaque.

Analyse de la complexité de l'attaque.

Nous souhaitons calculer maintenant la complexité de cette attaque. Ici, nous cherchons une *collision croisée* entre deux utilisateurs distincts. Pour cela, il faut avoir un groupe de $\sqrt[4]{N}$ et faire $\sqrt[4]{N}$ requêtes par utilisateur. Dans le cas de Chaskey, nous avons $N = 2^{128}$ et donc nous avons besoin d'avoir un groupe de 2^{32} utilisateurs et faire 2^{32} requêtes par utilisateur.

En pratique, cette dernière attaque utilise des techniques similaires à celles des attaques sur Chaskey présentées précédemment. Mais avec cette attaque nous obtenons de meilleurs résultats. En fait, l'innovation ici réside dans le fait que l'attaquant est en mesure de récupérer les clés de deux utilisateurs en détectant une seule collision entre eux. En outre, l'attaquant a besoin d'avoir un groupe d'utilisateurs plus petit que pour les autres attaques. Cependant, si l'attaquant souhaite récupérer les clés de tous les utilisateurs, il lui faut un groupe plus grand, et plus précisément, un groupe de 2^{43} utilisateurs comme précédemment.

Algorithme 28 Attaque sur Chaskey dans un modèle multi-utilisateurs en cherchant des collisions croisées : client.

ENTRÉES :

- Les fonctions F_{f_1} et F_{f_2}
- L'ensemble de points distingués $\mathcal{S}_0$

SORTIES :

- La clé K

{*Étape 1 : Génération des chaînes*}
Choisir aléatoirement deux points de départ x_0 et y_0
pour $1 \leq \ell \leq 2^{64}$ **faire**
  $d_1 \leftarrow g_{f_1}(x_{\ell-1})$
  $d_2 \leftarrow g_{f_2}(y_{\ell-1})$
  $x_\ell \leftarrow x_{\ell-1} \oplus d_1$
  $y_\ell \leftarrow y_{\ell-1} \oplus d_2$
  si $d_1 \in \mathcal{S}_0$ **alors**
    $x_{\ell+1} \leftarrow F_{f_1}(x_\ell)$
    Stocker dans $T_1[\,][\,][\,]$ le triplet $\{x_{\ell-1}, d_1, x_{\ell+1}\}$
  fin si
  si $d_2 \in \mathcal{S}_0$ **alors**
    $y_{\ell+1} \leftarrow F_{f_2}(y_\ell)$
    Stocker dans $T_2[\,][\,][\,]$ le triplet $\{y_{\ell-1}, d_2, y_{\ell+1}\}$
  fin si
fin pour

{*Étape 2 : Envoi des chaînes au serveur*}
si $T_1[\,][\,][\,]$ ou $T_2[\,][\,][\,]$ non vides **alors**
  Activer la connexion avec le serveur
  Envoyer $T_1[\,][\,][\,]$ si non vide et $T_2[\,][\,][\,]$ si non vide au serveur
  Terminer la connexion avec le serveur
fin si

Algorithme 29 Attaque sur Chaskey dans un modèle multi-utilisateurs en cherchant des collisions croisées : serveur.

ENTRÉES :
- Les données récupérées par les clients

SORTIES :
- La clé K de deux utilisateurs

{*Étape 1 : Recherche et vérification des collisions*}
pour Utilisateurs A et B distincts **faire**
  pour $1 \leq i \leq 2^{64}$ **faire**
    pour $1 \leq j \leq 2^{64}$ **faire**
      si $T_1^{(A)}[\,][i][\,] == T_2^{(B)}[\,][j][\,]$ **alors**
        si $T_1^{(A)}[\,][\,][i] == T_2^{(B)}[\,][\,][j]$ **alors**
          Une collision est détectée
          si $T_1^{(A)}[i][\,][\,] == T_2^{(B)}[j][\,][\,]$ **alors**
            La collision n'est pas utilisable : détection de deux sous-chaînes
          fin si
        sinon
          La collision détectée n'est pas une vraie collision
        fin si
      fin si
    fin pour
  fin pour
fin pour

{*Étape 2 : Récupération de la clé K*}
$\Delta_1 \leftarrow T_1^{(A)}[i][\,][\,] \oplus T_2^{(B)}[j][\,][\,])$
$\Delta_2 \leftarrow$ le XOR des sorties correspondantes
Résoudre le système linéaire
retourne $K^{(A)}$ et $K^{(B)}$

4.4 Amélioration de l'attaque générique pour PRINCE et Chaskey

Dans ce chapitre, nous avons montré que la nouvelle attaque générique sur Even-Mansour peut avoir des applications à diverses primitives symétriques. Plus précisément, en utilisant les techniques de l'attaque générique, nous proposons des attaques sur l'algorithme *lightweight* de chiffrement par blocs PRINCE et le code d'authentification de message Chaskey. Dans les deux cas, nous proposons des compromis différents que l'attaque générique sur Even-Mansour. Pour conclure ce chapitre, nous résumons ici, les caractéristiques de ces algorithmes qui nous permettent de proposer des attaques

ayant des meilleurs résultats que l'attaque générique.

Comme nous l'avons déjà expliqué, l'algorithme PRINCE suit la construction du schéma Even-Mansour. Néanmoins, la différence primordiale entre les deux constructions est que PRINCE utilise la fonction Pcore paramétrée par la clé K_1 de l'utilisateur au lieu de la permutation publique du schéma Even-Mansour. En outre, grâce à la propriété de α-*reflection*, pour PRINCE nous avons la possibilité de construire deux chaînes pour chaque utilisateur : une chaîne de chiffrement $\mathcal{E}$ et une chaîne de déchiffrement $\mathcal{D}$.

En ce qui concerne notre attaque sur PRINCE dans un modèle multi-utilisateurs, l'idée de base reste la même que l'attaque sur Even-Mansour : nous construisons des chaînes et nous attendons qu'elles deviennent parallèles. Cependant, pour l'attaque sur PRINCE, nous n'avons pas besoin de construire un graphe pour récupérer les clés des utilisateurs. En effet, dès qu'une collision entre deux utilisateurs est détectée, nous récupérons leurs clés. Cette amélioration est due au cadencement des clés de l'algorithme et à la propriété de α-*reflection*. Comme nous l'avons vu lors de la présentation du fonctionnement de l'attaque, grâce à la propriété de α-*reflection*, à partir d'une collision entre une chaîne de chiffrement pour l'utilisateur $U^{(1)}$ et une chaîne de déchiffrement de l'utilisateur $U^{(2)}$, nous récupérons :

$$K_0^{(1)} \oplus K_0^{'(2)} = A \text{ et } K_0^{'(1)} \oplus K_0^{(2)} = B \ (*).$$

La relation entre les deux clés de blanchiment K_0 et $K_0^{'}$:

$$K_0^{'} = (K_0 \ggg 1) \oplus (K_0 \gg 63),$$

nous permet de résoudre le système linéaire inversible $(*)$ et récupérer les valeurs des clés de blanchiment. Pour résumer, ces deux spécificités de PRINCE, la propriété de α-*reflection* et la relation entre les deux clés de blanchiment, ne rendent pas seulement notre attaque possible mais elles nous permettent de récupérer les clés avec une seule collision.

Également, dans le cas de Chaskey, nous présentons une attaque dans un modèle multi-utilisateurs proposant un meilleur compromis que l'attaque sur Even-Mansour. Comme nous l'avons vu lors de la description de son fonctionnement, la construction de Chaskey suit le schéma de Even-Mansour itéré. Néanmoins, quand le message traité contient un seul bloc, Chaskey devient un schéma Even-Mansour standard. Ici aussi, comme pour PRINCE, nous pouvons construire deux types de chaînes pour chaque utilisateur. En effet, deux sous-clés K_1 et K_2 sont générées à partir de la clé K. Quand le dernier bloc du message à traiter est un bloc entier, la clé K_1 est utilisée. Dans le cas contraire, la clé K_2 est utilisée. Cela nous permet de construire deux chaînes pour chaque utilisateur en modifiant notre message pour que les deux clés soient utilisées.

Cependant, cette spécificité de Chaskey, nous permet d'avoir des meilleurs résultats. Plus précisément, nous avons présenté une variante de l'attaque en multi-utilisateurs qui nécessite juste une collision *croisée* entre deux utilisateurs pour récupérer leurs clés. En effet, quand une telle collision est détectée, nous récupérons un système linéaire

inversible qui nous donne les clés des utilisateurs. Grâce à cela, nous récupérons les clés dès qu'une collision *croisée* arrive et donc sans avoir à utiliser l'idée de la construction du graphe.

Pour conclure ce chapitre, nous remarquons, que dans les deux cas de PRINCE et de Chaskey, ce qui améliore nos attaques par rapport à l'attaque sur le schéma Even-Mansour est la génération des clés de blanchiment et, plus précisément, la relation entre eux.

5. Conclusion

Dans cette thèse, nous nous sommes intéressés aux algorithmes de type Even-Mansour. Plus précisément, nous avons étudié le schéma Even-Mansour et nous avons proposé des attaques sur le schéma lui-même mais aussi sur des primitives dont la structure lui ressemble.

Nous avons introduit un nouveau type d'attaque sur le schéma Even-Mansour. Ce type d'attaque utilise deux nouvelles idées algorithmiques. La première idée modifie l'algorithme de recherche parallélisable des collisions de van Oorschot et Wiener. Dans leur algorithme, van Oorschot et Wiener utilisent des chaînes calculées en itérant une fonction aléatoire. Pour détecter les collisions, ils utilisent la méthode des points distingués pour détecter les chaînes qui fusionnent. Nous appliquons une méthode similaire sur le schéma Even-Mansour mais, dans notre cas, nous n'attendons pas que les chaînes fusionnent mais qu'elles deviennent *parallèles*. En utilisant cette nouvelle idée, nous proposons une attaque générique sur le schéma Even-Mansour qui fait $\mathcal{D} = N^{2/5}$ requêtes à la permutation avec clé, $\mathcal{T} = N^{3/5}$ requêtes à la permutation publique et utilise $\mathcal{M} = N^{1/3}$ de mémoire. La deuxième idée est utilisée pour appliquer l'attaque générique dans un modèle multi-utilisateurs. Dans ce cas, après avoir détecté les chaînes parallèles entre les différents utilisateurs du groupe, nous construisons un graphe qui illustre les relations obtenues entre les clés. Nous prouvons qu'une composante géante apparaît avec grande probabilité et que donc nous sommes capables de récupérer la majorité des clés de notre groupe.

Inspirés de cette attaque, nous avons proposé des attaques sur les algorithmes de chiffrement par blocs PRINCE et DESX. Pour chaque algorithme, nous avons proposé à la fois des attaques dans un modèle classique et dans un modèle multi-utilisateurs. Dans le cas de PRINCE, en exploitant des spécificités liés à la génération des clés et la propriété de α - *reflection* de l'algorithme, nous proposons une attaque dans un modèle multi-utilisateurs qui nous permet de récupérer les clés de deux utilisateurs dans un groupe de 2^{32} utilisateurs en faisant 2^{65} opérations. En outre, nous proposons une attaque dans le modèle classique. Cette attaque a une phase de pré-calcul qui coûte en mémoire 2^{32} opérations et 2^{96} opérations en temps. En revanche, la phase en ligne coûte en temps seulement 2^{32} opérations. Nous appliquons des attaques similaires sur l'algo-

rithme de chiffrement par blocs DESX. L'attaque proposée dans un modèle classique s'applique dans le cas où les deux clés de blanchiment de l'algorithme sont égales. La phase de pré-calcul coûte 2^{28} en mémoire et 2^{84} opérations en temps et la phase en-ligne coûte 2^{28} opérations en temps. L'attaque dans un modèle multi-utilisateurs utilise un groupe de 2^{62} utilisateurs et, en faisant 2^{62} requêtes nous arrivons à récupérer les clés d'une grande partie d'utilisateurs.

Finalement, nous adaptons nos techniques pour proposer des attaques de récupération de clé sur l'algorithme MAC Chaskey. Nous proposons deux attaques dans un modèle classique qui ne contredisent pas la sécurité de Chaskey prouvée dans le papier d'origine. En outre, nous montrons que l'attaque générique sur le schéma Even-Mansour dans un modèle multi-utilisateurs peut s'appliquer sur l'algorithme Chaskey. Avec cette attaque, nous arrivons à obtenir presque toutes les clés d'un groupe de 2^{43} utilisateurs en faisant 2^{43} requêtes par utilisateur et 2^{43} requêtes à l'utilisateur public. Finalement, nous présentons une variante de cette attaque qui est spécifique à la construction de Chaskey et qui arrive à récupérer les clés de deux utilisateurs d'un groupe de 2^{32} en faisant 2^{32} requêtes par utilisateur.

Évidemment, tous les sujets abordés pendant cette thèse laissent des questions ouvertes, qui demandent à être explorées. Une première voie est de continuer de travailler sur l'algorithme MAC Chaskey. L'étude et les attaques que nous avons effectuées sur cet algorithme nous montrent que sa construction est perfectible. Par exemple, la clé K qui est XORée à la sortie de chaque application de la permutation π, s'élimine lors de l'itération suivante. Il serait intéressant d'étudier la paramétrisation de cette clé par le nombre de tours ou par un autre biais. En outre, étant un algorithme très récent, il existe très peu d'articles le concernant dans la littérature. Il serait donc intéressant de continuer à étudier Chaskey dans le but de proposer des attaques pratiques qui améliorent ou complètent notre attaque. Il faudrait également voir s'il est possible de pallier ses faiblesses à coût presque constant.

Pour conclure, une autre piste serait d'étudier la possibilité de trouver d'autres applications de notre méthode de chaînes parallèles. En effet, les attaques par collision sont utilisées dans plusieurs contextes, c'est-à-dire pour attaquer des fonctions de hachage, des algorithmes de chiffrement par blocs, etc. Dans cette thèse, nous avons pu appliquer cette technique sur des algorithmes de chiffrement par blocs et des codes d'authentification de messages. Néanmoins, toutes les constructions que nous avons attaquées suivent le schéma d'Even-Mansour. Il serait donc intéressant d'explorer la possibilité d'utiliser les chaînes parallèles pour attaquer des constructions différentes.

Bibliographie

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash : A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, pages 489–508, 2012.

[ABM13] Elena Andreeva, Andrey Bogdanov, and Bart Mennink. Towards Understanding the Known-Key Security of Block Ciphers. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, pages 348–366, 2013.

[ALL12] Farzaneh Abed, Eik List, and Stefan Lucks. On the Security of the Core of PRINCE Against Biclique and Differential Cryptanalysis. *IACR Cryptology ePrint Archive*, 2012 :712, 2012.

[Bab95] S. H. Babbage. Improved Exhaustive Search Attacks on Stream Siphers. In *1995., European Convention on Security and Detection*, pages 161–166, May 1995.

[BC14] Eli Biham and Yaniv Carmeli. An Improvement of Linear Cryptanalysis with Addition Operations with Applications to FEAL-8X. In *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, pages 59–76, 2014.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 208–225, 2012.

[BCKL15] Christina Boura, Anne Canteaut, Lars R. Knudsen, and Gregor Leander. Reflection Ciphers. *Designs, Codes and Cryptography*, pages 1–23, November 2015.

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC : Fast and Secure Message Authentication. In *Advances in*

Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, pages 216–233, 1999.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT : An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 450–466, 2007.

[BKL+12] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, François-Xavier Standaert, John P. Steinberger, and Elmar Tischhauser. Key-Alternating Ciphers in a Provable Setting : Encryption Using a Small Number of Public Permutations - (Extended Abstract). In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 45–62, 2012.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of Cipher Block Chaining. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, pages 341–358, 1994.

[BMS05] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved Time-Memory Trade-Offs with Multiple Data. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, pages 110–127, 2005.

[Bol85] Béla Bollobás. *Random Graphs*. Academic Press, London - New York, 1985.

[BR94] Mihir Bellare and Phillip Rogaway. Entity Authentication and Key Distribution. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 232–249, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[BR05] John Black and Phillip Rogaway. CBC MACs for Arbitrary-Length Messages : The Three-Key Constructions. *J. Cryptology*, 18(2) :111–131, 2005.

[Bre80] Richard P. Brent. An Improved Monte Carlo Factorization Algorithm. *BIT*, 20 :176–184, 1980.

[BRW04] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, pages 389–407, 2004.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013 :404, 2013.

[BU02] John Black and Hector Urtubia. Side-Channel Attacks on Symmetric Encryption Schemes : The Case for Authenticated Encryption. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 327–338, 2002.

[BW99] Alex Biryukov and David Wagner. Slide Attacks. In *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings*, pages 245–259, 1999.

[BW00] Alex Biryukov and David Wagner. Advanced Slide Attacks. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 589–606, 2000.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 272–288, 2009.

[CFG+14] Anne Canteaut, Thomas Fuhr, Henri Gilbert, María Naya-Plasencia, and Jean-René Reinhard. Multiple Differential Cryptanalysis of Round-Reduced PRINCE. In *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, pages 591–610, 2014.

[CKM00] Don Coppersmith, Lars R. Knudsen, and Chris J. Mitchell. Key Recovery and Forgery Attacks on the MacDES MAC Algorithm. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 184–196, 2000.

[CLS15] Benoit Cogliati, Rodolphe Lampe, and Yannick Seurin. Tweaking Even-Mansour Ciphers. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 189–208, 2015.

[CMS11] Sanjit Chatterjee, Alfred Menezes, and Palash Sarkar. Another Look at Tightness. In *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, pages 293–319, 2011.

[CNV13] Anne Canteaut, María Naya-Plasencia, and Bastien Vayssière. Sieve-in-the-Middle : Improved MITM Attacks. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 222–240, 2013.

[CS14] Shan Chen and John P. Steinberger. Tight Security Bounds for Key-Alternating Ciphers. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 327–350, 2014.

[CS15] Benoit Cogliati and Yannick Seurin. On the Provable Security of the Iterated Even-Mansour Cipher Against Related-Key and Chosen-Key Attacks. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 584–613, 2015.

[Dae91] Joan Daemen. Limitations of the Even-Mansour Construction. In *Advances in Cryptology - ASIACRYPT '91, International Conference on the Theory and Applications of Cryptology, Fujiyoshida, Japan, November 11-14, 1991, Proceedings*, pages 495–498, 1991.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Trans. Information Theory*, 22(6) :644–654, 1976.

[DKS12] Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in Cryptography : The Even-Mansour Scheme Revisited. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 336–354, 2012.

[DP15] Patrick Derbez and Léo Perrin. Meet-in-the-Middle Attacks and Structural Analysis of Round-Reduced PRINCE. In *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, pages 190–216, 2015.

[DR99] Joan Daemen and Vincent Rijmen. AES Proposal : Rijndael, 1999.

[EM91] Shimon Even and Yishay Mansour. A Construction of a Cipher from a Single Pseudorandom Permutation. In *Advances in Cryptology - ASIACRYPT '91, Fujiyoshida, Japan, November 11-14, 1991, Proceedings*, pages 210–224, 1991.

[EM97] Shimon Even and Yishay Mansour. A Construction of a Cipher from a Single Pseudorandom Permutation. *J. Cryptology*, 10(3) :151–162, 1997.

[FIP77] FIPS PUB 46. Data Encryption Standard (DES). FIPS Publications, janvier 1977. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[FIP01] FIPS PUB 197. Advanced Encryption Standard (AES). FIPS Publications, novembre 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[FIP02] FIPS PUB 180-2. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. FIPS Publications, august 2002.

[FIP15] FIPS PUB 202. SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions. FIPS Publications, august 2015.

[FJM14] Pierre-Alain Fouque, Antoine Joux, and Chrysanthi Mavromati. Multi-user Collisions : Applications to Discrete Logarithm, Even-Mansour and PRINCE. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, pages 420–438, 2014.

[FO89] Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, pages 329–354, 1989.

[Fou11] Jean-Claude Fournier. *Théorie des graphes et applications*. Hermès - Lavoisier, 2nd edition, 2011.

[FP15] Pooya Farshim and Gordon Procter. The Related-Key Security of Iterated Even-Mansour Ciphers. In *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, pages 342–363, 2015.

[GJMN15] Jian Guo, Jérémy Jean, Nicky Mouha, and Ivica Nikolic. More Rounds, Less Security ? *IACR Cryptology ePrint Archive*, 2015 :484, 2015.

[GNL11] Zheng Gong, Svetla Nikova, and Yee Wei Law. KLEIN : A New Family of Lightweight Block Ciphers. In *RFID. Security and Privacy - 7th International Workshop, RFIDSec 2011, Amherst, USA, June 26-28, 2011, Revised Selected Papers*, pages 1–18, 2011.

[Gol97] Jovan Dj. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 239–255, 1997.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 326–341, 2011.

[Hel80] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4) :401–406, 1980.

[IK03] Tetsu Iwata and Kaoru Kurosawa. OMAC : One-Key CBC MAC. In *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, pages 129–153, 2003.

[JLR00] Svante Janson, Tomasz Luczak, and Andrzej Rucinski. *Random Graphs*. Wiley, New York, 2000.

[JNP+13] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, Lei Wang, and Shuang Wu. Security Analysis of PRINCE. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, pages 92–111, 2013.

[Jou09] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 1st edition, 2009.

[JPS03] Antoine Joux, Guillaume Poupard, and Jacques Stern. New Attacks against Standardized MACs. In *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, pages 170–181, 2003.

[Jut00] Charanjit S. Jutla. Encryption Modes with Almost Free Message Integrity. *IACR Cryptology ePrint Archive*, 2000 :39, 2000.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC : Keyed-Hashing for Message Authentication, 1997.

[Ker83] Auguste Kerckhoffs. La Cryptographie Militaire. *Journal des sciences militaires*, 9 :5–83, 161–191, janvier, février 1883.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.) : Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KR96] Joe Kilian and Phillip Rogaway. How to Protect DES Against Exhaustive Key Search. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 252–267, 1996.

[KR01] Joe Kilian and Phillip Rogaway. How to Protect DES Against Exhaustive Key Search (an Analysis of DESX). *J. Cryptology*, 14(1) :17–35, 2001.

[KR11] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.

[Kro06] Ted Krovetz. Message Authentication on 64-Bit Architectures. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, pages 327–341, 2006.

[Leu15] Gaëtan Leurent. Differential and Linear Cryptanalysis of ARX with Partitioning – Application to FEAL and Chaskey. *IACRCryptology ePrint Archive*, 2015 :968, 2015. `http://eprint.iacr.org/2015/968`.

[LJW13] Leibo Li, Keting Jia, and Xiaoyun Wang. Improved Meet-in-the-Middle Attacks on AES-192 and PRINCE. *IACR Cryptology ePrint Archive*, 2013 :573, 2013.

[LPS12] Rodolphe Lampe, Jacques Patarin, and Yannick Seurin. An Asymptotically Tight Security Analysis of the Iterated Even-Mansour Cipher. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 278–295, 2012.

[LST12] Will Landecker, Thomas Shrimpton, and R. Seth Terashima. Tweakable Blockciphers with Beyond Birthday-Bound Security. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 14–30, 2012.

[Mav15] Chrysanthi Mavromati. Key-Recovery Attacks Against the MAC Algorithm Chaskey. In *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, pages 205–216, 2015.

[Men12] Alfred Menezes. Another Look at Provable Security. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, page 8. Springer, 2012.

[Men15] Bart Mennink. XPX : Generalized Tweakable Even-Mansour with Improved Security Guarantees. Cryptology ePrint Archive, Report 2015/476, 2015. `http://eprint.iacr.org/`.

[MMH+14a] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey : An Efficient MAC Algorithm for 32-bit Microcontrollers. *IACR Cryptology ePrint Archive*, 2014 :386, 2014.

[MMH+14b] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey : An Efficient MAC Algorithm for 32-bit Microcontrollers. In *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, pages 306–323, 2014.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NIS01] NIST SP 800-38A. Recommendation for Block Cipher Modes of Operation : Methods and Technics. NIST Special Publication 800-38A, 2001. `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf`.

[NIS04] NIST SP 800-38C. Recommendation for Block Cipher Modes of Operation : The CCM Mode for Authentication and Confidentiality. NIST Special Publication 800-38C, mai 2004. `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf`.

[NIS07] NIST SP 800-38D. Recommendation for Block Cipher Modes of Operation : Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, novembre 2007. `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`.

[Niv04] Gabriel Nivasch. Cycle Detection Using a Stack. *Inf. Process. Lett.*, 90(3) :135–140, 2004.

[Pol75] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3) :331–334, 1975.

[PvO95] Bart Preneel and Paul C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, pages 1–14, 1995.

[PY04] Kenneth G. Paterson and Arnold K. L. Yau. Padding Oracle Attacks on the ISO CBC Mode Encryption Standard. In *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, pages 305–323, 2004.

[QD89a] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search ? Application to DES (Extended Summary). In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, pages 429–434, 1989.

[QD89b] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 408–413, 1989.

[RBB03] Phillip Rogaway, Mihir Bellare, and John Black. OCB : A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3) :365–403, 2003.

[Rog00] Phillip Rogaway. PMAC : A Parallelizable Message Authentication Code, 2000.

[Rog04] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, pages 16–31, 2004.

[SBY+13] Hadi Soleimany, Céline Blondeau, Xiaoli Yu, Wenling Wu, Kaisa Nyberg, Huiling Zhang, Lei Zhang, and Yanfeng Wang. Reflection Cryptanalysis of PRINCE-Like Ciphers. In *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, pages 71–91, 2013.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, pages 181–195, 2007.

[STA+] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose. Minalpher v1. `http://competitions.cr.yp.to/round1/minalpherv1.pdf`.

[Ste12] John P. Steinberger. Improved Security Bounds for Key-Alternating Ciphers via Hellinger Distance. *IACR Cryptology ePrint Archive*, 2012 :481, 2012.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 534–546, 2002.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1) :1–28, 1999.

Table des figures

Liste des algorithmes

Table des matières

Index