Sorbonne Université Campus Pierre et Marie Curie

RAPPORT SUR L'IMPLEMENTATION DE L'ATTAQUE GENERIQUE CONTRE LE CHIFFREMENT DOUBLE

Auteurs :

Asmaa Cherief

Tingting LI

SFPN

Encadrant :

Charles BOUILLAGUET

10 juin 2020

Remerciements

Avant tout, nous remercions très chaleureusement, et exprimons notre gratitude à tous ceux qui adoucissent notre confinement et qui font en sorte de nous préserver au maximum.

En second lieu, nous tenons à remercier du fond du cœur nos chers parents qui ont été toujours à nos côtés pour nous aider, nous guider et nous soutenir. Merci infiniment.

Nous réservons une couronne de remerciements qui restera éternelle à notre encadrant Monsieur Charles BOUILLAGUET pour ses précieux conseils, ses judicieuses orientations, sa disponibilité et son soutien non-négligeable.

À nos enseignants ainsi que tout le corps professoral et administratif de la Sorbonne Université Campus Pierre et Marie Curie, pour leurs efforts déployés durant le cursus, afin de nous assurer une formation de qualité.

Nous saisissons également cette occasion pour adresser nos profonds remerciements aux deux chercheurs canadiens Paul C. van Oorschot, Michael J. Wiener, qui indirectement, nous ont énormément aidé, notamment en ce qui concerne la partie pratique de ce projet.

Merci à tous ceux qui ont contribué de prés ou de loin à l'élaboration de ce travail.

Abstract

Cryptography is commonly defined as the art of protecting information. On the contrary, cryptanalysis is all the methods used to bypass these protections. These two disciplines are grouped under the generic term cryptology. Historically, cryptography and cryptanalysis have been widely used and studied for and by armies in order, on the one hand, to ensure the confidentiality of transmissions and, on the other hand, to collect sensitive information about their adversaries.
Nowadays, with the development of many digital tools, whether for communication or information storage, etc., and the widespread use of them, cryptology is one of the leading disciplines in the fields of computer science and mathematics. Indeed, in order to ensure the confidentiality, the authentication and the integrity of a conversation (telephone, e-mail, etc.) or data (personal data, cloud, etc.), it is necessary to use cryptography.
Our project treats one of the fundamental categories of encryption that exists : block cipher, which is the process of dividing the plaintext into blocks of fixed size and then encrypting them using keys. In particular, the main theme addressed in our project concerns the double encryption, which consists of using the same block cipher twice in a row, with two different keys. Our goal is to program an algorithm, called "**Parallel collision search**", which allows us to attack this type of encryption. In order to make the implementation more efficient and high performance, we can parallelize the algorithm using several machines.
Our first step is to study in detail the article "**Parallel Collision Search with Cryptanalytic Applications**", published in 1996 by the two Canadian researchers Paul C. van Oorschot and Michael J. Wiener. Then is followed by a discussion of how the results (assumed to be optimal) from the paper compare with the results of the simulation of our own implementation.
The "**Parallel collision search**" algorithm could work with DES encryption. Only in our case we had to choose small size keys to test our code, given the performance of our machines. The use of AES-128 block encryption is due to the fact that it is more practical.

Keywords : Block Cipher, Advanced Encryption Standard (AES), Dual Ciphers, Data Encryption Standard (DES), 2-DES, 2-AES, Data security, Cryptanalysis, Meet-in-the-middle attack, Parallel Collsiion Search.

Table des matières

Notations

— m : le message clair

— C : le message chiffré

— (k_1, k_2) : la paire de clefs utilisées pour le chiffrement double

— $E_x(m)$: la fonction de chiffrement qui chiffre le message m avec la clef x

— $D_x(C)$: la fonction de déchiffrement qui déchiffre le message C avec la clef x

— N : l'ensemble des clefs possibles

— n : le nombre d'élément dans l'ensemble N

— S : l'ensemble des valeurs possibles pour le message m chiffré une fois

— x_0 : le premier élément d'un trail

— x_D : le dernier élément d'un trail

— D : la longueur d'un trail

— k_p : le nombre de bits significatifs dans la paire de clefs

— l : la condition d'arrêt d'un trail qui est fixée à "l bits de x_i sont à zéro"

— f : la fonction qui permet de passer d'un x_i à x_{i+1}

Introduction

Communément, la cryptographie est définie comme l'art de protéger les informations. Au contraire, la cryptanalyse, elle, regroupe les techniques introduites afin de passer outre ces protections et de retrouver ces informations secrètes. Ces deux disciplines sont regroupés sous le terme générique de cryptologie. Historiquement, cryptographie et cryptanalyse ont vastement été employées et étudiées pour et par les armées afin, d'un côté, d'assurer la confidentialité des transmissions et de l'autre, de collecter des informations sensibles sur leurs adversaires.

Aujourd'hui, le développement de nombreux outils numériques, que ce soit pour la communication, le stockage d'informations etc. et la généralisation de leur emploi, font de la cryptologie l'une des disciplines phares de l'informatique et des mathématiques. En effet, afin d'assurer la confidentialité, l'authentification ou l'intégrité d'une conversation (téléphone, e-mail, etc.) ou de données (données personnelles, cloud, etc.), il est nécessaire de recourir à la cryptographie.

Notre projet revêt l'une des catégories fondamentales de chiffrement existantes ; le chiffrement par blocs. Ce dernier revient à découper le message clair en blocs de taille fixe, puis les chiffrer (séquentiellement ou en parallèle selon l'algorithme) en utilisant des clés.

Plus particulièrement, le thème abordé dans ce manuscrit concerne le cryptosystème des chiffrements doubles et qui consiste à utiliser le même dispositif de chiffrement par blocs deux fois de suite, avec deux clefs différentes. Le but des travaux effectués étant de programmer un algorithme, appelé "**Parallel collision search**", et qui permet de casser ce type de chiffrement. Une parallélisation de ce programme sera effectué par la suite sur plusieurs machines afin de rendre efficace et "haute performante" l'implantation.

Pour ce faire, il a fallu dans un premier temps, étudier en détail l'article intitulé "**Parallel Collision Search with Cryptanalytic Applications**", publié en 1996 par les deux chercheurs canadiens Paul C. van Oorschot et Michael J. Wiener, et qui décrit en profondeur et de manière détaillée le sujet en se basant sur des simulations de l'algorithme sur le chiffrement double-DES. Puis dans un second temps aborder la comparaison entre les résultats (qu'on suppose optimaux) notés sur l'article, et ceux de la simulation de notre propre algorithme.

L'algorithme "**Parallel collision search**" pourrait fonctionner avec le chiffrement DES. Seulement, dans notre cas il fallait choisir des clés de petite taille pour tester le code, en vue des performances de nos machines. C'est pour cela que nous avons recouru à l'usage de AES-128, qui est plus

pratique, ce qui nous donne la possibilité de choisir la taille de la clef et donc de rendre réalisable notre travail.

Organisation du manuscrit

Dans une première partie, nous commencerons par rappeler les notions de base de la cryptographie.

Nous spécialiserons notre étude par la suite dans une deuxième partie, sur le chiffrement double ses principes et vulnérabilités. Ensuite nous discuterons brièvement des algorithmes et méthodes connues précédemment dans la recherche de collision.

Une troisième partie nous fournira les détails de notre algorithme "**Parallel Collision Search**" théoriquement et pratiquement parlant, ainsi que la définition et l'usage des différentes fonctions du code.

Dans une dernière partie, nous nous intéresserons à l'étude de nos résultats et la comparaison avec ceux obtenus dans l'article, tout en soulevant les possibles anomalies rencontrées au cours du chemin.

Partie 1

Généralités

1.1 Préliminaire

La cryptographie est traditionnellement utilisée pour sécuriser la transmission de données sensibles dans un environnement non sécurisé. Le principe est d'utiliser un algorithme de chiffrement, qui est une technique permettant d'assurer la confidentialité des données ou des communications en transformant les données en une suite de symboles compréhensibles par toute personne ne possédant pas un certain secret, appelé une clé.

Néanmoins, pour que le destinataire retrouve le message clair à partir de cette suite inintelligible, il va devoir employer un algorithme de déchiffrement. Celui-ci ne pourra être appliqué sans la connaissance du secret.

La sécurité des cryptosystèmes est déterminée par le nombre d'opérations nécessaires pour retrouver le message en clair, à partir du chiffré, lorsque la clé n'est pas connue. Nous considérons qu'un cryptosystème dont le niveau de sécurité est n bits repose sur des problèmes dont la résolution nécessite au moins 2^n opérations.

Les primitives cryptographiques sont divisées en deux familles, qui dépendent de la manière d'exploiter ces clés : la cryptographie à clé secrète (ou cryptographie symétrique) et la cryptographie à clé publique (ou cryptographie asymétrique). De manière générale, la cryptographie à clé secrète fournit des outils sensiblement plus efficaces (en terme de stockage et de temps) mais elle nécessite au préalable une phase d'échange de clé (qui sera utilisée aussi bien pour chiffrer que pour déchiffrer) entre les participants.

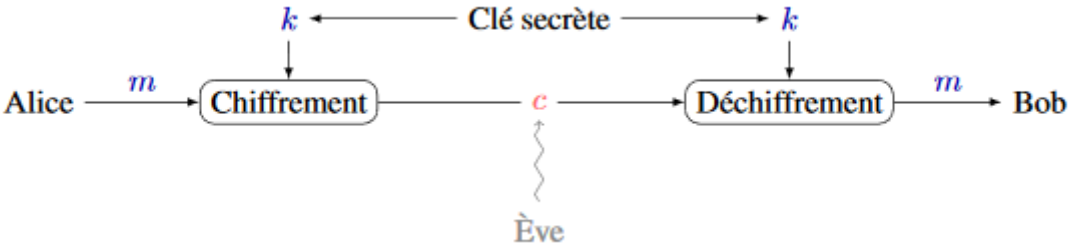

FIGURE 1.1 – Illustration du fonctionnement d'un schéma de chiffrement symétrique

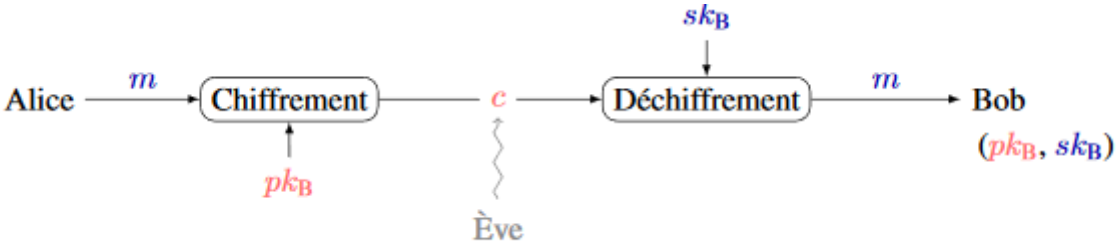

FIGURE 1.2 – Illustration du fonctionnement d'un schéma de chiffrement asymétrique

Le principe général de ces cryptosystèmes à clef publique est d'employer une fonction simple à appliquer mais difficile à inverser pour transformer le message. La sécurité de ces cryptosystèmes dépend donc en partie de la difficulté d'inverser cette fonction.

Ce qui n'est pas toujours le cas pour la cryptographie symétrique. En effet, prenons par exemple le cas du One-Time Pad qui est basé sur le XOR. Malgré le fait que ça soit une fonction facile à inverser, ce dernier est qualifié de chiffrement à secret parfait, c'est-à-dire que la connaissance du message chiffré ne donne aucune information sur le message clair. C'est alors le caractère aléatoire du masque qui protège le secret et pas la difficulté des calculs.

De nos jours, la plupart des cryptosystèmes utilisés en pratique sont basés sur des problèmes issus de la théorie des nombres : la factorisation d'un entier en ses diviseurs premiers pour RSA, la résolution du logarithme discret dans un corps fini ou sur des courbes elliptiques.

Dans ce manuscrit, nous allons nous intéresser au standard DES, qui est un chiffrement par blocs à clé symétrique publié par le National Institute of Standards and Technology .

Nous étudierons plus particulièrement les deux applications successives de l'algorithme DES sur le même bloc, appelé double-DES ou 2-DES, les vulnérabilités ainsi que les attaques génériques effectuées sur ce dernier. Seulement en pratique, nous utiliserons comme substitut : l'AES-128 avec clefs tronquées ce qui nous donnera la possibilité de choisir la taille des clefs.

1.2 Les chiffrements par blocs et leurs composants

1.2.1 Le chiffrements par bloc

Il s'agit d'une des deux grandes catégories de chiffrement en cryptographie symétrique, l'autre étant le chiffrement par flot. Le nom du premier vient du fait que l'émetteur découpe le message à chiffrer en blocs de longueur fixe et traite ces blocs successivement.

Les algorithmes de chiffrement par blocs permettent uniquement de chiffrer des messages de taille fixe. Il existe différentes manières d'utiliser ces primitives pour chiffrer des messages de taille quelconque, appelés modes opératoires.

Les méthodes de construction de systèmes de chiffrement par blocs sont empiriques et la plus utilisée est celle des algorithmes itératifs, dont le schéma de Feistel. Ces algorithmes sont composés de plusieurs tours, *à chaque tour* ils font agir une **même** fonction F paramétrée par une **différente** sous-clef K_p générée à partir de la clef maître K et d'un algorithme de diversification de clefs.

1.2.2 Le schéma de Feistel

Soit une fonction f qui prend comme argument un mot de n bits. L'algorithme de chiffrement va procéder en chiffrant des blocs de $2n$ bits, qu'on partage en deux parties de n bits chacune : les parties gauche G et droite D.

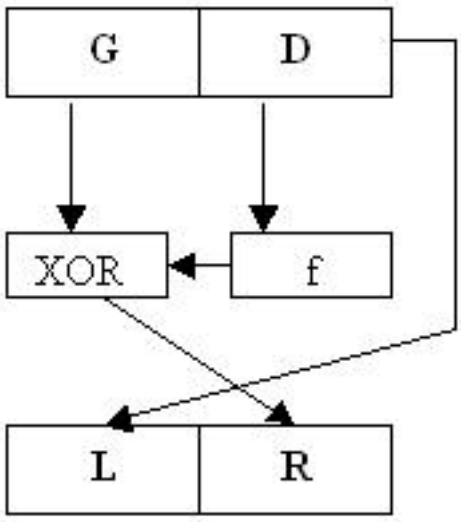

FIGURE 1.3 – Fonctionnement du schéma de Feistel

L'image du bloc **(G,D)** est le bloc **(L,R)**,

$$\text{avec : } \begin{cases} L = D \\ R = GXORf(D) \end{cases}$$

Cette transformation est bijective, car si on a un couple **(L,R)**, on retrouve bien **(G,D)**

$$\text{par : } \begin{cases} D = L \\ G = RXORf(L) \end{cases}$$

La partie droite n'a pas été transformée mais juste envoyée à la gauche. Il faut donc répéter le schéma de Feistel un certain nombre de fois : nous parlons de tours.

Parmi les chiffrements de Feistel, nous trouvons en particulier le chiffrement **DES**.

Remarque : Dans le chiffre de Feistel le chiffrement et le déchiffrement les mêmes, à quelques détails près : il faut permuter les deux moitiés et inverser l'ordre des sous-clefs.

Partie 2

Double chiffrement

2.1 Principe

Le double chiffrement consiste à appliquer deux fois successivement le même algorithme de chiffrement par blocs (DES, AES...).

L'idée de ce chiffrement multiple afin d'améliorer la sécurité des systèmes est apparue dans le but de cacher certaines propriétés statistiques dans les cas où des chiffrements simples et facilement exploitables seraient utilisés.

Cette méthode permet normalement de s'assurer qu'en cas de vulnérabilité d'un des systèmes utilisés, les autres pourront prendre le relais et assurer encore une forte protection.

Malencontreusement, cette technique n'a pas été très efficace dans le cas du chiffrement par blocs du 2-DES. En effet, il n'a pas pu être utilisé à cause de ses clefs de tailles petites. Il peut donc être cassé par une recherche exhaustive de l'espace des clefs, malgré la multiplicité de son application. Avec du matériel spécifique ou de grands réseaux de stations de travail, il est maintenant possible de déchiffrer ce cryptogramme.

En général, ce sur-chiffrement n'apporte pas de sécurité supplémentaire par rapport au chiffrement simple du fait qu'il présente des risques, malgré le fait que la taille des clefs k_p semble être doublée et la complexité d'une recherche exhaustive naïve de la paire de clefs nécessite $\mathcal{O}(2^{2k_p})$ évaluations de l'algorithme de chiffrement.

Cette attaque justifie l'utilisation du 3-DES qui consiste à réaliser un sur-chiffrement triple avec le chiffrement DES, afin d'élargir l'espace des clefs et rendre plus complexe l'exploitation de la faille du DES. Ce dernier est utilisé dans les pass Navigo ainsi que dans l'application StopCovid (même si une adaptation de l'attaque par compromis temps-mémoire permet de retrouver la clef au bout de 2^{112} chiffrements DES sous une attaque à clairs connus).

2.2 Méthodes d'exploitation naïves des vulnérabilités du double chiffrement

Il existe plusieurs méthodes qui permettent de casser le double chiffrement. Nous allons dans la suite en lister quelques unes.

1. **L'attaque Meet-in-the-Middle** :

   Elle consiste à choisir une paire de clefs (k_1, k_2) de taille k_p bits, puis chiffrer une fois le message clair et déchiffrer une fois le message doublement chiffré.

   Nous avons donc : $m = D_{k_2}(D_{k_1}(C))$ et $C = E_{k_1}(E_{k_2}(m))$ où m est le message clair et C le message doublement chiffré par la paire de clefs (k_1, k_2).

   Cela donne alors :

   $$\begin{cases} E_{k_2}(m) & = E_{k_2}(D_{k_2}(D_{k_1}(C))) \\ E_{k_2}(m) & = D_{k_1}(C) \end{cases}$$

   et

   $$\begin{cases} D_{k_1}(C) & = D_{k_1}(E_k(E_{k_2}(m))) \\ D_{k_1}(C) & = E_{k_2}(m) \end{cases}$$

   donc :

   $$E_{k_2}(m) = D_{k_1}(C)$$

   Nous retrouvons ainsi la paire de clef (k_1, k_2) en $O(2^{2k_p})$.

2. Quisquater et Delescaille **Finding DES collisions** :

   Le but de cette méthode est de trouver des collisions pour l'algorithme de chiffrement DES, en se basant sur la sauvegarde des points distingués (distinguished points).

   Afin de trouver une paire de clef (k_1, k_2) qui chiffre un message clair m en message chiffré C, cette méthode utilise pour itérer la fonction f tel que : $f(x) = g(E_{k_1}(m))$ où g est une fonction qui chiffre un bloc de texte de 64 bits en une clef de 56 bits.

   La perte d'informations dans la fonction g conduit à l'existence des "pseudo-collisions". En effet, si nous avons deux clefs qui chiffrent des textes différents, g peut éventuellement retourner la même valeur. Sauf que seule une collision sur 2^8 dans f est la bonne.

   Nous avons donc besoin de 2^{32} étapes pour trouver la bonne collision et nous nous attendons à avoir 2^8 pseudo-collisions avant de tomber sur la bonne.

   Cependant, les très grandes exigences de mémoire de cette méthode la rendent infaisable dans la pratique, sauf pour les très petits exemples.

C'est pour ça qu'on s'intéresse aux algorithmes de recherche de collision. En effet, ces algorithmes de recherche de cycle réussissent à trouver des collisions, et ne nécessitent qu'un espace mémoire négligeable.

Partie 3

Recherche de Collision Parallèle

3.1 Recherche de Collision Parallèle (en théorie)

Le **"Parallel Collision Search"** est une méthode de recherche de collision par pas pseudo-aléatoire utilisée contre le chiffrement double. Elle sert à réaliser des attaques de type Meet-in-the-Middle.

Remarque : L'utilisation la plus connue de cette méthode est celle des logarithmes discrets dans des groupes cycliques spécifiques (ex. $\mathbf{Z}_p^{\times}$) sur des courbes elliptiques.

Dans l'algorithme **"Parallel Collision Search"**, nous avons deux fonctions $f_1 : N_1 \to S$ et $f_2 : N_2 \to S$ (f_1 est la fonction de chiffrement et f_2 la fonction de déchiffrement par exemple) où N_1 et N_2 sont les ensembles de clefs et S l'ensemble des valeurs possibles pour le message chiffré une fois.

Pour obtenir une collision, nous avons besoin de trouver le couple (a, b) où $a \in N_1$ et $b \in N_2$ tel que $f_1(a) = f_2(b)$.

Nous posons $n_1 = |N_1|$ et $n_2 = |N_2|$. Dans notre cas, nous avons $n_1 = n_2$, les deux clefs appartiennent au même ensemble. Nous notons donc $n = n_1 = n_2$.

Pour le double chiffrement, nous avons un couple de messages clair/chiffré (m, C) avec $C = E_{k_2}(E_{k_1}(m))$. Nous notons $f_1(x) = E_x(m)$ et $f_2(x) = D_x(C)$. Donc trouver une collision revient à trouver (k_1, k_2) tel que $E_{k_2}(m) = D_{k_1}(C)$. Et par conséquent réussir à trouver $f_1(k_1) = f_2(k_2)$.

Il est possible d'avoir plusieurs couples qui satisfont la condition. En effet, il y a 2^{2k_p} paires de clefs, chaque paire a une chance sur 2^n (n étant la taille du bloc) de correspondre "par accident", donc il va y avoir $2^{(2k_p-n)}$ paires de clefs valables. c'est pour cela que nous avons besoin de tester ces couples sur un second texte chiffré afin d'augmenter la probabilité que ça soit la “golden collision” (c.a.d la “bonne” paire de clefs qui a chiffré le message).

3.2 Recherche de Collision Parallèle (en pratique)

Une première approche différente, de celle que nous allons utiliser, consiste à faire cette recherche de collision naïvement, de la façon suivante :

— Calculer toutes les valeurs de $f_1(k_1) = E_{k_1}(m)$ pour tout $k_1 \in n$ et sauvegarder les couples $(k_1, f_1(k_1))$ trouvés dans une table de hachage.
— Calculer $f_2(k_2) = D_{k_2}(C)$ pour tout $k_2 \in n$, tout en vérifiant dans la table de hachage, si collision.
— S'il y a une collision, alors tester si la paire trouvée est la "bonne" sur un autre message chiffré.

Nous pouvons voir que cette méthode n'est clairement pas optimale. En terme de complexité, nous avons besoin de $\frac{n_1+n_2}{2}$ évaluations de fonction et de l'espace mémoire pour n_1 paires de clefs.

Une seconde approche, **celle que nous allons utilisé dans dans la suite**, se base sur la recherche de collision **"sans mémoire"**.

Pour commencer, notre but est de trouver des collisions, c'est-à-dire de trouver des a et b tel que $f(a) = f(b)$. Pour cela, nous devons produire des "chemins", que nous allons appeler dans la suite "trails". Nous avons besoin de construire une fonction $f : S \to S$ et qui devrait être tellement complexe qu'elle ressemblera à une fonction aléatoire, donc sa répartition sera uniforme.

Nous allons procéder de la façon suivante pour générer un trail : nous générons aléatoirement une clef $x_0 \in S$, puis nous calculons $x_i = f(x_{i-1})$ jusqu'à x_D. x_D étant défini selon une condition d'arrêt que nous avons fixé à "les l derniers bits de x_i sont à zéro". Selon l'article, le choix de la condition d'arrêt ne va en aucun cas changer les performances de l'algorithme. Le point x_D est appelé le point distingué. Une fois que nous avons le points distingué, nous le sauvegardons dans une liste et nous obtiendrons ainsi un trail.

Afin de trouver une collision, nous répétons la procédure précédente dans le but de construire des trails. Nous obtiendrons une collision une fois que pour deux trails nous avons $x_0 \neq x'_0 \to x_D = x'_D{}'$. La paire ($a$,$b$) recherchée est représentée par les deux valeurs des trails se trouvant juste avant la collision $f(a) = f(b)$. Nous pouvons le voir sur la figure (en rouge).

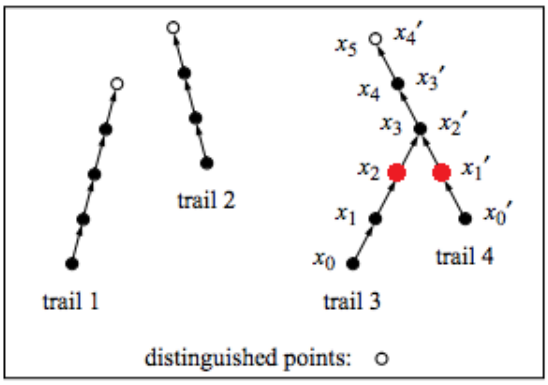

FIGURE 3.1 – trails et collisions des trails

Pour les mêmes entrées la fonctions itérée f génère toujours les mêmes sorties, et c'est ce déterminisme qui nous permet d'affirmer que si nous avons une collision pour f alors nous avons une collision entre les fonctions de chiffrement/déchiffrement f_1/f_2 pour le message donné avec la paire de clefs trouvée.

Nous pouvons avoir énormément de paires de clefs qui satisfont la condition de collision : nous avons donc besoin de produire un nombre conséquent de collisions avant de tomber sur la bonne. Pour cela, nous allons forcément répéter plusieurs fois la procédure précédente qui permet de trouver une collision.

Remarque : Les collisions n'ont pas la même probabilité d'apparition avec différentes versions de la fonction f, donc il faut faire varier de temps en temps f pour éviter de tomber tout le temps sur les même collisions.

Une fois que nous savons comment produire des collisions avec des trails, la méthode de recherche de collision parallèle consiste à chercher des collisions potentielles puis vérifier leur valabilité sur un second message chiffré, nous permettons ainsi de trouver la **"Golden Collision"** et donc de casser le double chiffrement.

3.3 Les différentes fonctions de l'algorithme et leurs usages

3.3.1 Pour les fonctions de hachage : MD5 et SHA256

Avec pour objectif de bien comprendre le fonctionnement de l'algorithme et le tester sur des cas simples. Nous avons commencé par le mettre en oeuvre pour trouver une collision entre deux fonctions de hachage cryptographiques (tronquées) soit MD5 et SHA256.

Explication des fonctions :

— `f_cut_k(fhash, val, k)` : Elle représente la fonction `f` qui permet de passer de x_i à x_{i+1}. Elle applique la fonction de hachage `fhash` sur la valeur *val* puis retourne la valeur de retour à k bits.

— `trail(fhash, k , l)` : Nous générons un `trail` avec une fonction de hachage `fhash` avec k bits significatifs et l pour la condition d'arrêt : quand les l derniers bits de x_i sont égales à zéro, alors nous arrêtons la fonction. Elle retourne (x_0, x_D, D), avec x_0 la première valeur du `trail`, x_D la dernière valeur et D la longueur du `trail`.

— `collision_detection(fhash, k, l)` : Elle retourne le couple de couplet $((a, fhash_1), (b, fhash_2))$ où $fhash_1(a) = fhash_2(b)$ et chaque valeur est tronquée à k bits. Elle applique la fonction `trail` et sauvegarde les résultats jusqu'à tomber sur une collision de `trail`. On retourne ainsi les deux valeurs des trails qui se trouvent juste avant la première collision de valeurs.

3.3.2 AES-128 avec des clefs de k_p bits significatifs

Comme alternative au DES le sujet nous propose d'utiliser le chiffrement RC5 (sur lequel nous pouvons choisir la longueur des clefs de chiffrement). En pratique, il nous a paru plus simple d'utiliser l'AES-128 avec des clefs tronquées.

Explication des fonctions :

— `new_step_cte(f, M, kp, xi, cte)` : Elle représente la fonction qui permet de passer de x_i à x_{i+1}. Elle applique la fonction f (qui peut être une fonction de chiffrement ou de déchiffrement) sur le message M avec la clef x_i (qui a k_p bits significatifs). L'entier *cte* représente la version de la fonction `new_step_cte` utilisée.

— `trail(f, msg, kp, l, x0, cte)` : Elle permet de produire un trail (x_0, x_D, D).

— `collision_detection(F, M, C, kp, l, dico, cte)` : Elle retourne le couple de clefs de k_p bits significatifs suivant : $((x, F_1), (y, F_2))$ où $F_1(M, x) = F_2(C, y)$. F est définie par la fonction $F(b)$: elle retourne la fonction de chiffrement ou déchiffrement selon l'entier b passé en paramètres. Le paramètre *dico* représente le dictionnaire qui contient tous les trails déjà trouvés. Cette fonction fait appel à la fonction *trail* pour générer des trails et les sauvegarde dans le dictionnaire *dico*. Elle s'arrête quand elle trouve une collision.

— `golden_collision_parr(F, M1, C1, M2, C2 ,kp, l)` : Elle utilise la fonction `collision_detection` pour trouver des collisions sur le couple de messages clair/chiffré (M_1, C_1). Une fois la collision trouvée, nous pouvons utiliser le couple (M_2, C_2) pour vérifier que c'est la bonne.

Partie 4

Applications

4.1 Les fonctions de hachage : MD5 et SHA256

Notre code permet de trouver une collision entre la fonction de hachage MD5 et SHA256. Voici une application :

```
In [24]: k = 20
         l = 5
         print(collision_detection(F, k, l))

         ((533242, 'openssl_md5'), (154498, 'openssl_sha256'))

In [26]: print(f_cut_k(hashlib.md5, 533242, 20))
         print(f_cut_k(hashlib.sha256, 154498, 20))

         615040
         615040
```

FIGURE 4.1 – Résultats de Recherche de Collisions entre deux fonctions de hachage

Observation. Nous pouvons trouver une collision des k premiers bits avec les fonctions de hachage MD5 et SHA256.

4.2 Le chiffrement AES-128 avec des clefs tronquées à k_p bits

1. Fixer la valeur de k_p et faire varier celle de l :

   Nous générons des clés aléatoires avec $k_p = 8$: ce qui veut dire qu'on a 8 bits significatifs dans les clefs et le reste est à zéro.

   Nous allons donc regarder le temps d'exécution moyen et le nombre de collisions que nous avons trouvé avant d'obtenir la bonne collision pour l allant de 0 à 5. Pour cela nous

allons lancer l'algorithme de recherche de collision avec des clefs générées aléatoirement avec $k_p = 8$. Voici le résultat obtenu :

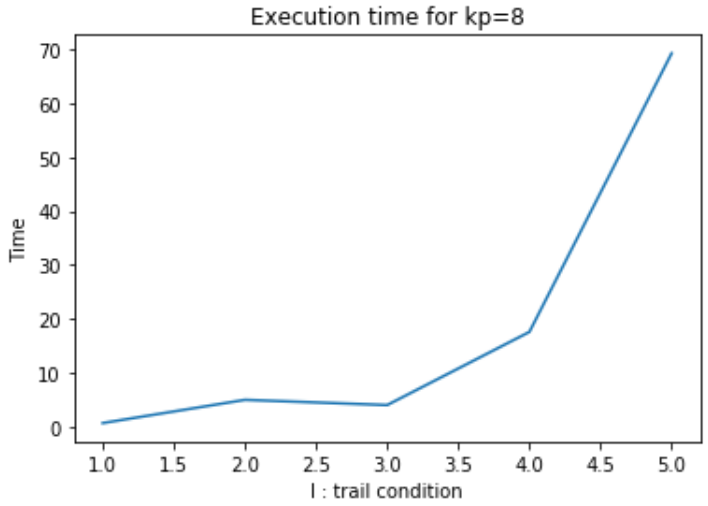

FIGURE 4.2 – Temps d'exécution pour $k_p = 8$ et l variant

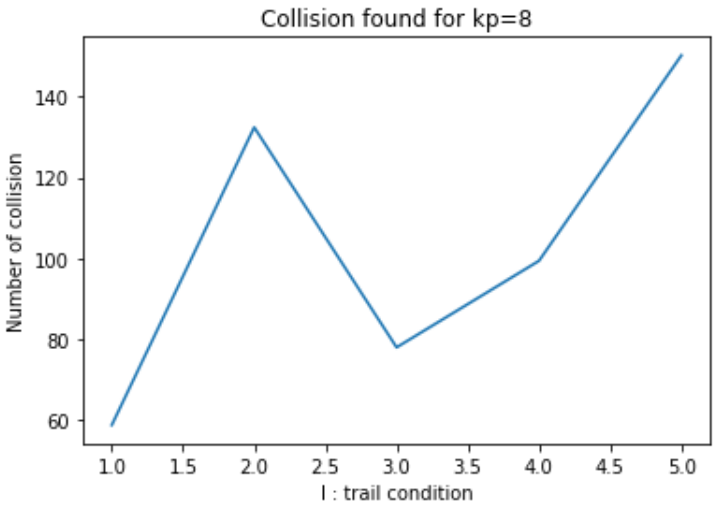

FIGURE 4.3 – Collisions trouvés pour $k_p = 8$ et l variant

En gardant ces simulations faites avec $k_p = 8$, nous effectuons plusieurs autres avec différentes valeurs de k_p et en variant toujours la valeur de l afin de comparer et vérifier la véracité et l'exactitude de ce que nous avons avancé avant, nous remarquons que :

— En moyenne, l'exécution est beaucoup plus rapide pour $l \in [1, 3]$ quelque soit la taille de la clef kp.

— Le temps d'exécution augmente très fortement à partir de $l = 4$

— Le nombre de collisions trouvées avant de tomber sur la golden collision semble être aléatoire. Ce qui est normal vu qu'il dépend de la paire de clefs générée aléatoirement.

Dans l'article, et se basant sur les résultats des simulations de leur propre code, les deux chercheurs canadiens C. van Oorschot et Michael J. Wiener affirment que la valeur de l doit dépendre de k_p. Conclusion que nous n'avons pas pu tirer étant donné que la limite raisonnable à laquelle nous avons pu varier k_p avec nos machines ne dépasse pas les 16 bits. En effet, pour $k_p = 16$ et $l = 1$, nous avons réaliser une simulation sur une paire de clefs générés aléatoirement. Le temps d'exécution était de 25067 secondes (soit environ sept heures).

C'est pour cela que dans notre cas et avec nos propres simulations, il nous a paru plus correcte de dire que nous arrivons généralement à trouver de bons résultats et surtout rapidement en fixant $l = 1$ peu importe la valeur de k_p.

2. Faire varier la valeur de k_p et fixer celle de $l = 1$:

   Dans un second temps, nous allons regarder le temps d'exécution en faisant varier la valeur de k_p. Nous obtiendrons les temps d'exécution et les nombres de collisions totales suivants :

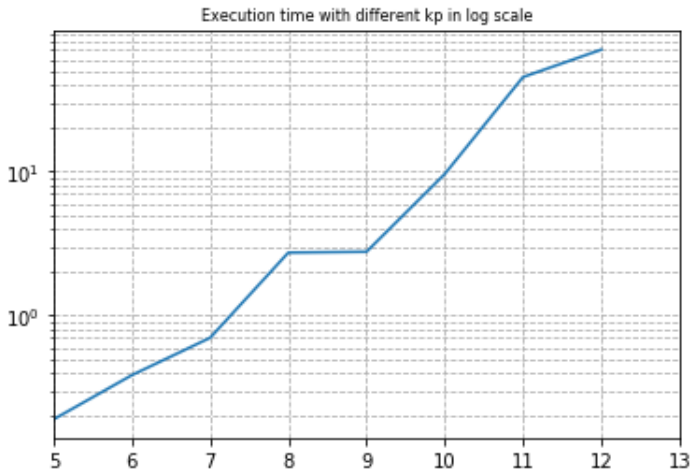

FIGURE 4.4 – Temps d'exécution en fonction de k_p en échelle logarithme

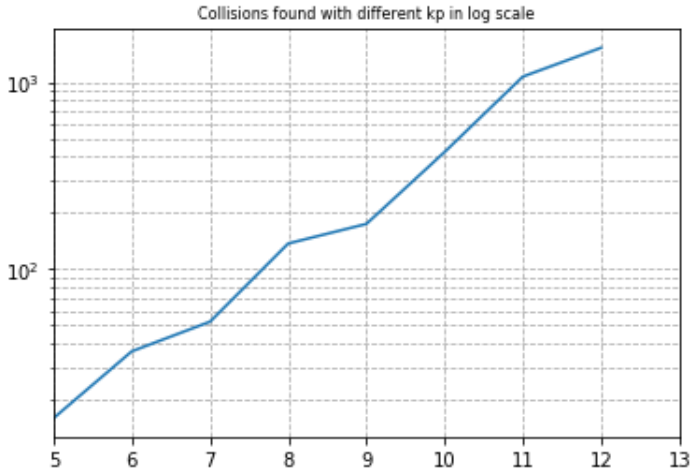

FIGURE 4.5 – Nombre de collisions trouvées en fonction de k_p en échelle logarithme

Observation. Nous observons ici que le nombre de collisions trouvées ainsi que le temps d'exécution augmentent fortement à partir de $k_p = 10$ quelque soit la valeur prise pour l.

4.3 Quelques problèmes *algorithmiques* rencontrés

1. **Problème de cycles :**

   C'est un problème que nous avons rencontré dans la fonction `'trail'`. En effet, un cycle apparaît à chaque fois que nous tombons sur l'une des valeurs précédentes de `'tmp'` (c.a.d le résultat de la fonction itérée f).

   — *Solution proposée :*

   La fixation d'un seuil (ce que nous avons appelé dans le code `max_it`) pour `'d'` (c.a.d la longueur du `trail`) à $4 * 2^l$. Ce qui marche bien pour les valeurs petites/moyennes de k_p. La multiplication par 4 est justifiée par le fait que d'après nos tests il est rare de tomber sur un cycle avant `max_it`.

   — *Critique de M. Charles Bouillaguet :*

   Il y a un risque d'avoir beaucoup de cycles si la longueur moyenne des trails s'approche trop près de $2^{k/2}$. Alors et en se basant sur **le paradoxe des anniversaire** nous savons que la probabilité de générer deux fois la même valeur le long du trail augmente sensiblement. Il est donc suffisant de garantir que l soit sensiblement plus petit que k/2.

   — *Solution retenue :* Nous avons fixé un nombre maximal d'itérations égal à : $\frac{20}{1/2^l}$ en se basant l'article `"Parallel collision search with cryptanalytic applications"`.

2. **Problème du temps d'exécution :**

   Nous avons constaté que le temps d'exécution devient beaucoup trop grand dans deux situations.

D'abord, cela arrive quand nous stockons les collisions trouvées par la fonction `collision_detection` dans une liste avant de les tester sur un second message dans le but de trouver la `golden_collision`. Et aussi quand le dictionnaire dans `collision_detection` est initialisé à 0 à chaque exécution, ce signifie que à chaque itération de `collision_detection` nous perdons les anciennes valeurs stockées.

— *Critique de M. Charles Bouillaguet :* Afin d'améliorer cela et surtout dans le but de réduire le temps d'exécution, il nous a été proposé de tester au fur et à mesure les collisions sur le deuxième message.
— *Solution retenue :* Nous avons traduit les conseils de notre encadrant par l'incorporation de la vérification dans la fonction `golden_collision`. Ensuite, Pour économiser du temps et profiter des valeurs trouvées précédemment dans `collision_detection`, il était préférable de n'initialiser le dictionnaire à 0 qu'au début de la fonction `golden_collision`. En fixant bien évidemment un seuil (taille du dictionnaire) à ne pas dépasser. Si cela arrive il suffit d'écraser (et donc sacrifier) une des anciennes valeurs (qui pourrait être la golden) pour libérer de l'espace.

3. **Problème de variation de la fonction itérée `new_step` :**

La fonction `new_step` nous permet de construire des trails. C'est donc sur cette fonction que nous désirons trouver une collision.
Nous avions commencer l'implémentation de notre algorithme avec une unique version de `new_step` qui génère les x_i de la manière suivante :
— Applique la fonction de chiffrement ou de déchiffrement une fois.
— Tronque le résultat obtenu à k_p bits.
— Bourre le résultat obtenu avec des zéros.
Le problème que nous avions eu avec cette version est qu'au bout d'un certain nombre d'itérations, nous n'arrivons plus à trouver de nouvelles collisions. Et donc seul le nombre des `idem_collision` (collisions déjà trouvées) augmente.
Une des solution que M. Charles Bouillaguet nous a proposé consistait à faire varier la fonctions `new_step`, en créant d'autres versions. Chose que nous avons réussi à faire en utilisant et en mélangeant différents opérateurs logiques (XOR, addition, et/ou logique, shift logique ...).
Or, le problème persistait lorsque nous testions l'algorithme pour des clefs de taille plus grande.
La solution retenue a été de réaliser une fonction `new_step` qui dépend d'une constante passée en paramètres. Dans cette fonction, nous générons nos clefs relativement à cette constante. Ainsi, à chaque fois que nous épuisons le quota de collisions qu'une version de

`new_step` peut trouver, nous pouvons faire varier la fonction en augmentant la valeur de la constante.

Conclusion et perspectives

L'intérêt de ce modeste travail s'inscrit dans une longue tradition d'évaluation de la sécurité des algorithmes de chiffrement au regard des cryptanalyses dont ils font l'objet compte tenu de l'importance de la sécurité qui est un paramètre primordial.

La question abordée ici, est celle de la sécurité du chiffrement double. Ce super-chiffrement désigne le résultat final d'un processus de chiffrement multiple. Pour la plupart d'entre nous, l'application de la logique de base signifie que la sécurité serait automatiquement améliorée si quelque chose est crypté à nouveau. Cependant, comme c'est le cas avec la cuisson, il est tout au sujet de la façon dont vous utilisez les ingrédients. La meilleure approche est de suivre la recette, plutôt que la logique.

Les anciennes attaques génériques sur le chiffrement double se sont avérées inefficaces, et ont révélé la nécessité de la parallélisation, ce qui se traduit par des gains plus conséquents en matière de temps de calcul.

C'est dans cette optique, que nous avons décidé d'implémenter un algorithme efficace de recherche de collision parallèle, et de s'en servir pour réaliser l'attaque contre le chiffrement double. Outre le code, l'essence même du projet consistait à étudier tous les problèmes théoriques et/ou pratiques rencontrés (probabilité de succès, gestion du volume de données, communications, facteurs limitant les performances, etc..) afin d'explorer et éclaircir les zones d'ombre.

La partie conceptuelle du projet nous a permis de mettre en relation différents enseignements acquis durant les années précédentes. En effet, afin de mener à bien ce projet, il fallait que nous ayons acquis préalablement des connaissances et des savoir-faire spécifiques dans divers domaines (cryptologie, probabilités, programmation ....etc)

Le constat qui s'impose à l'issue de ces travaux souligne le rôle paradoxal joué par la taille des clés. En effet nous rappelons que dans le cadre de notre analyse, le choix du AES a été défini comme une alternative "forcée" pour le DES, étant donné que la faisabilité du "Parallel collision search" dépend en grande partie de la possibilité de choisir la taille des clés.

Le souhait que nous avons actuellement est de pouvoir améliorer les performances de nos fonctions, en commençant par traduire le code en langage C, afin d'exploiter pleinement la parallélisation et donc d'avoir des résultats significativement plus optimaux.

Bibliographie

[1] Cryptographie - introduction à des. <https://web.maths.unsw.edu.au/~lafaye/CCM/crypto/des.htm>. [En ligne ; consulté en mai-2020].

[2] Schémas de feistel, ou chiffrement par blocs. <http://www.cryptage.org/feistel.html>. [En ligne ; consulté en mai-2020].

[3] Rc5 (chiffrement). <https://fr.wikipedia.org/wiki/RC5_(chiffrement)>, 2017. [En ligne ; consulté en mai-2020].

[4] Veron Pascal Barthelemy Pierre, Rolland Robert. *Cryptographie : principes et mises en œuvre*. Lavoisier, 2012.

[5] Charles Bouillaguet. Implantation de l'attaque générique contre le chiffrement double. <https://polsys.lip6.fr/~safey/PSFPN/2019/Sujets/psfpn-sujet-bouillaguet1.pdf>, 2020. [En ligne ; consulté le 30-Janvier-2020].

[6] Julia Chaulet. Étude de cryptosystèmes à clé publique basés sur les codes mdpc quasi-cycliques. <https://tel.archives-ouvertes.fr/tel-01599347/document>, 2017. [En ligne ; consulté en mai-2020].

[7] Itai Dinur. Tight time-space lower bounds for finding multiple collision pairs and their applications. <https://eprint.iacr.org/2020/229.pdf>, 2020. [En ligne ; consulté en mars-2020].

[8] Springer Johannes A. Buchmann. *Introduction à la cryptographie*. Sciences Sup, Dunod, 2006.

[9] Mira Nasiri. Dual representations for aes block cipher. <https://www.researchgate.net/publication/306058103_Dual_representations_for_AES_Block_Cipher>, 2016. [En ligne ; consulté en mars-2020].

[10] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. Data encryption standard (des). <https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf>, 1999. [En ligne ; consulté en mai-2020].

[11] Michael J. Wiener Paul C. van Oorschot. Parallel collision search with cryptanalytic applications. <https://people.scs.carleton.ca/~paulv/papers/JoC97.pdf>, 1996. [En ligne ; consulté le 08-Février-2020].

[12] Omar Reyad. Cryptography and data security : An introduction. <https://www.researchgate.net/publication/327388046_Cryptography_and_Data_Security_An_Introduction>, 2018. [En ligne ; consulté en mai-2020].

[13] Joëlle Roue. Analyse de la résistance des chiffrements par blocs aux attaques linéaires et différentielles. <https://hal.inria.fr/tel-01245102v2/document>, 2016. [En ligne ; consulté en mai-2020].

[14] Damien Vergnaud. *Exercices et problèmes de cryptographie - 3e édition.* InfoSup, Dunod, 2018.

[15] Marion VIDEAU. Critères de sécurité des algorithmes de chiffrement à clé secrète. <http://videau.lecte.com/media/MarionVideauPhD.pdf>, 2005. [En ligne ; consulté en mai-2020].