UNIVERSITÉ DE LIMOGES
FACULTÉ DES SCIENCES ET TECHNIQUES

MASTER 2 CRYPTIS

Attaques sur RC5 et MD5 en utilisant des chaînes arc-en-ciel

BORE Nicolas CHOUGAR Malik DE BONFILS LAVERNELLE Jean

VAREILLAUD Julien

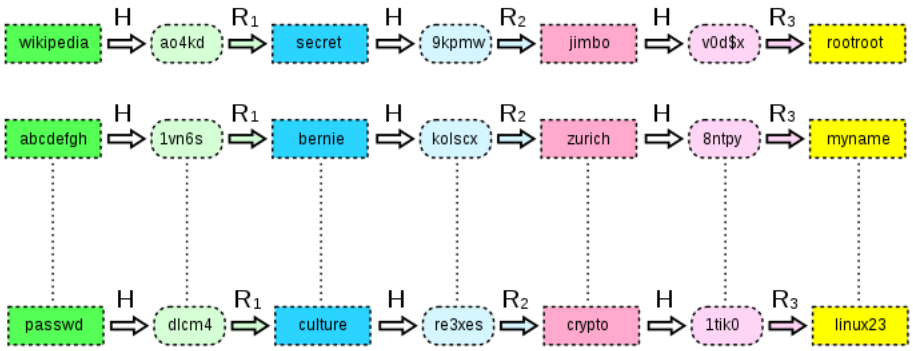
Janvier 2022

Contents

1 Principe théorique des tables arc-en-ciel

Il existe beaucoup de méthodes permettant d'inverser une fonction de hachage ou d'attaquer un système de chiffrement symétrique afin de trouver la clé de chiffrement à partir d'un clair connu. Parmi ces méthodes, nous avons les tables de hachage et les méthodes de recherches exhaustives (brute-force) qui ne sont en majorité pas intéressantes du point de vue complexité et temps d'exécution. Nous allons à travers ce projet nous intéresser à une méthode basée sur des *chaînes arc-en-ciel* ou *rainbow tables* découverte par *Philippe Oechslin* et qui représente une amélioration d'une méthode plus ancienne proposée par *Martin Hellman*. Le principe des *chaînes en arc-en-ciel* est basé l'utilisation de d'une fonction de hachage noté H et de plusieurs fonctions de réduction notées R_i *(i dépend de l'ordre d'application de la fonction de réduction)*. Cette technique permet notamment d'attaquer les bases de données qui stockent des hachés de mots de passe utilisateurs et permet donc de retrouver les mots de passe originaux avec une complexité meilleure que les techniques basées sur de la recherche exhaustive. La fonction de hachage va donc permettre d'obtenir une empreinte de taille fixe de l'entrée de la fonction. La fonction de réduction va quant à elle transformer le haché précédent en une donnée utilisable (un nouveau mot de passe). Les *chaînes arc-en-ciel* sont souvent stockées dans de gros fichiers de plusieurs téraoctets appelés *tables arc-en-ciel* ou *rainbow tables* contenant des centaines de milliers de chaînes ou dans des disques volumineux et sont utilisées pour attaquer les systèmes sécurisés stockant des hachés de mots de passe. Une *chaîne arc-en-ciel* contiendra l'application itérée d'une fonction de hachage notée H et de fonctions de réduction notée R_i *(i dépend de l'ordre d'application de la fonction de réduction)*. Une fois les *chaînes en arc-en-ciel* calculées, on pourra stocker que le début (mot de passe initial) et la fin (le haché final) de chaque chaîne. Généralement, la fonction de réduction consiste à coder en ***base64*** le hachage, puis à le tronquer à un certain nombre de caractères.

Ci-dessous, un schéma illustratif des *chaînes arc-en-ciel* :



Dans le cadre de ce projet, nous allons construire des *chaines arc-en-ciel* qui seront utilisées afin d'attaquer deux fonctions : une fonction de hachage et une fonction de chiffrement symétrique. Le choix des deux fonctions sera détaillé et argumenté.

1.1 Application sur les fonctions de hachage

Les fonctions de hachage sont des fonctions très utilisées dans le domaine de la cryptographie afin de faire de l'authentification et vérifier l'intégrité des données. Les fonctions de hachage sont associées à des propriétés cryptographiques très importantes : la *compression des données* et la *facilité de calcul*. Pour la compression de données, une fonction de hachage doit prendre en entrée une donnée x de taille aléatoire et retourner un haché ou empreinte *h(x)* de taille fixe. Pour la deuxième propriété relative à la facilité de calcul, une empreinte doit être obtenue en temps très rapide.
D'autres propriétés peuvent être évoquées et sont indispensables pour estimer l'optimalité d'une fonction de hachage :
Soit h : X → Y une fonction de hachage :
1 - h est dite à sens unique si, pour presque tout y de Y, il est calculatoirement infaisable de trouver x tel que y = h(x).
2 - h est dite faiblement sans collision si, pour un x donné, il est calculatoirement infaisable de trouver X_0 tel que h(X_0) = h(x).
3 - h est dite sans collision s'il est calculatoirement infaisable de trouver x et X_0 tels que h(x) = h(X_0).

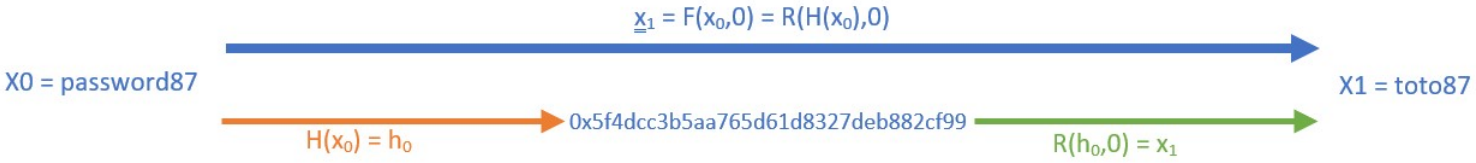
Il est donc clair que si une fonction de hachage ne vérifie pas toutes les propriétés citées ci-dessus, elle est donc considérée comme étant faible et on ne peut donc pas l'utiliser dans la création d'un logiciel cryptographique sûr.

1.1.1 Phase de pré-calcul

Pour générer une chaîne, à partir d'un mot de passe, on applique la fonction de hachage H suivi d'une réduction R de ce hash pour obtenir un nouveau mot de passe (dans l'espace de recherche) qui correspond donc au deuxième maillon de la chaîne. Pour générer un maillon x_{i+1} (un mot de passe) à partir du maillon précédent x_i, on applique donc la fonction suivante :

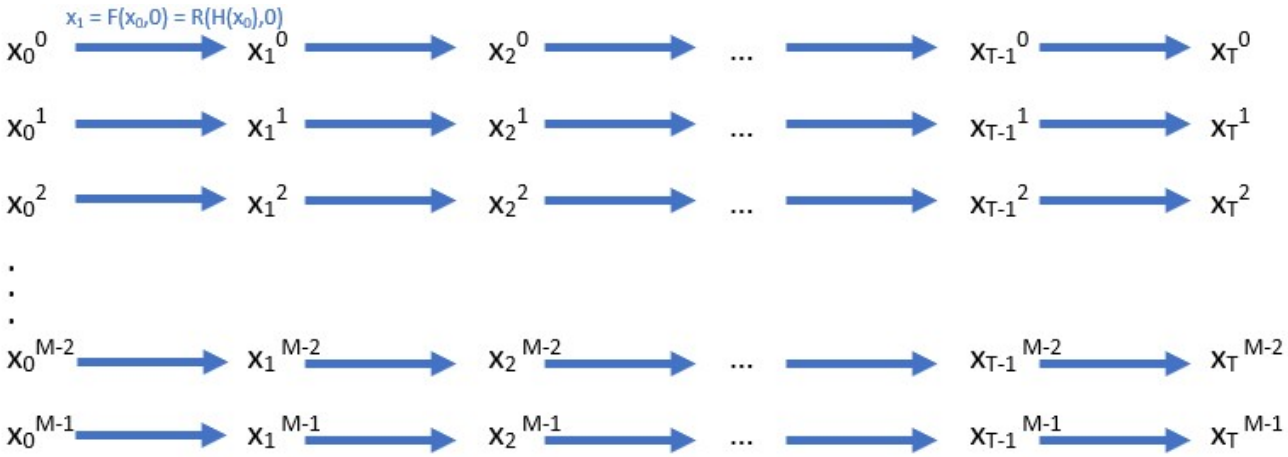
$$F(x_i, i) = (R \circ H)(x_i) = R(H(x_i), i)$$

Cette fonction représente simplement la composition de la fonction de hachage et de la fonction de réduction où i est l'indice du maillon dans la chaîne. *Cet indice est indispensable en entrée de la fonction de réduction pour éviter qu'un même hash en entrée donne un mot de passe identique en sortie et ainsi que cela crée des ramifications (voir fonction de réduction).*
Ci-dessous, un schéma expliquant comment générer un maillon, autrement dit, générer un mot de passe x_1 à partir d'un mot de passe x_0 :



Une table arc-en-ciel se caractérise par le paramètre **M** représentant le nombre de chaînes contenues dans la table arc-en-ciel et par le paramètre **T** correspondant au nombre de maillons d'une chaîne où chaque maillon est obtenu en appliquant la fonction F sur le maillon précédent. Une chaîne de la table arc-en-ciel peut donc être modélisée par le schéma suivant:

Une table arc-en-ciel de taille G **avec** $G = MXT$ et **T** représentant le nombre de maillons par chaîne et **M** représentant le nombre de chaînes dans la table est similaire à une matrice de **T** colonnes et **M** lignes.

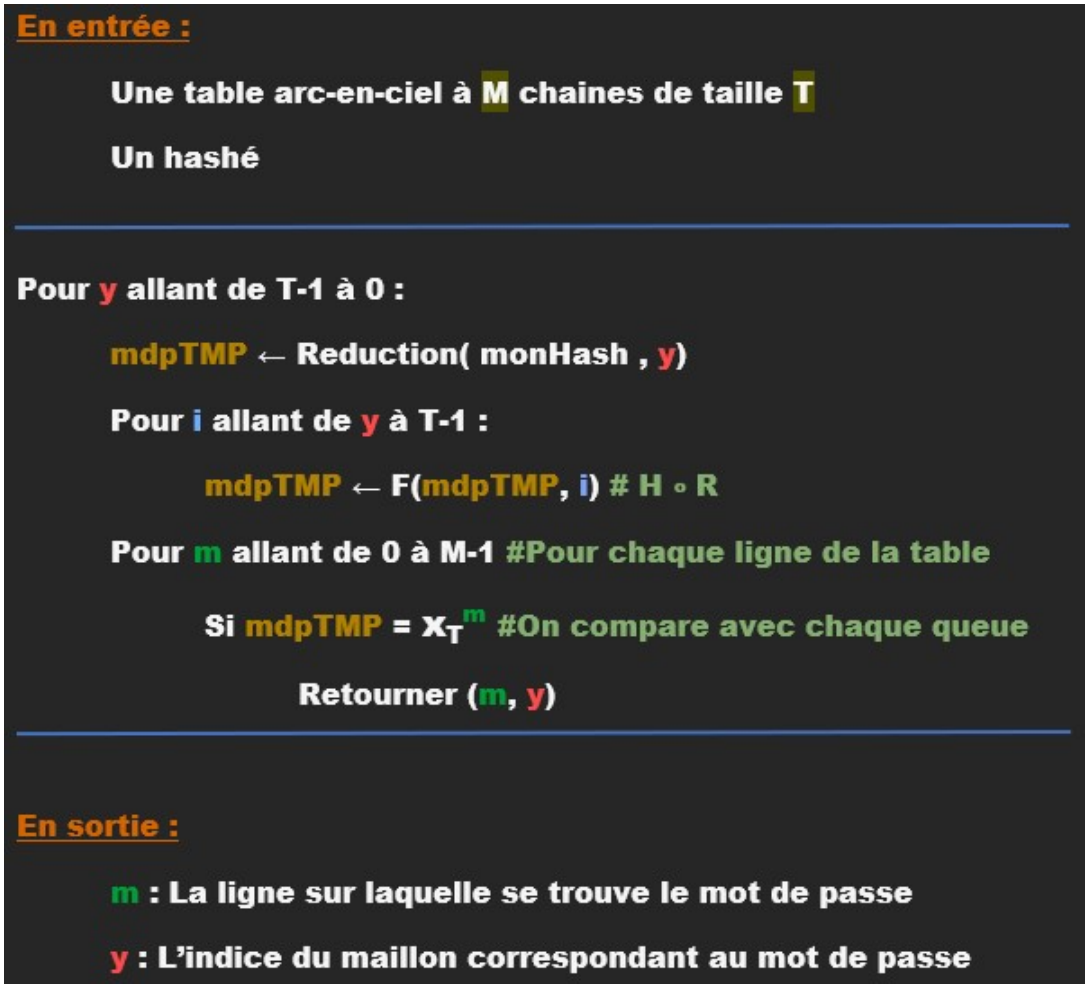


Cependant, en pratique, puisque les maillons intermédiaires ne sont pas indispensables à la phase de recherche du hash, **seules les têtes x_0 et les queues x_{T-1} des chaînes sont stockées dans la table afin d'économiser l'espace de stockage.** Cette étape, qui représente la phase de pré-calcul, est réalisée uniquement une seule fois. Elle génère une table qui permettra de trouver le mot de passe correspondant à un hash donné d'une manière plus rapide que la recherche exhaustive puisque les hashs auront été préalablement calculés.

1.1.2 Phase de recherche du hash

A partir de la table arc-en-ciel générée durant la phase de *pré-calcul*, il va être possible de retrouver, à partir d'un hash donné noté y, un mot de passe x tel que $h(x) = y$. Cette partie se décompose en *deux principales phases*:
La *première phase* consiste à rechercher dans la table si une chaîne contient un tel mot de passe. Si on applique la fonction de hachage sur ce mot de passe, on obtient le haché en entrée. Si un tel mot de passe existe, la première phase permet de connaître **l'indice** de la chaîne ainsi que **l'indice** du maillon sur cette chaîne **correspondant au mot de passe engendrant le haché en entrée.**

Ci-dessous, l'algorithme permettant cela :



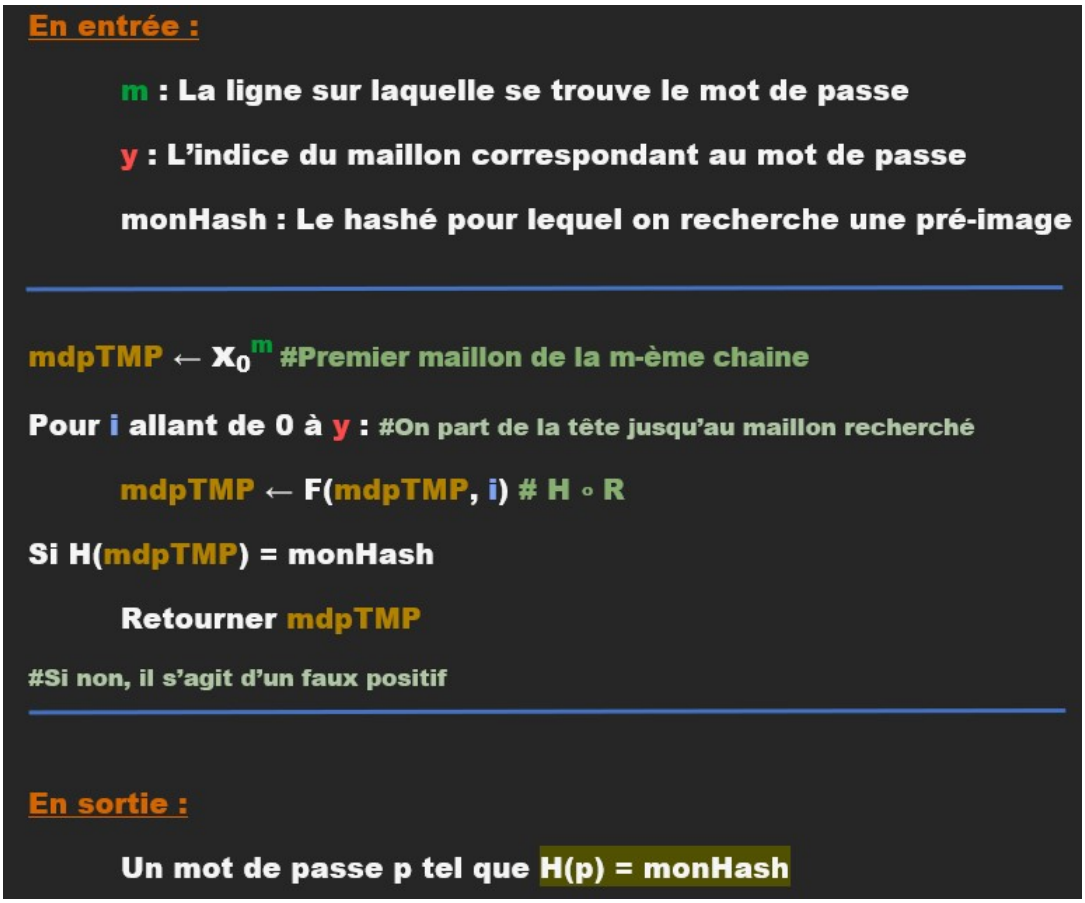
Rappel: Pour un même haché en entrée mais avec des indices de maillon différents, la fonction de réduction donne deux résultats différents. Cela permet d'éviter de créer des ramifications (voir fonction de réduction).

Durant la phase de recherche, le nombre de fois où la fonction $F(x_i, i) = (R \circ H)(x_i)$ doit être appliquée pour parcourir une chaîne de taille **t** dans son entièreté est donc défini par la formule suivante :

$$\sum_{k=1}^{t-1} k = \frac{t(t-1)}{2} = 1 + 2 + 3 + ... + (t-1)$$

En plus de cela, il faut appliquer au total **t** fois la fonction de réduction en plus, puisque, à chaque itération, il est nécessaire de se placer sur le mot de passe engendré par le haché donné.

La *deuxième partie* de la phase de recherche consiste globalement à repartir de la tête de la chaîne et à appliquer la série de hachage et de réduction jusqu'à arriver au maillon x_y^m censé produire le résultat $H(x_y^m) = monHash$, avec y **l'indice** du maillon sur la m-ème chaîne (y et m étant obtenus durant la première partie de la phase de recherche).



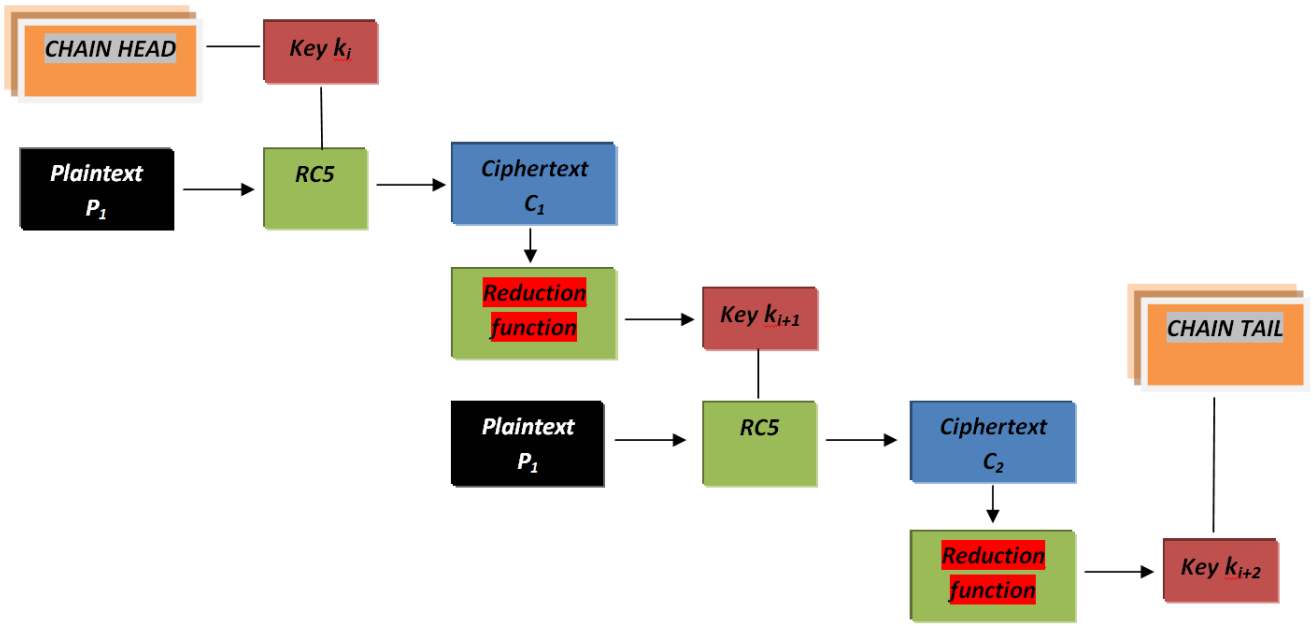
Cependant,des faux positifs peuvent subvenir. En effet, puisque la fonction de réduction n'est pas une simple bijection et, comme son nom l'indique, elle réduit un haché de taille n en un message de taille inférieure à n. Fatalement, des collisions existent et on peut donc avoir deux hachés différents *H1* et *H2*, tels que $R(H1, y) = R(H2, i)$ avec $i \neq y$. Cependant, on peut tout même éviter cette solution, d'où l'importance d'une bonne fonction de réduction.

Enfin, on constate que la première partie de la phase de recherche nécessite seulement les queues de chaînes tandis que la deuxième partie nécessite, quant à elle, seulement les têtes de chaînes (le premier maillon de chaque chaîne). Il est donc bien inutile de stocker les maillons intermédiaires dans la table arc-en-ciel.

1.2 Application sur les fonctions de chiffrement

Dans le cas d'un schéma de chiffrement symétrique, le but est de trouver la clé de chiffrement. Dans la phase de pré-calcul *(off-line)*, on va créer des chaînes de chiffrement dans lesquelles chaque chiffrement est celui d'un clair fixé à l'avance associé à une clé K_i (i dépend de l'itération de la chaîne) de chiffrement qui dépend de l'itération de la chaîne. Pour chaque chaîne de la *rainbow table*, il suffit donc de ne mémoriser que la clé initiale et la clé finale.
Dans la phase d'attaque, on commence à partir d'un chiffré correspondant à un clair connu et ayant servi à la création des chaînes. A partir de là, il faut extraire la clé de chiffrement et on va suivre le même processus pour calculer une nouvelle chaîne tout en essayant de trouver à chaque itération la valeur de la clé intermédiaire obtenue parmi les valeurs finales de l'ensemble des chaînes pré-calculées. Si on trouve cette clé intermédiaire, il faut repartir du début de la chaîne ayant cette valeur finale jusqu'à l'obtention du chiffré désiré. La clé qu'on recherche sera donc celle qui a permis de produire le chiffré désiré.

Théoriquement, les chaines arc-en-ciel qui seront utilisées pour cryptanalyser la fonction *RC5* auront la propriété d'être composée que de la *clé initiale (Chain Head)* et la *clé finale (Chain Tail)*. A partir d'un texte en clair P_1 et une clé de chiffrement K_i (où i dépend de l'étape de l'application de la fonction RC5), on applique la fonction RC5 afin d'obtenir un texte chiffré C_1. Contrairement à la fonction *MD5* où la réduction est appliquée sur un haché pour produire un nouveau mot de passe, dans le cas de *RC5*, la fonction de réduction est appliquer à un chiffré obtenu à l'étape i pour produire une nouvelle clé qui sera utilisée par la fonction *RC5* à l'étape $i + 1$. Ce processus est répété t fois, t étant la taille de la chaîne arc-en-ciel.



1.3 Apport et compromis temps-mémoire

L'inversion d'une fonction de hachage consiste à partir d'un haché *h(x)* à retrouver la donnée en entrée *x*. Dans le cas d'un schéma de chiffrement symétrique, les *chaînes arc-en-ciel* permettent pour un clair connu à l'avance d'extraire la clé de chiffrement et donc de cryptanalyser le système. La méthode des *chaînes arc-en-ciel* comportent plusieurs phases. Dans un premier temps, il faut faire une phase de pré-calcul *(off-line)*, calculatoirement coûteuse, cette phase doit être réalisée qu'une et une seule fois et va servir à établir un grand volume de données qui sera utilisé par la suite dans la phase d'attaque *(on-line)*. Étant donné un espace mémoire composé de N éléments, les *chaînes arc-en-ciel* permettent d'obtenir un compromis temps-mémoire de complexité $N^{2/3}$, ce qui est nettement meilleur comparé aux autres méthodes dites de recherches exhaustives. L'un des gros avantages des *rainbow tables* est qu'on peut choisir le nombre de liens entre le mot de passe initial et le haché final de chaque chaîne, ce qui correspond au nombre d'applications itérées de la fonction de hachage et des fonctions de réduction. Plus, il y a de liens entre le mot de passe initial et le haché final, plus il y a un grand nombre de mots de passe capturés. Cependant, la méthode présente aussi quelques désavantages. En effet, on ne peut pas choisir à l'avance les mots de passe capturés à l'intérieur des chaînes, ce qui peut nous amener à rencontrer un nombre restreint de mots de passe courants. Il existe

également un autre désavantage lié à la récupération du mot de passe, plus les chaînes sont longues, plus il faut d'opérations et donc de temps pour trouver le mot de passe correspondant au haché en entrée.

La table ne stockant seulement les **têtes** et les **queues de chaînes**, la taille de chaque chaîne **T** (le nombre de maillons) n'aura aucune influence sur la taille du fichier en terme de stockage. En effet, cela dépendra uniquement du paramètre M correspondant au nombre de chaînes et donc de lignes. C'est ici que le compromis temps-mémoire se situe, pour deux tables de taille $G = MXT$ de même ordre de grandeur, si l'on choisi un paramètre **M** plus élevé au profit de peu de maillons (T), alors la taille du fichier contenant la table sera plus lourde et le bénéfice de la méthode des chaînes arc-en-ciel impacté (Si $T = 1$, cela revient à une recherche du haché dans un dictionnaire contenant des mots de passe générés aléatoirement). Cependant, la phase de recherche du haché sera plus rapide. A l'inverse, si le nombre de maillons **T** est plus élevé devant **M**, toujours pour une table de taille **G** de même ordre de grandeur, alors le poids de la table sera naturellement réduit, cependant la phase de recherche du haché sera plus longue. *(Voir Analyses des résultats)*5

Pour résumer, le paramètre **M** impacte directement le poids de la table, tandis que le paramètre **T** impacte, quant à lui, directement la phase de recherche (cependant les deux phases ne sont pas impactées de la même façon). La phase de pré-calcul n'étant réalisée seulement une seule fois, contrairement à la phase de recherche, il s'agit de trouver le bon compromis, temps-mémoire donc, entre **M** et **T**. En effet, il s'agit bien d'un compromis temps-mémoire, puisqu'en augmentant le nombre de chaînes **M** au profil d'une réduction de la taille des chaînes **T**, on tend vers une recherche arbitraire si l'on considère que les têtes de chaînes sont prises aléatoirement. A l'inverse, si l'on choisit une taille de chaînes trop grande, au profil d'un faible nombre de chaînes, alors on tendra également vers une recherche arbitraire d'un haché puisque la fonction de réduction est censée produire un résultat "aléatoire" dans l'espace de recherche.

Étant donné G, avec $G = TXM$:

- Dans le cas où $T = 1$ (et donc $M = G$), la mémoire sera exploitée au maximum, de même que la phase de pré-calcul qui ne se résume donc plus qu'à la création d'un dictionnaire liant des mots de passe et leurs hachés. La phase de recherche, quant à elle, se résume à une simple recherche dans ce dictionnaire. On maximise donc l'espace mémoire nécessaire mais on minimise le temps de calcul de la phase de recherche.

- Dans le cas où $M = 1$ (et donc $T = G$), la mémoire est exploitée au minimum, la phase de pré-calcul est au moins aussi longue que dans le cas précédent, et le temps de calcul de la phase de recherche explose. On minimise donc l'espace mémoire nécessaire mais on augmente drastiquement le temps de calcul de la phase de recherche.

2 Choix d'implémentations

Dans le cadre de ce projet, nous avons choisi d'appliquer la méthode des *chaînes en arc-en-ciel* sur la fonction de hachage **MD5 - Message Digest 5**, créée par *Ronald Rivest* en 1991. Elle est aujourd'hui encore très utilisée mais reconnue comme étant faible d'un point de vue cryptographique. La fonction permet de calculer l'empreinte d'un fichier en produisant une sortie de 128 bits. La fonction a été cassée par des chercheurs chinois et il a été démontré que la fonction présente des collisions, ce qui signifie qu'il est calculatoirement faisable de trouver deux entrées x et x_0 telles que MD5(x) = MD5(x_0). Le fait qu'elle présente des collisions la rend vulnérable et donc inutilisable pour la conception d'un protocole cryptographique sécurisé. En revanche, elle reste toujours utilisée pour la vérification de l'intégrité de fichiers téléchargés (md5sum).
Nous avons choisi la fonction *MD5* car elle est facilement implémentable et très bien documentée sur Internet.

Pour la fonction de chiffrement que l'on va cryptanalyser en utilisant la méthode des *chaînes en arc-en-ciel*, nous avons choisi la fonction **RC5**, créée également par *Ronald Rivest* en 1994. La fonction est basée sur un chiffrement symétrique par bloc en utilisant des clés de chiffrement dont la taille varie de 40 à 2040 bits. Elle prend en entrée des blocs de taille variable de 32, 64 ou 1024 bits. Elle est également basée sur un mécanisme de tours qui peuvent varier de 1 à 255 tours. Nous avons choisi **RC5** car la fonction est très similaire à celui de la fonction **MD5**. En revanche, **RC5** est aussi reconnue pour être vulnérable pour un nombre de tours inférieur ou égal à 12 *(vulnérabilité avérée pour une attaque différentielle utilisant 2^{44} textes clairs)*. Il est donc conseillé d'utiliser un nombre de tours situé entre 18 et 20 tours pour avoir la sécurité nécessaire permettant de contrer les attaques différentielles.

Le langage de programmation qui a été choisi pour implémenter nos deux fonctions de hachage et de chiffrement symétrique est le langage de programmation C. Nous détaillerons dans la section *2.1* l'implémentation de la fonction **MD5** et dans la section *2.2* l'implémentation de la fonction **RC5**.

2.1 Implémentation de MD5

Deux versions de la fonction de hachage **MD5** ont été implémentées. La première étant disponible dans le dossier *Implémentation-Haut-Niveau-MD5* et la deuxième dans le dossier *Implémentation-Bas-Niveau-MD5* de l'archive du projet.
La première version étant moins efficace que la deuxième d'un point de vue complexité et temps d'exécution, nous utiliserons donc dans le cadre de ce projet la deuxième version de la fonction **MD5** *Implémentation-Bas-Niveau-MD5*. C'est donc cette version que nous présenterons en détail dans cette partie. Dans la première version de *MD5*, nous avons manipuler la plus petite unité (bit) alors que dans la deuxième version, nous avons utilisé des octets.

Dans notre deuxième version de l'implémentation de *MD5*, nous avons fait le choix de traiter toutes les opérations binaires et hexadécimales bit à bit. En comparaison avec un tableau de n octets censés représentés n bits et l'implémentation de fonctions et d'opérations permettant de gérer ce tableau, l'implémentation utilisée sur la deuxième version permet un gain de temps d'exécution que l'on a constaté après plusieurs exécutions et en comparant les temps d'exécution des deux versions sur les mêmes entrées. Pour un calcul de 2000 hachés, le temps d'exécution de la première version a été estimé à 0.857324 secondes (contre 0.03 secondes pour le deuxième version - ratio x28). Cela s'explique par la taille de la représentation des données (huit fois plus petite dans la deuxième version) et des fonctions natives en C permettant d'effectuer des opérations bit à bit de manière aisée pour n'importe quelle machine. Les opérations sur des tableaux de plus grande taille étant plus coûteuses en terme de mémoire et de temps d'exécution suffisent à expliquer la grande différence d'efficacité entre les deux versions.
Dans cette deuxième version de la fonction *MD5*, nous avons essayé également de rendre le code le plus compact et le plus lisible possible via l'utilisation d'un Header regroupant les prototypes de fonctions, des noms de variables et de fonctions explicites et une organisation générale du code permettant une application spécifique d'une fonction à une tâche dédiée. Vous pourrez retrouver dans les fichiers *md5bis* toutes les fonctions propres au fonctionnement de *MD5*, c'est à dire toutes les étapes successives de l'algorithme. Dans *fonctionsGen*, vous trouverez des fonctions plus génériques qui sont utilisées dans les étapes de l'algorithme *MD5* mais qui pourraient être utilisées à d'autres fins.

Pour cette version, nous avons pris le parti d'exclure tous les mots de passe en clair excédant une taille de 56 octets. *Pourquoi 56 octets ?* Car cela correspond dans la première étape de *MD5* à la taille réservée au clair, à laquelle viennent s'ajouter 8 octets de padding. Cette étape s'appelle la **complétion**. Son mécanisme est le suivant : La complétion s'attache à traiter des blocs de 64 octets, soit 512 bits. Sur ces 64 octets, 56 sont réservés au mot de passe en clair. Si celui-ci dépasse 56 octets alors on recrée un second bloc, ce que nous excluons volontairement dans notre implémentation. Si en revanche le mot de passe est d'une taille inférieure à 56 octets, on le complète par 0x800... et autant de 0 nécessaire jusqu'à atteindre 56 octets. Ce dernier cas sera omniprésent dans notre cas de figure. Les 8 derniers octets correspondent enfin à la taille du clair en hexadécimal little-endian. Cela permet d'avoir des mots de passe allant jusqu'à 2^{64} octets (!). Vous pourrez retrouver cette étape dans la fonction *psswdto512* du fichier *md5bis*. Pour la suite de l'algorithme, on s'attachera à traiter des blocs de 32 bits, soit 4 octets, ce qui correspond à une division par 16 des données issues de l'étape de complétion.
Pour les structures de données, nous avons choisi d'effectuer tous les calculs binaires à l'aide du type *uint32t* de la bibliothèque *stdint.h*. Ce type correspond parfaitement à la taille des constantes de base de *MD5* ainsi qu'au traitement des 16 blocs de 32 bits après complétion. Le mot de passe de départ est considéré comme étant un *unsigned char ** et le haché en sortie est également reconstitué avec ce même type. Des fonctions natives en C permettent d'effectuer rapidement la traduction de l'un à l'autre. Vous pourrez les retrouver dans le fichier *fonctionsGen.c* du code.

Le fonctionnement de *MD5* est cyclique et s'appuie sur des constantes qui permettent de garantir son déterminisme. La seule variable de l'algorithme est le bloc de 512 bits. Nous avons donc défini les quatres constantes hexadéximales dans le header de *md5bis* qui seront ensuite attribuées aux variables 0xaa, 0xbb, 0xcc et 0xdd. Ces quatres buffers initiaux de 32 bits chacun sont des constantes car leur valeur est la même à chaque fois. Cependant, il est impératif de créer des variables car ce sont ces 4 blocs de 32 bits,

modifiés à chaque tour dans les étapes suivantes, qui vont constitués par réassemblage les 128 bits de sortie de *MD5*. Nous avons également défini les structures qui sont des tableaux rassemblant des données qui pourront être utilisées à chaque cycle de md5. Ces tableaux restent les mêmes peu importe les 512 bits en entrée. Il convient enfin de définir les expressions régulières de rotation et plus généralement les opérations binaires. Ces dernières se trouvent dans le header de *md5bis*. Une fois cela achevé, nous avons établi toutes les constantes nécessaires au fonctionnement de *MD5* ainsi que la fonction de complétion du clair à hacher. Suivant le déroulement de *MD5*, il faut ensuite diviser le bloc de 512 bits en 16 blocs de 32 bits et effectuer 4 tours d'opérations faisant appel aux constantes de départ, aux expressions régulières et aux structures définies. Chaque tour sera constitué d'une boucle de 16 itérations durant lesquelles sont effectuées des rotations et des permutations des buffers constants initiaux regA, regB, regC et regD avec les données des blocs du clair initial et de ses successives modifications.

Enfin, ces étapes achevées, il convient de recueillir les 4 buffers initiaux ainsi modifiés et des les assembler. Nous avons donc 4 blocs de 32 bits qui fournissent après assemblage un bloc de 128 bits correspondant au haché de sortie de la fonction *MD5*. Nous retrouvons l'enchaînement de ces différentes étapes dans la fonction éponyme de *MD5* du fichier *md5bis* que nous avons voulu clair et concise.

2.2 Implémentation de RC5

Afin d'implémenter *RC5*, nous avons repris la même idée que pour *MD5*. Diviser le code en étapes successives afin de rendre le code plus lisible et la compréhension du fonctionnement de l'algorithme plus aisée. Cependant, nous avons rencontré quelques difficultés lors de l'implémentation. La principale difficulté résidait dans le fait de traiter des blocs en clair de taille variables. Pour être plus précis, *MD5* traite des clairs de taille variable également, mais il intègre un mécanisme de division par blocs et de complétion qui permet de manipuler à l'arrivée des blocs de taille identique pour n'importe quelle entrée. L'espace alloué au mot de passe en clair est de 56 bits à chaque fois. Dans le choix des structures de données, nous nous étions donc concentrés sur le traitement de blocs de 32 bits. Le type *uint32t* convient dans ce cas parfaitement pour ce type de calculs. En revanche, pour *RC5*, les tailles en entrée peuvent variées et donc une première idée a été d'adapter notre code à 3 tailles différentes en utilisant successivement *uint16t*, *uint32t* et *uint64t*. Afin de gérer tous les cas, le code s'est avéré être très long, remplis de conditions et de foncions répétitives aux appels conditionnés par un unique paramètre qui représente la taille du bloc d'entrée. Cela nous a semblé lourd 'un point de vue complexité, temps d'exécution et allocation mémoire. Nous avons donc exploré l'idée offerte par le langage C de faire nos calculs par généricité de type. Cela est possible par l'utilisation du type *void **. Mais après un certain temps à appréhender son fonctionnement et explorer ses possibilités, nous nous sommes confrontés aux mêmes problèmes. En effet, après s'être documentés sur Internet, nous avons constaté que l'utilisation du type *void ** est intéressante pour certains cas de figure mais n'est que très rarement recommandée. Il nous fallait réaliser un *cast* sur toutes nos données d'un type à l'autre avant de les soumettre à la fonction. De plus, un tel fonctionnement n'est pas compatible avec les expressions régulières. Nous avons donc, après de nombreuses tentatives sans succès, décidé d'opter simplement pour la structure *unsigned char **, qui nous paraissait au début peu adapté, mais qui s'est avérée être la plus pratique.

Il faut noter que toutes sur toutes les implémentations de *RC5* disponibles sur Internet, le type utilisé est un *unsigned long **. Cependant, le cas traité est celui d'une entrée de 32 bits, donc pas adapté à des variations.

Nous pouvons diviser *RC5* en *deux étapes* dont la première est divisée en plusieurs étapes. La première étape est la configuration de toutes nos données variables et constantes en fonction des paramètres fournis. Ainsi, dans la fonction *RC5*, vous retrouverez tout un processus d'initialisation qui va dans un premier temps récupérer la clé et le clair, calculer leur taille et ainsi décider de la taille des tableaux à allouer, des variables de taille importantes et des constantes à définir. La seconde étape consiste à utiliser ces données ainsi configurées pour effectuer des calculs avec le clair. Toutes ces fonctions sont à retrouver dans le fichier *rc5.c* et leurs prototypes dans le fichier *rc5.h*.

La première sous-étape de la première partie est de définir les deux constantes P et Q en fonction de la taille d'entrée. Nous aurons donc une suite hexadécimale de 16, 32 ou 64 bits qui dépend de la taille de l'entrée.

Ensuite, nous devons créer un tableau L en fonction de la clé fournie, de sa taille et de la taille du mot en clair afin de créer des "mots" de la taille équivalente au clair d'entrée. Dans la boucle à l'intérieur de la fonction, nous aurons besoin du paramètre *rounds* que nous avons décidé de fixer en fonction de la taille du mot. Certains lui attribuent une constante, nous avons opté pour ce choix, l'important étant que l'algorithme soit capable de retrouver cette valeur pour qu'elle soit la même à la fois pour le déchiffrement et le chiffrement. Dans le cas contraire, il nous serait impossible de déchiffrer et de retrouver notre clair initial. Les calculs peuvent ainsi s'effectuer sur des tableaux d'*unsigned char* de même taille. Cette étape nous permet également de calculer les variables c et u. La variable c est égale au quotient b/u avec b taille de la clé en bits et $u = w/8$ avec$w$ taille du mots en clair en bits. Nous récupérons ces valeurs que nous utiliserons par la suite via des pointeurs en paramètre de fonctions.
La troisième étape consiste à créer une sous clé S, de la forme *unsigned char ** à l'aide des constantes P et Q.

Pour la quatrième et dernière étape de cette phase d'initialisation, nous utilisons les tableaux S et L afin de créer une clé secrète. On utilisera pour cela les paramètres t et c calculés durant les étapes précédentes. La clé secrète sera contenue dans S.

Après ces étapes d'initialisation, nous pouvons soit chiffrer, soit déchiffrer. L'initialisation est la même selon les deux cas. Évidemment, l'étape de déchiffrement consistera à effectuer les opérations de chiffrement dans le sens inverse afin de retrouver le clair initial. Ainsi, pour le chiffrement, nous créons deux registres A et B qui contiendront les blocs du clair une fois divisé. Le nombre d'itérations est défini par la variable rounds que nous avons évoquée plus haut. Pour chacune d'entre elles, on effectue des décalages à gauche et une addition avec la valeur i de la clé secrète. Pour le déchiffrement, nous retirons la valeur i et nous effectuons le même nombre de décalage à droite.

2.2.1 Difficultés rencontrées

Cette partie nous permet de faire un point, à quelques jours du rendu, sur notre production aux regards des attentes. Nous avons passé énormément de temps à essayer de rendre un programme fonctionnel mais nous n'avons pas été en mesure de concrétiser la dernière et essentielle étape, à savoir retrouver un clair voulu à partir d'un chiffré connaissant une clé. Cette erreur ne nous permet pas d'intégrer notre code à la rainbow table. Pour être plus précis, nous sommes passés par plusieurs étapes qui ont été riches en enseignements sur la manière dont est gérée la mémoire en C et comment mener un debuggage.

Tout d'abord, nous avons voulu créer des structures qui nous permettaient de regrouper les paramètres en fonction de leur utilité dans le code. Ainsi, nous avons *keyUser, plainText et parameters*. Notre première erreur, qui nous a coûté beaucoup de temps, a été de déclarer les trois structures à la suite dans notre fonction *RC5* et d'allouer chacun de leurs paramètres après ces trois déclarations. En procédant ainsi, les zones mémoires allouées pour les structures étaient contiguës et donc lorsque l'on écrivait des valeurs dans certains éléments de ces structures, d'autres éléments d'autres structures se trouvaient modifiées. C'est en scrutant les zones mémoires à l'aide de *gdb* que nous nous sommes rendus compte de ce problème. Nous avons donc dû déclarer chaque structure et allouer chacun de ses paramètres à la suite.

Une autre erreur a été pour l'utilisation des *sscanf*. La fonction *sscanf(char * txt, "%x", uint32_t val)* permet de copier la valeur hexadécimale contenue dans une chaîne de caractère dans un *uint 16, 32 ou 64*. Sauf que lors dans son utilisation, certaines valeurs étaient modifiées à des endroits du programme dans lesquelles nous ne manipulions pas du tout cette zone mémoire / variable. Après de multiples vérifications sur les zones mémoires en question, nous avons donc commencé à penser que le problème venait de l'utilisation de la fonction. En effet, les chaînes de caractères, en C, se terminent toutes par le caractère `"\0"`, correspondant à la valeur 0 de la table ASCII, NULL. Ce marqueur permet de reconnaître la fin d'une chaîne de caractères. Par conséquent, quand on faisait un *sscanf* pour attribuer à une zone mémoire le contenu d'une chaîne de caractère, cette zone était remplie par les caractères mais ne prenait pas en considération le caractère de fin de chaîne qui lui était copié dans l'adresse mémoire suivante. Dans notre cas, lorsque nous modifions *a16* avec *sscanf*, alors b16 recevait la valeur 0 et lorsque nous voulions l'afficher sous forme de *uint*, le programme nous retournait 0, alors que nous n'avions pas touché à *b16* dans notre code.

Une autre erreur a été, dans la première étape, celle de l'attribution de valeurs constantes à p et q sous forme hexadécimales directement. En procédant ainsi, la chaîne de caractères ne correspond pas à une valeur hexadécimale mais aux caractères correspondant, dans la table ASCII, à ces valeurs hexadécimales. *Pourquoi est-ce dérangeant ?* Parce que pour utiliser les fonctions *fprintf* et *fscanf* et passer du type *unsigned char ** au type *uint*, il faut que la chaîne de caractères représente une chaîne hexadécimale, sinon cela nécessite à chaque fois de passer par une chaîne de caractères intermédiaires qui va transformer l'ASCII en hexadécimal. De plus, la chaîne de caractère représentant la chaîne hexadécimale est de taille deux fois plus grande qu'une chaîne de caractères normale, étant donné que l'hexadécimal se représente par un digit pour un octet. Pour l'exemple, si on déclare chaine = 0x73616c7574, la valeur de la chaine sera "salut", il faut dans notre cas le déclarer comme suit : chaine = "73616c7574".
Ces problèmes de représentation/manipulation des données ont été récurrents et très difficile à corriger. Nous avons ainsi appris à utiliser *gdb* et ses différentes options qui permettent de gérer le déroulement d'une exécution et débugger en temps réel le code. Arrivés au point où tous les accès mémoires étaient corrects, les résultats nous semblaient quant à eux cohérents. Pour la première fois, à ce stade, l'algorithme était déterministe. Pour des paramètres d'entrée similaires, nous obtenons les mêmes sorties, que ce soit pour le chiffrement ou le déchiffrement. Nous étions à ce moment-là à dix jours du rendu, après des jours et des jours de déboggage. Nous avons testé notre code, en respectant à la ligne l'algorithme de *MD5*. Cependant, il n'existe aucune implémentation de *RC5* dans openssl, ni aucune sur Internet qui respecte la généricité que

nous avons voulu mettre en place . Donc aucun moyen de tester le bon fonctionnement de notre algorithme en comparant des sorties. De plus, et là était le plus gros problème, le chiffré une fois passé par la fonction de déchiffrement, ne permettait pas de retrouver le clair. Nous avons donc scruté toutes les étapes, veillé à ce que les calculs soient effectués selon les mêmes bases, le même ordre et vérifié les initialisations... Et nous n'avons trouvé aucune ligne de code qui ne soit pas fidèle à la description de l'algorithme *RC5* original. Comme le problème ne venait pas de problèmes techniques comme l'accès mémoire ou des poblèmes de transcription entre structures de données, nous n'avons pas été en mesure de faire fonctionner notre code, à notre plus grand regret, car cela nous prive de son utilisation dans la rainbow table également.

2.3 Fonction de réduction et espace de recherche

Les têtes de chaînes sont générées aléatoirement. Le programme prend en entrée les paramètres définissants l'espace de recherche **qui correspond à l'ensemble des valeurs que peut prendre un mot de passe**. Par exemple, l'ensemble des "mots" en minuscules entre 5 et 10 caractères. En effet, en pratique on connaît souvent le format du mot de passe et réduire l'espace de recherche permet ainsi de maximiser ses chances de trouver un mot de passe. Ces paramètres sont également indispensables à la fonction de réduction pour générer des mots de passe se trouvant dans ce même espace de recherche. Contrairement à la fonction générant une tête de chaîne et utilisant de l'aléa, la fonction de réduction doit être déterministe puisqu'elle est utilisée lors de la phase de génération de la table arc-en-ciel mais également lors de la phase de recherche. On rappelle que **R()** représente la fonction de réduction, **H()** une fonction de hachage et x_i^n le maillon ayant l'indice **i** et se trouvant sur la **n-ème** chaîne de la table.
En plus d'un hash en entrée la fonction de réduction prend l'indice du maillon courant. Ce paramètre supplémentaire est indispensable à une bonne fonction de réduction et permet d'éviter les ramifications des chaînes dans la table arc-en-ciel. En effet, puisque la fonction est pseudo-aléatoire, pour une même entrée, la fonction de réduction produit la même sortie. Or, il serait très probable pour une table arc-en-ciel de grande taille d'avoir deux maillons égaux $x_y^m = x_i^n$ et donc $R(H(x_y^m))$ produirait le même résultat que $R(H(x_i^n))$ et ainsi de suite $R(H(R(H(x_i^n))))$ donnerai le même résultat que $R(H(R(H(x_y^m))))$ causant alors une sorte de "ramification" des chaînes m et n ainsi qu'une succession de doublons, une perte de temps de calcul et d'espace. L'utilisation de l'indice du maillon en entrée de la fonction de réduction permet de prévenir cela. $R(x_y^m, y)$ donnera un résultat différent de $R(x_i^n, i)$ ce qui évitera la ramification des deux chaînes. Cependant, bien que cela soit peu probable, cette méthode n'empêche pas une ramification de chaînes dans le cas où deux maillons sur deux chaînes différentes possèdent la même valeur et porte le même indice i sur leurs chaînes respectives: $x_i^m = x_i^n \Rightarrow R(x_i^n, i) = R(x_i^m, i)$.

Dans le cas où $R(x_y^m, y) = R(x_i^n, i), i \neq y$ (pour des indices i et y différents), la ramification ne sera que temporaire puisque $R(x_{y+1}^m, y+1)$ donnera très probablement un résultat différent de $R(x_{i+1}^n, i+1)$. Cela permet donc bien d'éviter les ramifications causant seulement l'apparition d'un seul doublon. Idéalement, on voudrait qu'en sortie de la fonction de réduction les messages générés soit parfaitement répartis dans l'espace de recherche pour minimiser les chances d'apparitions de doublons. Cette propriété définie, en quelque sorte, la qualité de la fonction de réduction. Puisque des doublons peuvent avoir lieu des faux positifs peuvent subvenir. En effet, lors de la phase de recherche, il est possible d'avoir le cas où, pour $x_y^m \neq x_{T-1}^n$ et donc $H(x_y^m) \neq H(x_{T-1}^n)$, il est possible que $R(H(x_y^m), y) = R(H(x_{T-1}^n), T-1) = x_T^n$ (où T est le nombre de maillons composant les chaînes et donc x_T^n est une queue de chaîne). Le hash dérivé correspondant bien à une queue de chaîne, le programme va passer à la deuxième partie de la phase de recherche et repartir depuis le début de la chaîne n pour retrouver le mot de passe donnant le hash. Cependant puisque $x_y^m \neq x_{T-1}^n$ et donc $H(x_y^m) \neq H(x_{T-1}^n)$ le mot de passe ne sera pas dans la chaîne, il s'agira donc d'**un faux positif**. Dans ce cas-là, le programme doit reprendre la première partie de la phase de recherche là où il s'était arrêté.

Ces problèmes de ramifications sont liés à la fonction de réduction qui est une fonction surjective et qui, comme son nom l'indique, réduit un message de taille t en un message de taille inférieure à t. Inévitablement des collisions peuvent se produire et ainsi les doublons sont inévitables. Dans un cas concret où l'espace de recherche est l'ensemble des mots de passe de 10 caractères en minuscules et la fonction de hashage utilisée est MD5. Cette dernière produisant une sortie sur 128 bits, la fonction de réduction a donc $2^128 \approx 3.4028237e{+}38$ valeurs différentes potentielles en entrée. Cependant, dans ce cas-là, elle n'aura que $26^10 \approx 1.411671e+14$ valeurs différentes possibles en sortie.

La fonction de réduction prend donc en entrée l'indice du maillon, les paramètres de l'espace de recherche et le hash ainsi qu'un pointeur vers le message en sortie. Tout d'abord, le programme génère une chaîne de taille 16 pseudo aléatoirement grâce aux octets du hash et l'indice du maillon. Réduire l'octet obtenu modulo *char_range* et additionner le résultat avec *ascii_offset* permet que la valeur de l'octet obtenu corresponde à un caractère dans l'espace de recherche. La taille finale du message en sortie est aussi déterminée pseudo

aléatoirement grâce au hash en entrée.

```
37  //Reduit un hash en un mot de passe seulement les paramètres de l'espace de recherche
38  /*
39      chain_index             Evite qu'un meme hash donne la meme réduction (évite les ramifications)
40      password_min_length     Taille min du mot de passe
41      password_max_length     Taille max du mot de passe
42      ascii_offset            Premier charactere dans la table ASCII de l'espace de recherche (ex: a-z -> ascii_offset=97 )
43      chars_range             Taille de l'espace de recherche (ex: a-z -> chars_range=26 )
44      hash                    Le hash à reduire en mot de passe
45      new_passwd              Mot de passe généré à partir de la reduciton du hash
46  */
47  void str_reduction(unsigned int chain_index,unsigned int password_min_length, unsigned long int psswd_max_length,  unsigned int ascii_offset, un
48  {
49      //Alea ne peut pas etre utilisé donc pseudo random pour que l'espace de recherche soit réparti équitablement
50      unsigned int psswd_length = 0; //(rand() % (psswd_max_length - password_min_length + 1)) + password_min_length;
51
52      //Pour chaque Bytes du hash
53      for(unsigned int charIdx=0; charIdx != MD5_DIGEST_LENGTH;charIdx++)
54      {
55          new_passwd[charIdx] = ((hash[charIdx]+chain_index) % char_range)+ascii_offset; //Permet d'avoir un charactere dans l'espace de recherche
56          psswd_length += hash[charIdx];
57      }
58      //Permet de definir une taille de MDP en sortie en fonction des données
59      psswd_length = (psswd_length % (psswd_max_length - password_min_length + 1)) + password_min_length ;
60
61      //On coupe la chaine pour obtenir la taille voulue
62      //memset(new_passwd+psswd_length, 0, PSSWD_AREA - psswd_length );
63      memset(new_passwd+psswd_length, 0, MD5_DIGEST_LENGTH - psswd_length );
64  }
```

Après plusieurs tests et recherches, nous avons décidé de garder cette version de notre fonction de réduction. En effet, en générant des tables de très grandes tailles pour des messages de tous types de caractères, nous avons constaté que très peu de doublons apparaissaient et n'avons pas constaté de ramifications. Cependant, dans le cas de mots de passe avec une taille variable, les messages en sortie ne sont pas parfaitement répartis dans l'espace de recherche. En effet, pour des messages en sortie compris entre 10 et 15 caractères, notre fonction de réduction produit autant de sorties de 10 caractères que de sorties de 15 caractères, **or il y a beaucoup plus de "mots" possibles de taille 15 que de taille 10.**

Programme générant les chaînes dans la table arc-en ciel:

```
74  /* Fonction générant la rainbow table
75
76      fp                      Fichier dans lequel générer la rainbow table
77      M                       Nomre de chaine
78      T                       Taille d'une chaine
79      password                Mot de passe de départ pour le maillon M=1 et T=1
80      password_min_length     Taille min du mot de passe
81      password_max_length     Taille max du mot de passe
82      ascii_offset            Premier charactere dans la table ASCII de l'espace de recherche (ex: a-z -> ascii_offset=97 )
83      chars_range             Taille de l'espace de recherche (ex: a-z -> chars_range=26 )
84  */
85  void gen_rainbow_table(FILE* fp, unsigned int M, unsigned int T ,unsigned int password_min_length, unsigned int password_
86  {
87      //Ecriture des paramètre sur la premiere ligne
88      fprintf(fp,"%d;%d;%d;%d;%d;%d\n",T,M,password_min_length,password_max_length,ascii_offset,chars_range);
89
90      //Variable dans laquelle est stockée le hashé temporairement
91      unsigned char * md5_hash = (unsigned char *) calloc(MD5_DIGEST_LENGTH+1, sizeof(unsigned char));
92
93      //Mot de passe à partir duquel est dérivé tous les hashés et reductions
94      unsigned char * password    = (unsigned char *) calloc(PSSWD_AREA, sizeof(unsigned char));
95
96      //Permet d'obtenir un mdp random dans l'espace recherché
97      get_rnd_str(password_min_length, password_max_length,ascii_offset,chars_range,password);
98
99      //Pour chaque ligne
100     for(unsigned int I=0;I<M;I++)
101     {
102         fprintf(fp,"%s ",password); //Ecriture dans le fichier de la tête de la chaine
103
104         //Pour chaque F k = R k ∘ F (Reduction + Hashage)
105         for(unsigned int K=0;K<T;K++)
106         {
107             md5(md5_hash,password);
108             //memset(password, 0, PSSWD_AREA );
109             memset(password, 0, strlen(password) );
110             str_reduction(K+1,password_min_length, password_max_length,ascii_offset,chars_range,md5_hash,password);
111         }
112         fprintf(fp,"%s\n",password); //Ecriture dans le fichier de la queue de la chaine
113
114         //Nouveau mot de passe en début de chaine (dans l'espace de recherche)
115         //memset(password, 0, PSSWD_AREA );
116         memset(password, 0, strlen(password) );
117         get_rnd_str(password_min_length, password_max_length,ascii_offset,chars_range,password);
118     }
119     free(md5_hash);
120     free(password);
121 }
```

Figure 1: Génération d'une table

La partie générant les chaînes est relativement simple. Pour chaque chaîne (au total M chaînes), le programme génère une tête de chaîne aléatoirement dans l'espace de recherche, l'écrit dans le fichier, applique T fois la fonction $F(x,i) = (R \circ H)(x,i)$ sur cette tête de chaîne et écrit le résultat, qui correspond à la queue de la chaîne, dans le fichier. Les deux valeurs de la chaîne sont séparées dans le fichier par un espace.

Programme recherchant une pré-image d'un haché donné grâce à une table arc-en-ciel :

```
138     //Pour chaque maillon des chaines
139     for(unsigned int i=T;i != 0;i--)
140     {
141         strcpy(hash_tmp,hash_searched);
142 
143         str_reduction(i,psswd_min_length,psswd_max_length,ascii_offset,nbr_chars,hash_tmp,str_password);
144 
145         for(unsigned int y=i;y != T;y++) //Hash + Réduction pour arriver au maillon courant
146         {
147             md5(hash_tmp,str_password);
148             //memset(str_password, 0, PSSWD_AREA);
149             memset(str_password, 0, strlen(str_password));
150             str_reduction(y+1,psswd_min_length,psswd_max_length,ascii_offset,nbr_chars,hash_tmp,str_password);
151         }
152 
153         fseek(fp, 0, SEEK_SET);
154         getline(&line, &len, fp);
155         while ((read = getline(&line, &len, fp)) != -1) { //Pour chaque ligne de la table
156             strtok(line, " ");
157             if(strcmp(strtok(NULL, ";\n"),str_password)==0) //Si le mot de passe dérivé correspond à la queue d'une chaine
158             {
159                 strcpy(original,strtok(line, " ")); //On récupere la tête de chaine courante, on repart de la tête de la chaine pc
160                 if(find_psswd_from_head(i-1,original,hash_searched,psswd_min_length,psswd_max_length,ascii_offset,nbr_chars)==1){
161                     fclose(fp);
162                     free(hash_searched);
163                     free(hash_tmp);
164                     free(str_password);
165                     return EXIT_SUCCESS;
166                 }
167             }
168         }
169         //memset(str_password, 0, PSSWD_AREA);
170         memset(str_password, 0, strlen(str_password));
171     }
172     printf("No match found ! \n");
```

Figure 2: Partie 1 de la phase de recherche

Tout d'abord, le programme parse les paramètres écrits sur la première ligne du fichier contenant la table. Ces paramètres définissent l'espace de recherche (taille maximum, taille minimum, format) ainsi que la taille M et T de la table arc-en-ciel. Le programme suit l'algorithme exposé dans le chapitre *"Phase de recherche du hash"* en dérivant le hash passé en paramètre successivement en partant de la fin et en comparant à chaque fois avec les queues de toutes les chaînes dans la table. Si une queue correspond au mot de passe dérivé, le programme passe à la deuxième partie de la phase de recherche avec la fonction **find_psswd_from_head()**.

```c
//Retrouve le mot de passe donnant le hash recherché grâce à la tête de la chaine passé en paramètre
/*
    idx_maillon             l'indice du maillon ayant donné une queue de chaine se trouvant dans la table
    head_chain              Le mot de passe en tete de chaine permettant de dériver jusqu'au hash recherché
    origin_hash             Le hash recherché
    password_min_length     Taille min du mot de passe
    password_max_length     Taille max du mot de passe
    ascii_offset            Premier charactere dans la table ASCII de l'espace de recherche (ex: a-z -> ascii_offset=97 )
    nbr_chars               Taille de l'espace de recherche (ex: a-z -> chars_range=26 )
*/
int find_psswd_from_head(unsigned int idx_maillon, unsigned char * head_chain, unsigned char * origin_hash,unsigned int ps
{
    unsigned char * hashTMP = (unsigned char *) calloc(MD5_DIGEST_LENGTH+1, sizeof(unsigned char));
    unsigned int K = 0 ;

    md5(hashTMP,head_chain);
    //On hash et on réduit successivement à partir de la tête de la chaine jusqu'à arrivé au maillon ayant donné une queu
    for(unsigned int K=0; K < idx_maillon ; K++ )
    {
        memset(head_chain, 0, strlen(head_chain));
        str_reduction(K+1,psswd_min_length,psswd_max_length,ascii_offset,nbr_chars,hashTMP,head_chain);
        md5(hashTMP,head_chain);
    }
    //Une fois arrive sur le maillon ayant donné une queue de chaine dans la table
    //On regarde si ce maillon donne le hashé recherché
    if(memcmp(hashTMP,origin_hash,MD5_DIGEST_LENGTH+1)==0)
        {
            head_chain[strlen(head_chain)-1] = 0;
            printf("Mot de passe original : %s \n",head_chain);
            free(hashTMP);
            return 1;
        }
    //printf("Faux positif ! \n");
    //Des faux positifs peuvent arriver puisque la fonction de reduction est une surjeciton
    free(hashTMP);
    return 0;
}
```

Figure 3: Partie 2 de la phase de recherche

Cette fonction réalise la deuxième partie de la phase de recherche qui consiste, à partir d'une tête de chaîne, à appliquer une succession de hashages et de réductions jusqu'à obtenir le mot de passe censé donner le hash recherché. La fonction prend en entrée le maillon supposé produire le hash recherché, la tête de chaîne, le hash recherché et les paramètres de l'espace de recherche utiles à la fonction de réduction. Le programme repart donc de la tête de la chaîne et applique autant de fois que nécessaire la fonction F jusqu'à arriver à l'indice du maillon supposé donner le hash recherché.

Une fois ceci fait, on hash la valeur du maillon et on compare le résultat au hash recherché. Si les deux valeurs correspondent, alors on a trouvé une pré-image du hash recherché. Dans le cas contraire, il s'agit d'un faux positif.

3 Utilisation des programmes

3.1 Table arc-en-ciel pour des hashés

3.1.1 Générer une table arc-en-ciel

Concernant le programme générant la table arc-en-ciel, il est nécessaire de renseigner en paramètres, T la taille des chaînes et M le nombre de chaînes composant la future table. De plus, la taille maximale et minimum des mots de passe doit être renseignée. Le programme demande ensuite le format du mot de passe (minuscules, majuscules, chiffres, tous les caractères). Ces derniers paramètres permettent de réduire l'espace de recherche et ainsi maximiser les chances de trouver un mot de passe engendrant le hash donné. Comme expliqué dans le chapitre *"Fonction de réduction et espace de recherche"* la fonction de réduction générera des mots de passe selon ces contraintes indiquées.

```
jean@VM-Jean:~/Proj/DLC_Projet$ ./gen 0
Veuillez indiquer en parametre :
 - La taille d'une chaine
 - Le nombre de chaines à générer
 - Taille minimum du mot de passe
 - Taille maximum du mot de passe
jean@VM-Jean:~/Proj/DLC_Projet$ ./gen 1000 1000 5 10
Password range: Lower case only (0) - Upper case only (1) - numbers only (2) - a
ll chars (3)
0
---Paramètres---
 T: 1000
 M: 1000
 Password minimum length: 5
 Password maximum length: 10
 ASCII OFFSET: 97
 NBR CHARS: 26
-----------------
Fichier table_1000X1000_5_LOWER_CASE généré
jean@VM-Jean:~/Proj/DLC_Projet$
```

Figure 4: Exécution du programme générant la table arc-en-ciel

Dans ce cas-là, on génère une table de 1000 chaînes, chacune de 1000 maillons, pour des mots de passe en minuscules entre 5 et 10 caractères. Le programme affiche en sortie tous les paramètres renseignés et génère un fichier contenant la table arc-en-ciel générée :

```
1 1000;1000;5;10;97;26
2 hjkdanht kfspjorczl
3 opewnsbpc cboyaqalnl
4 dcojbsyjtn qqfvjbqcmn
5 iunsan marwcgh
6 hhtxya jvbeeg
7 lswdxygb vofknb
8 rcgqmcfnk riefrs
9 afaoojxii fgwhnvcde
10 kejcambaud jrqlxxfayj
11 mfxetzj akdadyze
```

Figure 5: Contenu de la table arc-en-ciel générée

La première ligne dans le fichier correspond aux paramètres renseignés qui sont indispensables à la phase de recherche. Ensuite, chaque ligne correspond à une chaîne de la table. Pour chaque chaîne, est stocké

seulement la tête et la queue de celle-ci (le premier et le dernier maillon de la chaîne).

3.1.2 Retrouver un mot de passe à partir d'une table et d'un hash

Concernant la phase de recherche d'un mot de passe à partir d'un hashé, il suffit de passer en paramètres le fichier, contenant la table arc-en-ciel et généré par le programme précédent, ainsi que le hash au format hexadécimal:

```
jean@VM-Jean:~/Proj/DLC_Projet$ ./find table_1000X1000_5_LOWER_CASE d83571881ea6bdb49da14ac7f509d14e
---Paramètres---
 T: 1000
 M: 1000
 Password minimum length: 5
 Password maximum length: 10
 ASCII OFFSET: 97
 NBR CHARS: 26
-----------------
Mot de passe original : dzwpsoxuc
```

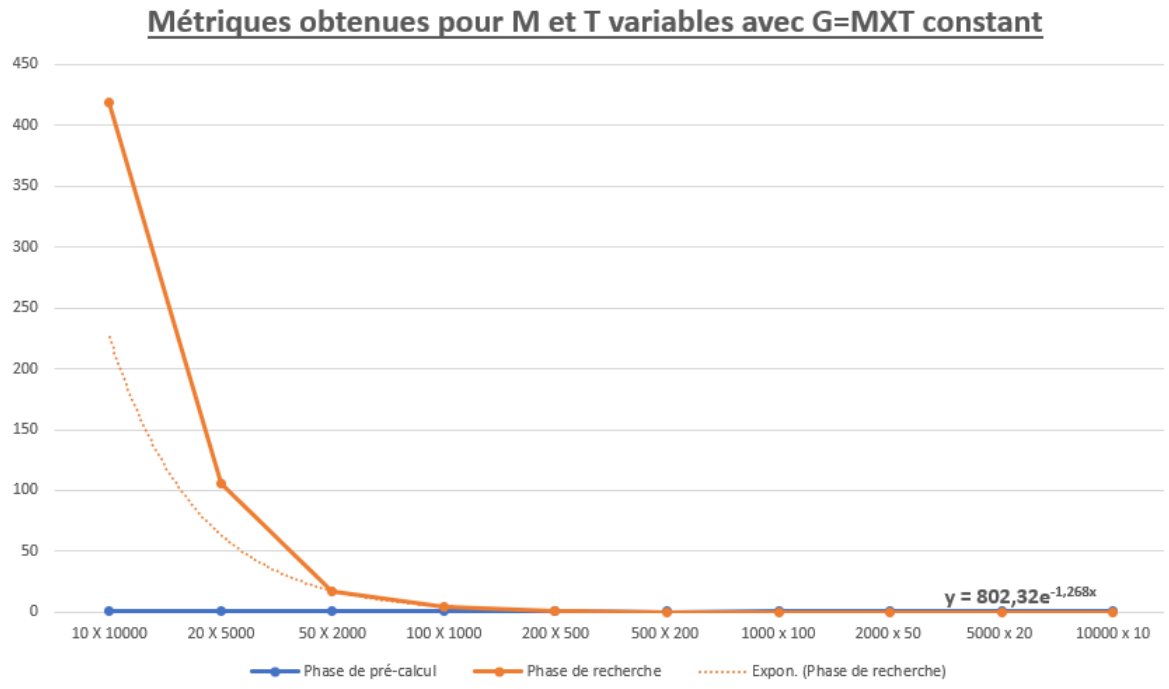
Figure 6: Exécution du programme réalisant la phase de recherche

Le programme affiche les paramètres de la table et se trouvant sur la première ligne du fichier. Enfin, il affiche, s'il en trouve un, un mot de passe engendrant le hash passé en paramètre.

4 Métriques et analyse des résultats obtenus

Voici, ci-dessous, l'évolution du temps d'exécution, du programme de génération d'une table arc-en-ciel ainsi que du programme permettant de retrouver un haché, quand M et T varient mais avec $G = MXT$ constant. Dans ce cas-là, le nombre total de mots de passe potentiels contenus dans la table ne varie pas.
Le temps d'exécution de la phase de recherche est mesuré dans le cas où le programme ne trouve pas de pré-image.

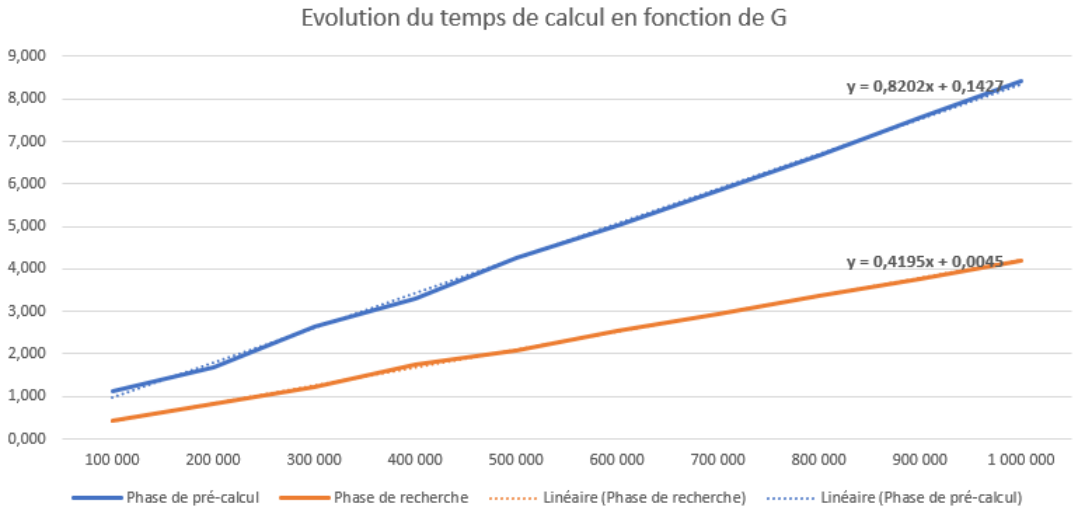
G=MXT = 100000	M	10	20	50	100	200	500	1 000	2 000	5 000	10 000,00
	T	10 000	5 000	2 000	1 000	500	200	100	50	20	10
Temps d'execution (s)	Phase de pré-calcul	0,8743	0,8460	0,8450	0,8680	0,8750	0,8340	0,8810	0,8660	0,8490	0,8660
	Phase de recherche	418,42	105,61	16,78	4,13	1,07	0,17	0,05	0,02	0,01	0,01
Poids de la table (Ko)	Poids de la table (Ko)	0,23	0,46	1,1	2,2	4,4	11	22	44	110	220



On constate, à travers les résultats obtenus, que l'augmentation de T, le nombre de chaînes, impactent énormément la phase de recherche d'un haché. Par ailleurs, la taille du fichier contenant la table augmente proportionnellement au nombre de lignes stockées. On remarque également que la phase de pré-calcul, quant à elle, n'est pas impactée seulement par M **ou** seulement par T.

Ci-dessous, l'évolution du temps d'exécution des deux programmes quand le nombre de mots de passe dans la table (G) augmente **mais que le rapport entre M et T est constant**. Dans ce cas-là $M = T = \sqrt{G}$:

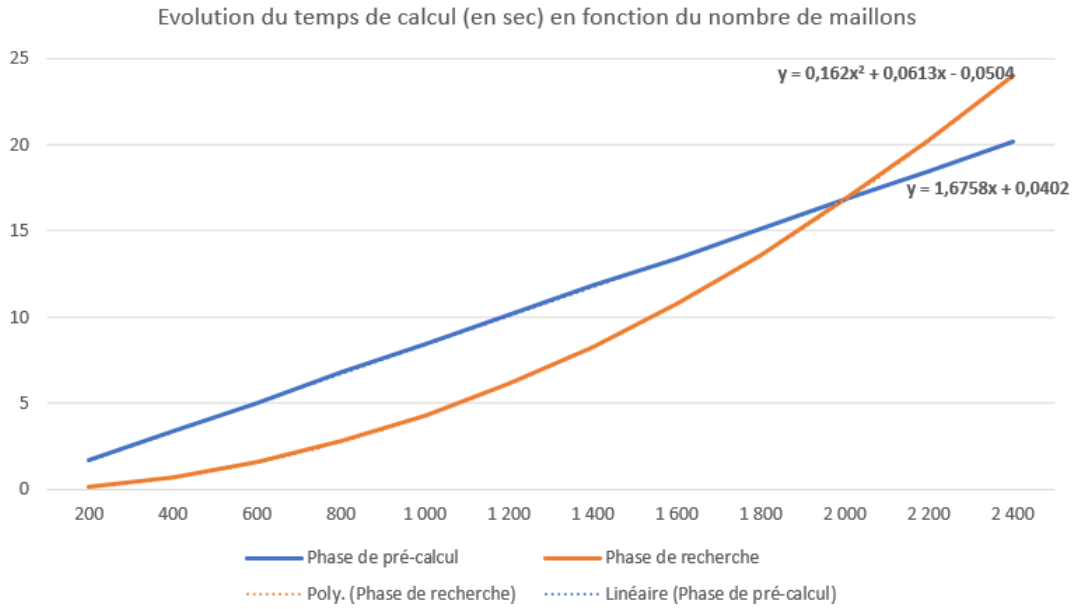
Avec M/T constant	G	100 000	200 000	300 000	400 000	500 000	600 000	700 000	800 000	900 000	1 000 000
	M	316	447	548	632	707	775	837	894	949	1 000
	T	316	447	548	632	707	775	837	894	949	1 000
Temps d'execution (s)	Phase de pré-calcul	1,120	1,680	2,650	3,320	4,260	5,020	5,830	6,670	7,560	8,430
	Phase de recherche	0,418	0,820	1,240	1,740	2,090	2,550	2,950	3,360	3,760	4,190



Bien que le premier cas ait montré que le temps d'exécution de la phase de pré-calcul n'était pas impacté par seulement M **ou** seulement T (mais par les deux en même temps), on constate, ici, que cette phase de pré-calcul est toutefois impactée par l'évolution de G. De plus, quand G augmente, c'est-à-dire le nombre de mots de passe potentiels contenus dans la table augmente, le temps d'exécution de la phase de recherche semble augmenter deux fois moins rapidement que la phase de pré-calcul.

Ci-dessous, les temps d'exécutions des deux programmes quand le nombre de maillons T augmente et que le nombre de chaînes M reste constant:

Avec M constant	**G**	**200 000**	**400 000**	**600 000**	**800 000**	**1 000 000**	**1 200 000**	**1 400 000**	**1 600 000**	**1 800 000**	**2 000 000**	**2 200 000**	**2 400 000**
	M	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000
	T	200	400	600	800	1 000	1 200	1 400	1 600	1 800	2 000	2 200	2 400
Temps d'execution (s)	Phase de pré-calcul	1,685	3,369	5,072	6,791	8,407	10,117	11,843	13,370	15,125	16,845	18,429	20,144
	Phase de recherche	0,182	0,702	1,600	2,820	4,270	6,160	8,300	10,832	13,570	16,828	20,270	23,978

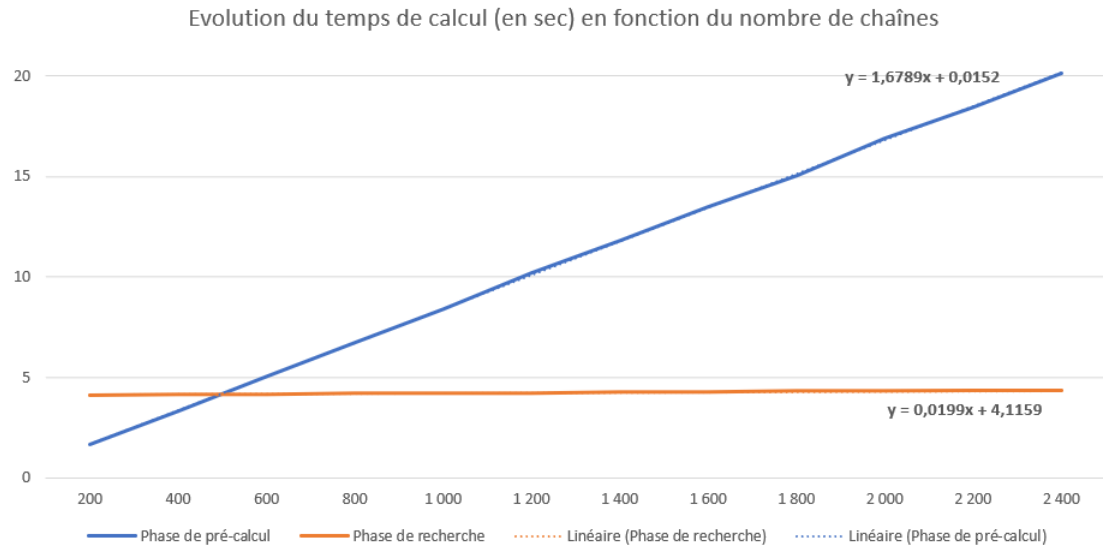


Quand le nombre de maillons augmente, on constate clairement que le temps de la phase de pré-calcul augmente linéairement tandis que la phase de recherche augmente de manière polynomiale. Contrairement à la phase de pré-calcul, puisque la phase de recherche est amenée à être réalisée plusieurs fois, une bonne valeur pour M et T peut correspondre au plus grand écart entre les deux courbes sur la section où le temps d'exécution de la phase de recherche est inférieur à celui de la phase de pré-calcul (ce n'est cependant pas le seul critère).

Ci-dessous, les temps d'exécutions des deux programmes quand le nombre de maillons T reste constant et que le nombre de chaînes M augmente:

Avec T constant	G	200 000	400 000	600 000	800 000	1 000 000	1 200 000	1 400 000	1 600 000	1 800 000	2 000 000	2 200 000	2 400 000
	M	200	400	600	800	1 000	1 200	1 400	1 600	1 800	2 000	2 200	2 400
	T	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000	1 000
Temps d'execution (s)	Phase de pré-calcul	1,673	3,328	5,030	6,728	8,410	10,230	11,794	13,470	15,030	16,880	18,440	20,123
	Phase de recherche	4,100	4,170	4,180	4,210	4,220	4,230	4,250	4,290	4,310	4,320	4,330	4,330

Figure 7: Exécution du programme réalisant la phase de recherche



Là encore, la phase de pré-calcul évolue linéairement ce qui est logique puisque l'on a vu que cette phase évoluait linéairement en fonction du nombre de maillons T mais qu'elle n'évoluait pas quand G était constant ($G = MXT$). Le nombre de chaînes, contrairement au cas précédent, fait évoluer linéairement le temps d'exécution de la phase de recherche. Concernant la phase de recherche, il est donc bien bénéfique de faire évoluer M au profil de T puisque dans un cas la courbe est polynomiale et dans l'autre cas linéaire.

Pour finir, voici quelques taux de réussite du programme de recherche de haché. Pour ces résultats, des mots de passe ont été choisis aléatoirement dans l'espace de recherche puis hachés. On a ensuite recherché une pré-image de ces hachés grâce à notre programme et mesuré le taux de succès :

Pour des mots de passe minuscules de 5 caractères

	M= 100000 T= 500
Taux de réussite moyen (30 tests)	**0,533**

Pour des mots de passe minuscules de 4 caractères

	M= 600 T= 600	M= 800 T= 800	M= 1000 T= 1000
Taux de réussite moyen (20 tests)	**0,2**	**0,5**	**0,65**

Remarque: Il n'est pas forcément bénéfique de générer des tables trop grandes pour un petit espace de recherche. En effet, ce cas-là causerait l'apparition de nombreux faux positifs pendant la phase de recherche du haché et impacterait grandement le temps d'exécution. Il faut donc choisir une taille de table en fonction du compromis temps-mémoire tout en prenant en compte l'obligation d'adapter G en fonction de la taille de l'espace de recherche.

Une table, dont la génération a nécessitée 70 minutes, de 1 000 maillons et 500 000 chaînes pour des mots de passe de 6 caractères minuscules est disponible à l'adresse suivante:
`https://drive.google.com/file/d/1aDjJIL705IyTaZu7E2HGpTbJ7fR9Xinm/view?usp=sharing`

5 Références

Webographie :

MD5 :

- https://fr.wikipedia.org/wiki/MD5#Pseudo-code
- https://fr.wikipedia.org/wiki/Fonction_de_hachage_cryptographique
- https://bibmath.net/crypto/index.php?action=affiche&quoi=moderne/md5
- https://github.com/pod32g/MD5/blob/master/md5.c

RC5 :

- https://en.wikipedia.org/wiki/RC5
- https://www.geeksforgeeks.org/rc5-encryption-algorithm/
- https://www.educba.com/rc5/
- https://github.com/stamparm/cryptospecs/blob/master/symmetrical/sources/rc5.c

Tables arc-en-ciel :

- https://en.wikipedia.org/wiki/Rainbow_table
- https://eprint.iacr.org/2010/176.pdf
- https://ee.stanford.edu/~hellman/publications/36.pdf
- https://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf
- https://www.thesslstore.com/blog/rainbow-tables-a-path-to-password-gold-for-cybercriminals/

Bibliographie :

- A Cryptanalytic Time - Memory Trade-Off _ MARTIN E. HELLMAN, FELLOW, IEEE
- A Comparison of Cryptanalytic Tradeoff Algorithms _ JIN HONG, SUNGHWAN MOON
- Making a Faster Cryptanalytic Time-Memory Trade-Of _ PHILIPPE OECHSLIN