UE : PROJET 2

mis en page par

Marta BOSHKOVSKA, Luc GUYARD, Antoine LAROCHE,
Sara SALÉ, Gireg GAMBRELLE

Licence 3 Informatique
Groupe 1B

AVRIL 2023

UNIVERSITÉ
CAEN
NORMANDIE

Coloration CS : Tables arc-en-ciel

Table des matières

1 Introduction

Dans le cadre de notre UE Projet 2, nous avons choisi le projet coloration cybersécurité Table arc-en-ciel construit autour de la question scientifique : quelle est l'efficacité de la table en fonction du nombre de couleurs, de sa taille et de la taille des mots de passe ?

Les mots de passe ne sont jamais stockés en clair dans une base de donnée pour des raisons de sécurité et aujourd'hui, la sécurité des mots de passe est un réel probléme : il faut prévenir de tout piratage et veiller au respect des normes de sécurité. Afin de tester ou forcer la sécurité, plusieurs méthodes sont utilisées notamment les tables arc-en-ciel.

Ainsi, tout au long de ce rapport, nous vous présenterons les moyens mis en place pour réaliser notre projet, les processus de modélisation et d'expérimentation et les conclusions tirées de nos différentes évaluations.

2 Compréhension des tables arc-en ciel et objectif du projet

2.1 Explication du sujet

Les tables arc-en-ciel sont une structure de donnée permettant de retrouver un mot de passe à partir de son empreinte. On parle d'une table étendue qui contient plusieurs millions de mots de passe en clair associé à des valeurs de hachage générées par la succession de processus de hachage/réduction. Les processus de hachage les plus connus à ce jour sont SHA256 et MD5. Il faut également avoir une fonction de réduction qui pour la plupart du temps varie selon la position dans la table permettant de recréer un nouveau mot de passe à partir de l'empreinte. Cette fonction est marquée par une couleur propre à chaque empreinte.

Le processus pour retrouver un mot de passe dans une table arc-en-ciel à partir d'une empreinte consiste en ces étapes suivantes.

- On vérifie que le hash n'est pas déjà dans notre table en fin de table.
- Si c'est le cas parfait, on passe à l'étape suivante. Sinon, on cherche à retrouver une fin de table, seul hash contenu dans notre rainbow table. On cherche à atteindre une fin de table à partir de toutes les positions possibles (comprendre les couleurs). Pour une table de taille 40, on a dans le pire des cas une recherche à faire depuis les 40 positions de départ (0-39) jusqu'à la position 40, la fin de table. on a donc une complexité factorielle dans le pire des cas par rapport à la longueur de la chaîne.

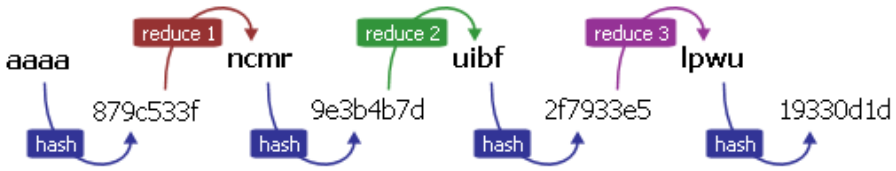


FIGURE 1 – Processus d'obtention d'un hash final, avec plusieurs hachages et réduction

- Une fois une fin de chaîne trouvée (la valeur) on a accès à un mot de passe qui marque l'entrée de la même chaîne (la clé). On a la position supposée du hash recherché par la position de départ choisi lors de la recherche, alors on effectue des réductions/hashes jusqu'à atteindre cette position.
  - Si le hash obtenu est le hash recherché, on renvoie le mot de passe réduit qui a produit cette chaîne.
  - Sinon, cela signifie qu'il y a eu une collision et on poursuit la recherche à l'étape 2 à l'endroit où nous nous étions arrêtés.
- Si on ne trouve pas le hash obtenu après avoir parcouru la recherche (0-39) en entier, deux cas de figure :
  - Le mot de passe n'était pas dans la table arc-en-ciel.
  - Il existe un cycle provoqué par une collision qui empêche l'accès au mot de passe pourtant présent dans la table. La probabilité de cet événement est de 1/(taille de la chaîne).

Le principe des tables arc-en-ciel est donc la réduction multiple et successif de hash à partir de clé pour arriver à une empreinte finale. En comparaison à d'autres méthodes de craquage, il y a un vrai compromis en matière de temps et de mémoire. Pour en savoir davantage sur les tables arc-en-ciel, vous trouverez des ressources supplémentaires en annexe.

Parmi les sujets proposés, nous avons choisi ce sujet car orienté Cyber-Sécurité. Nous voulions en apprendre davantage sur des thématiques liées à la sécurité. La réalisation semblait également intéressante sans pour autant être complexe, nous permettant de nous projeter un peu plus sur la question scientifique :

Quelle est l'efficacité de la table en fonction du nombre de couleurs, de sa taille et de la taille des mots de passe ?

2.2 Choix du langage et des librairies

Nous avons choisi le langage Python pour réaliser notre travail. D'une part, il permet l'utilisation d'un large éventail de librairies accessibles d'un point de vue modélisation, expérimentation et analyse de données. Et d'autre part, une grande partie des ressources en ligne et les travaux déjà existants s'appuient sur ce langage. Ayant déjà manipulé Python au cours de ces trois dernières années et après avoir demandé conseil à notre chargé de TP, ce choix nous a paru évident.

2.3 Objectifs

Les objectifs que nous nous sommes fixés pour réaliser ce projet sont les suivants :

- Créer un programme permettant la génération de table arc-en-ciel dans des formats de données divers ;
- Optimiser ce programme afin que l'enregistrement et la recherche dans les données soient plus faciles ;
- Créer un programme optimisé permettant de cracker un hash à partir de table générée précédemment ;
- Créer un client dans le terminal afin de simplifier l'expérience utilisateur ;
- Créer tous les tests unitaires ;
- Répondre à la question scientifique du sujet grâce à des tests utilisant nos programmes ;

Après cette étape de documentation et d'établissement des objectifs, nous nous sommes attaqués à la modélisation.

3 Modélisation

3.1 Organisation temporelle et humaine

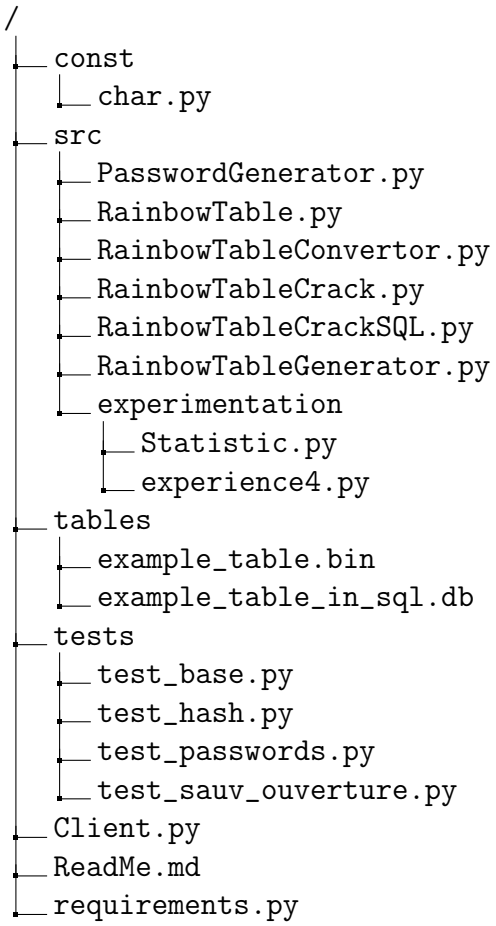
Voici un tableau présentant sommairement notre organisation. Une version plus explicite se trouve sur le wiki de la forge.

Semaines	Réalisations
Semaine 1	Documentation sur le sujet, choix du langages, définition des objectifs.
Semaine 2 et 3	Création des classes de génération d'une base de mots de passe, de création des fonctions de hash et de réductions. Stockage des mots de passe dans un dictionnaire binaire avec pickle.
Semaine 4	Mise en place de la classe pour craquer une table arc-en-ciel et des tests unitaires et préparation de l'oral de mi-parcours.
Semaine 5	Création d'un client afin de rendre l'expérience utilisateur plus agréable. Mise en place d'une base SQLite(langage SQL et fichier SQLite)en comparaison au binaire et implémentation d'un module statistique avec Numpy et Matplotlib
Semaine 6	Création d'un convertisseur binaire/sql et inversement. Ajout de nouvelles fonctionnalités au client et résolution de quelques problèmes d'implémentations. Amélioration du modèle statistique.
Semaine 7	Résolution de certaines bug et expérimentations

Sur le plan humain, la répartition des tâches s'est faite de manière progressive. Les premières semaines nous avons tous contribué. Puis après

- la génération de la table de mots de passe par Antoine
- la création de fonction de hachage et de réduction par Sara
- la documentation scientifique et les moyens de stockage des données par Luc
- les fonctions de crackage ont été réalisées par Marta & Gireg
- la mise en place des tests unitaires par Gireg
- la réalisation de l'interface client et l'implémentation du module statistique par Antoine

3.2 Architecture du projet



Il comprend 4 sous répertoires. Nous vous présenterons un peu plus en détail le répertoire des classes de la création, la gestion et la sauvegarde des tables et le répertoire lié aux tests. Les deux autres `const` et `tables` permettent respectivement de contenir des fichiers de variables globales et de stocker les tables créées.

Quant au fichier `Client.py`, c'est le fichier de lancement de notre programme. Ce dernier contient une interface client dans le terminal très simple. Ce client permet de faciliter la

manipulation des tables pour l'utilisateur. Par exemple, l'utilisateur peut choisir tous les paramètres lors de la génération comme ci-dessous :

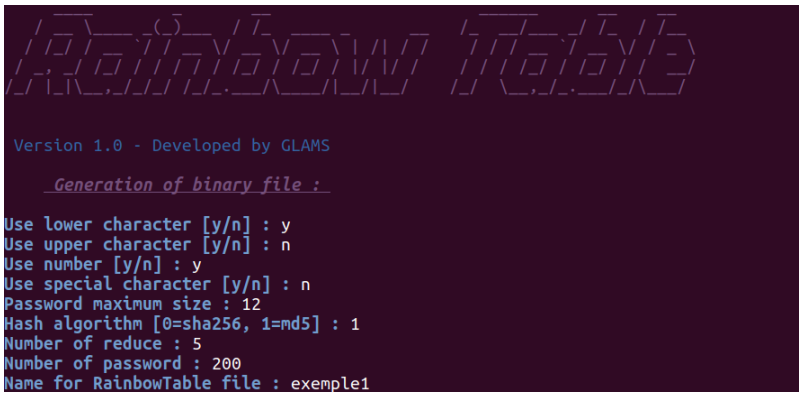


FIGURE 2 – Présentation de notre interface client

L'utilisation de toutes les fonctionnalités du client est détaillés dans le fichier `ReadMe.md` ou en utilisant la commande suivante :

```
$ python3 Client.py help
```

3.2.1 Tables arc-en-ciel

Pour réaliser les tables arc-en-ciel, nous avons mis en place 5 classes et une sixième classe pour l'aspect statistique de notre projet. Ce sont les classes suivantes :

- `PasswordGenerator` : gestion des critères de génération de mots de passe (taille minimale, caractères autorisés,...)
- `RainbowTable` : implémentation des fonctions de hash et de réductions ;
- `RainbowTableConvertor` : gestion de la sauvegarde des tables arcs en-ciel en binaire avec le module Pickle ou dans une base de donnée SQLite ;
- `RainbowTableCrack` : classe dédiée au craquage des mots de passe avec des fonctions. C'est en quelque sorte le backbone de la réalisation des tables arc-en-ciel ;
- `RainbowTableGenerator` : la gestion des données obtenues ;
- `Statistic` : analyse statistique suivie d'une représentation graphique de la génération des tables et leurs sauvegardes avec Numpy et Matplotlib.

3.2.2 Tests

Des tests unitaires ont été mis en place afin de vérifier la qualité de notre code à mesure que nous avancions. Ce sont :

- `test_base` : vérifie s'il y a bien une création de table arc-en-ciel avec
- `test_hash` : contrôle si dans la table générée, il existe bien des valeurs hachées ;
- `test_passwords` : vérifie si on peut bien générer tout type de mots de passe peu importe la longueur, la présence de caractères alphanumériques et spéciaux ;
- `test_sauv_ouverture` : vérifie que les fichiers créés suite à la création des tables sont ouvrables.

3.3 Processus de réalisation

Notre organisation temporelle est représentative des grandes étapes de notre processus de réalisation. C'est pourquoi dans cette partie, nous détaillons un peu plus nos fonctions majeures, leur impact et/ou leur importance dans la réalisation du projet.

3.3.1 Fonction de hachage

Les fonctions de hachage et de réduction sont étroitement liées et forment le socle de la création de tables arc-en-ciel.

Les fonctions de hachage utilisées sont SHA1, MD5 et GLAMS. Pour ce qui est des deux premières fonctions, elles ont toutes les deux étaient activement utilisées pour le hachage de mot de passe d'où leur intérêt dans nos expériences. Quant à la dernière, celle-ci est fortement inspirée de l'algorithme de Dan Bernstein et utilise la manipulation de bits et les nombres premiers pour créer un index de hachage à partir d'une chaîne de caractères. Notre apport a été la réduction du nombre de collisions avec l'ajout d'un petit caractère aléatoire au niveau du choix du nombre premier.

Ce que l'on attend d'une fonction de hachage suffisante :

- injective : Si hash(a) = hash(b) alors a = b (dans l'idéal car on sait que md5 et sha-1 présentent une faible probabilité de collisions)
- uniforme : la fonction de hachage parcourt l'ensemble de l'espace de sortie
- sensible à la perturbation : deux entrées extrêmement proches doivent avoir deux images très différentes
- non-inversible : la fonction inverse ne doit pas exister ou doit être impossible à effectuer en pratique.

3.3.2 Fonction de réduction

Pour la fonction de réduction, nous avons du concevoir notre propre fonction de réduction. Toutes les versions opèrent de la façon suivante :
la fonction reçoit un hash, un alphabet (charset), la longueur de la chaîne de caractère souhaitée et sa position dans la chaîne (color).

La fonction de réduction doit alors retourner un mot de passe tout en respectant certaines propriétés :

- injective
- uniforme

Afin de trouver les propriétés attendues, le plus simple est de s'appuyer sur une fonction de hachage ce qui permet de respecter ses propriétés sans effort. Dans la seconde on s'appuie donc sur sha256.

Algorithme 1 : Présentation de l'ancienne fonction de réduction

```
reducedText = "" pour letterIndex ← maximumSizePassword faire
    newLetterIndex = hashedtext[(color + letterIndex)%len(hashedtext)]
    conversion of letterIndex which is hex to int
    reducedText += ourcharset[newLetterIndex % len(charset)] new letter appended to
     the to-be returned stringfin
    retourner reducedText
```

La première fonction utilisée n'était ni injective ni uniforme ce qui provoquait énormément de collisions. en effet pour deux hashs avec les mêmes nombres hexadécimaux début on obtient la même réduction, c'est problématique.

Algorithme 2 : Présentation de la nouvelle fonction de réduction

```
Entrées : HashedText , color
pwd_length = maxPwdSize
 charset = getAllAcceptedChar()
 hash_obj = hashlib.sha256(hashedtext.encode())
 hash_bytes = hashobj.digest()
 combined_bytes = hash_bytes + color.to_bytes(4, byteorder = "big")
 reduced_text = ""
 tant que len(reduce_text < pwd_length faire
    hash_obj = hashlib.sha256(combined_bytes)
     hash_bytes = hash_obj.digest()
     combined_bytes = hash_bytes + color.to_bytes(4, byteorder = "big")
     hash_int = int.from_bytes(hash_bytes, byteorder = "big")
     index = hash_int%len(charset)
     reduced_text+ = charset[index]
fin
retourner reducedText;
```

La principale différence avec l'algorithme 1 est l'utilisation d'une fonction de hachage. Pour observer si la fonction est injective et surtout uniforme le fichier experience4.py nous fournit l'unicité (par la détection des doublons en utilisant un ensemble) des hashs générés à travers toutes les tables ainsi que l'unicité des mots de passe. Pour une table arc-en-ciel de grande taille on observe alors une unicité de 94% avec l'algorithme 2 contre 58% d'unicité pour l'algorithme 2.

3.3.3 Les fonctions de crack

Algorithme 3 : Stratégie pour trouver un mot de passe cracké

```
pour k ← N to 1 faire
    finalHash ← findTailHash(k)
    si finalHash alors
        pour tail ← tail in finalhash faire
            tableHead ←[head for head in rainbowSet if rainbowSet [head] = tail ]
            pour tail ← tail in finalhash faire
                solution ← findPasswordHash(tableHead)
                si solution! = None alors
                    retourner solution
                fin
            fin
        fin
        retourner None
    fin
fin
```

La partie 2.1 explique le fonctionnement de cet algorithme.

4 Expérimentation

4.1 Objectifs de l'expérimentation

La réalisation de ce projet a été effectué autour de la question scientifique : quelle est l'efficacité de la table en fonction du nombre de couleurs, de sa taille et de la taille des mots de passe ?

Afin de répondre à cette question on a fait de nombreuses expérimentations sur l'efficacité de notre fonctionnement en faisant varier plusieurs paramètres notamment la taille des mots de passe, les charsets utilisés pour générer les mots de passes aléatoires, le nombre de réduction (aussi appelé couleur) et le nombre de mot de passe.

N'ayant pas la possibilité de tout expérimenter et analyser, nous avons choisi de nous concentrer sur l'influence des paramètres ci-dessous dans notre recherche d'efficacité :

- la génération des tables,
- le rapport entre le pourcentage de succès du crack et le nombre de mots de passe,
- le rapport entre le temps moyen de crackage et le nombre de réduction,
- le rapport entre le pourcentage de succès du crack et le nombre de réduction
- le rapport entre le pourcentage de succès en fonction de la taille du mot de passe
- le rapport entre le pourcentage de succès en fonction du nombre de réduction, pour les mots de passe présents dans la table.

4.2 Expérimentation au niveau de la génération des tables

Nous avons des tables générées à partir de nos trois algorithmes de chiffrement (SHA1, MD5 et GLAMS) avec 1 000 000 de mots de passe et 20 réductions. Et notre génération de table se fait sous deux formats : SQLite et en binaire. Nous nous sommes positionnés selon trois angles :

- le temps (en secondes)
- la mémoire (en octets)
- le stockage (en mégaOctets)

Nous avons obtenu les deux graphiques suivants :

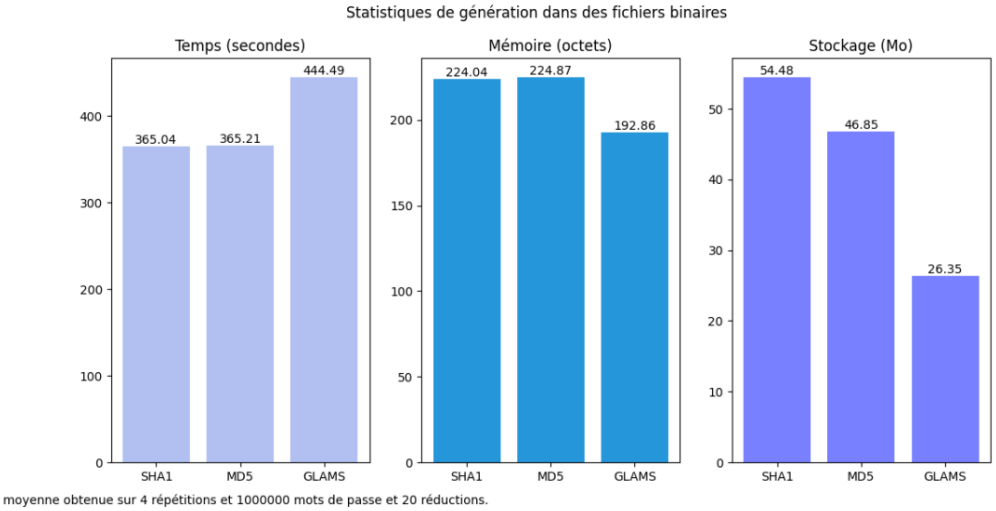


FIGURE 3 – Génération binaire

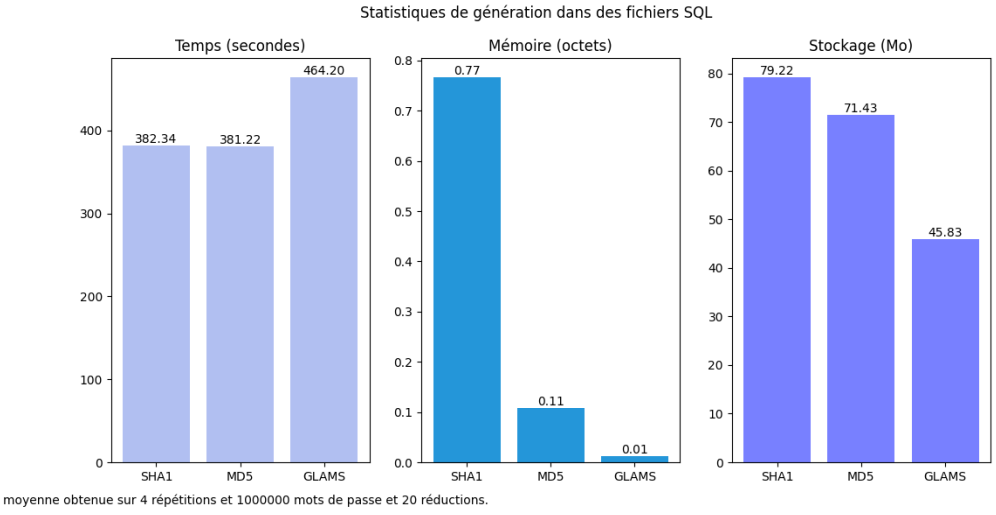


FIGURE 4 – Génération SQLite

4.2.1 Observations

Premièrement, on remarque que la génération des tables dans des fichiers binaires est légèrement plus rapide que dans des fichiers SQLite. Quant au stockage, Les fichiers SQLite

sont plus lourd que les fichiers binaires. Cependant, SQLite demande moins de mémoire pendant le calcul, car il ne stocke pas tous les résultats dans un dictionnaire avant de les écrire.

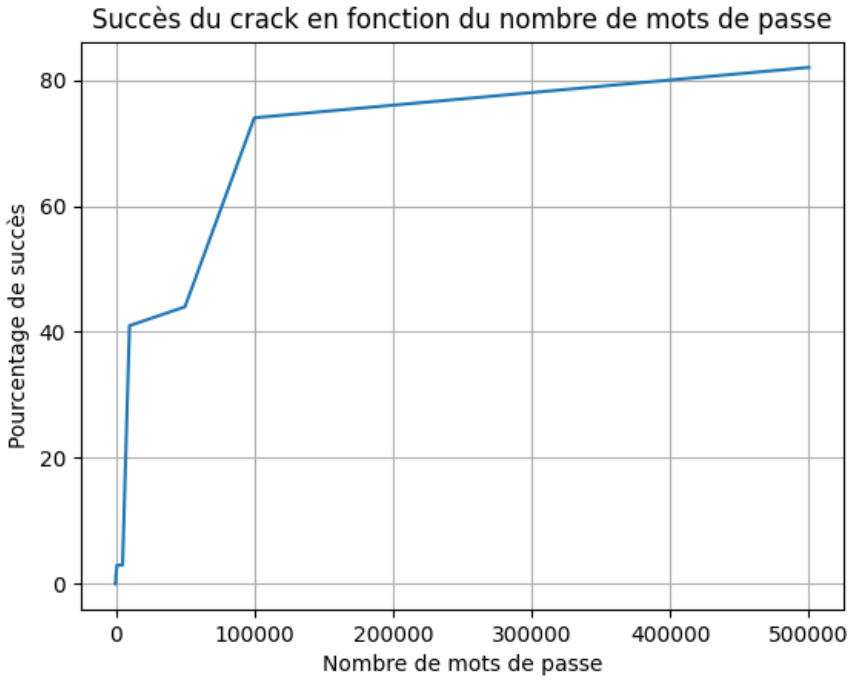
Dans ces cas de génération, on remarque aussi des variations selon les algorithmes :SHA1 et MD5 ont pratiquement la même efficacité en temps contrairement à l'algorithme GLAMS. Ce dernier est cependant plus long a exécuter. L'algorithme de Dan Bernstein est assez apprécié pour sa fluidité.

4.2.2 Conclusion partielle

Au niveau de la génération, les fichiers binaires montrent des avantages en terme de temps et de stockage. Ce qui est le principe même des tables arc-en-ciel. Les fichiers SQL, eux, offrent des avantages intéressants du point de vue de l'utilisation de la mémoire.

4.3 Expérimentation sur le succès d'un crackage en fonction du nombre de mots de passe

Le graphe ci-dessous montre le rapport entre le pourcentage de succès du crack et le nombre de mots de passe. Les tables utilisées vont de 100000 mots de passe à 500000 mots de passe et ont toutes été générées avec 40 réductions, un charset comprenant uniquement les lettres minuscules pour des mots de passe de 4 caractères.



4.3.1 Observations

Entre les tables de 0 à 50000 mots de passe, le pourcentage de succès est peu intéressant En ayant une table de 100000 mots de passe, l'évolution positive de la courbe est drastique.

Cependant, passé ce stade, on a une croissance assez ralentie, on a l'impression d'atteindre le pallier de croissance.

Dans notre cas, la table de 100000 est nettement plus avantageuse si l'on veut réussir à cracker efficacement.

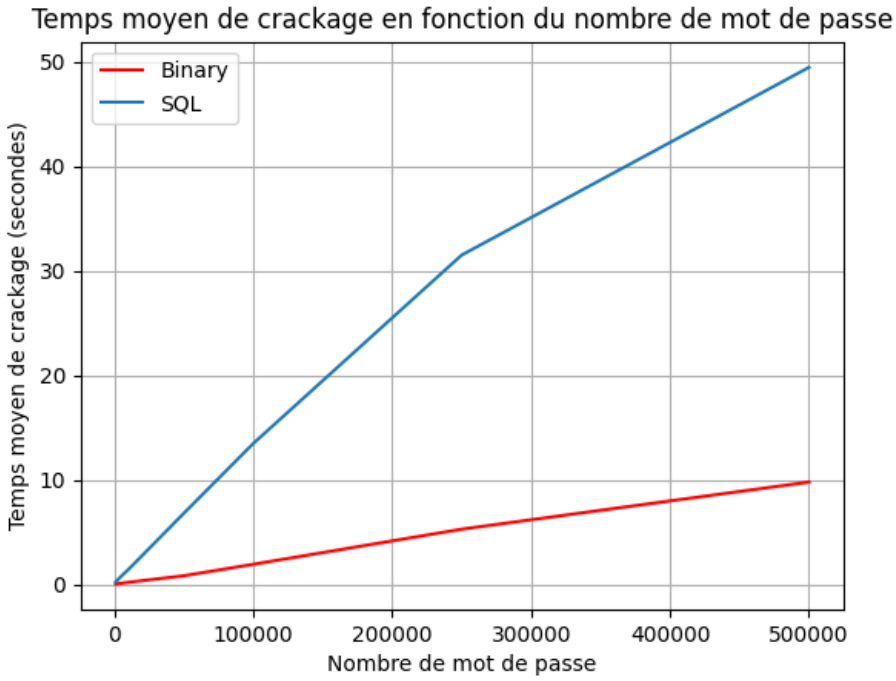
4.3.2 Conclusion partielle

En conclusion, il existe une limite à partir de laquelle augmenter la taille des tables favorise de manière négligeable le succès du crackage et il n'est pas nécessaire de dépasser cette limite. On risquerait d'avoir une grande table donc perte en mémoire sans avoir un pourcentage de succès conséquent, voir de créer des collisions inutiles.

Parallèlement, il serait intéressant de trouver un moyen de contrôle de la taille des tables pour ne pas rester dans l'intervalle du nombre optimal qui nous permet d'observer un réel succès sans empiéter inutilement sur la mémoire.

4.4 Expérimentation sur le temps moyen de crackage en fonction du nombre de mot de passe

Le graphe ci-dessous montre le temps moyen de crackage d'un mot de passe en fonction de la taille de la table arc-en-ciel dans les deux types de stockage.



4.4.1 Observations

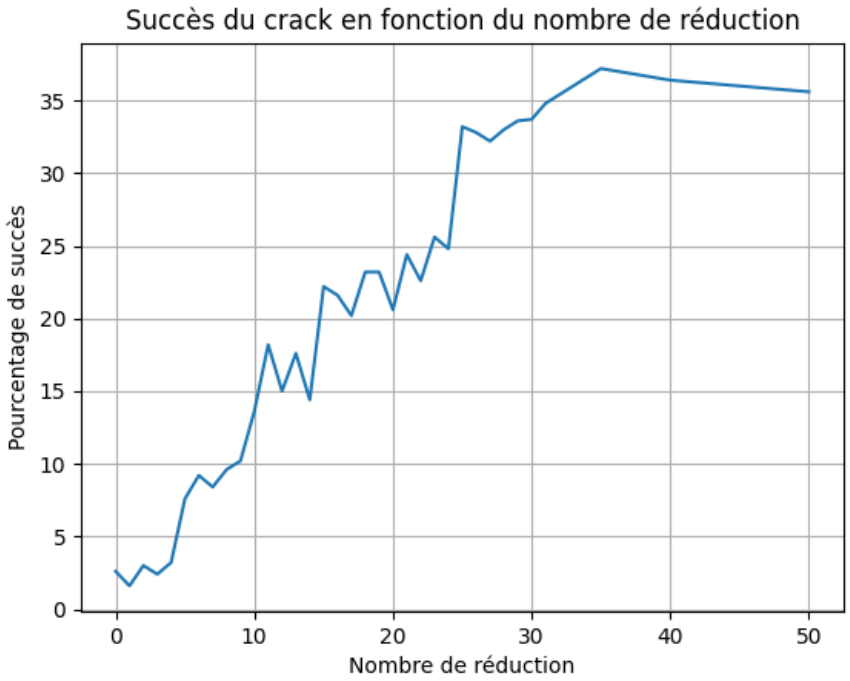
On remarque que dans les deux cas, le temps moyen pour cracker un mot de passe augmente si la taille de la table augmente. Cependant, Le crackage avec des fichiers SQL est nettement plus long qu'avec des fichiers binaires.

4.4.2 Conclusion partielle

Depuis le début de nos recherches nous étions convaincus que le crackage dans des fichiers SQL serait bien plus rapide que dans des fichiers binaires. Ce qui aurait compensé son temps de génération et en aurait fait le système de stockage le plus efficace. Malheureusement, ce n'est pas le cas et nous ne recommandons pas l'usage de ses fichiers pour une recherche efficace.

4.5 Expérimentation sur le succès d'un crackage en fonction du nombre de réduction

Le graphe ci-dessous montre comment le nombre de réduction influence le pourcentage de succès. Il a été généré avec des tables de 400 000 mots de passe, un charset composé des lettres en minuscules et des mots de passe de 5 caractères.



4.5.1 Observations

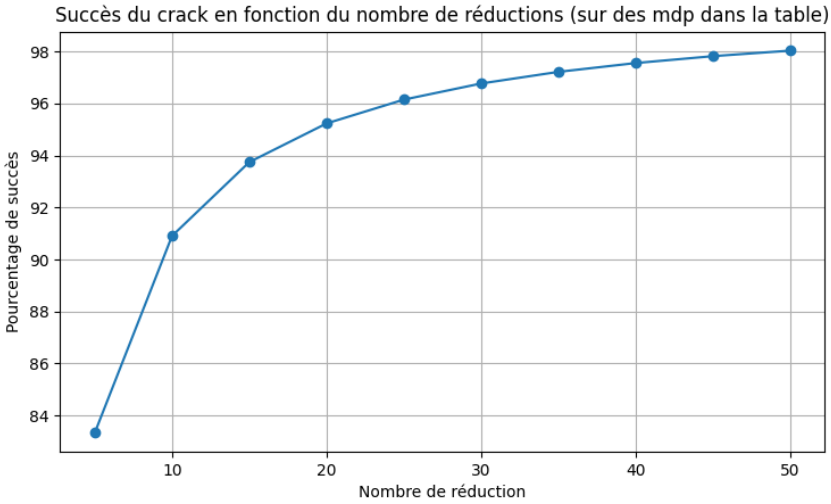
On constate que plus il y a de réductions ou couleurs, plus le pourcentage de réussite est élevé. Même s'il y a de légères diminutions du taux de réussite à certain moment, la courbe croît.

Cette légère diminution peut s'expliquer car les données ont été obtenue par l'utilisation de mot de passe générés aléatoirement.

4.5.2 Conclusion partielle

Augmenter le nombre de couleurs, c'est augmenter le taux de succès du craquage dans la majorité des cas .

4.6 Expérimentation sur le succès de crackage en fonction du nombre de réduction sur des mots de passe dans la table



4.6.1 Observations

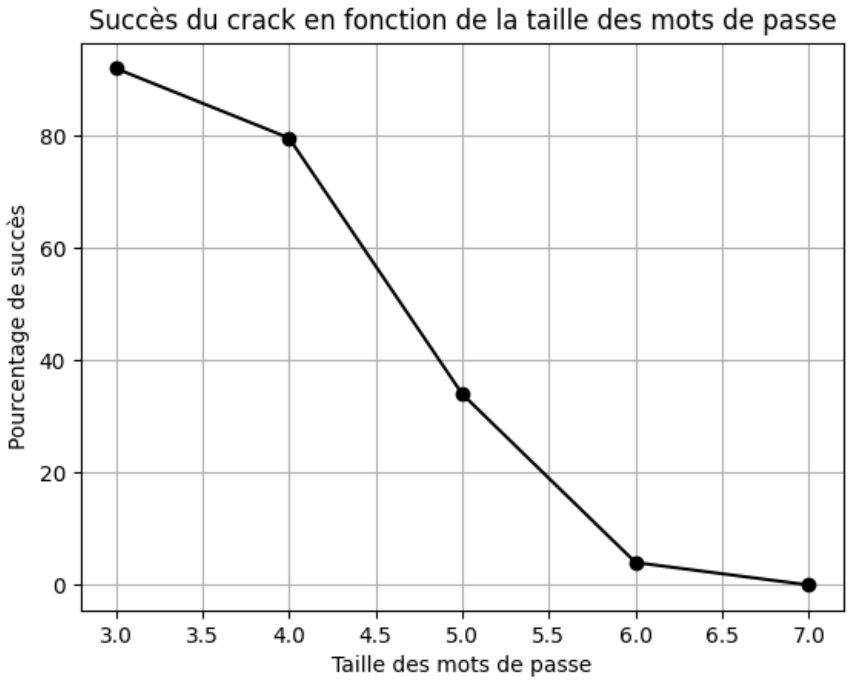
On peut remarquer que le succès de notre algorithme crack est plus élevé quand on utilise des mots de passe qui sont déjà dans la table.

4.6.2 Conclusion partielle

Cette expérience confirme notre conclusion précédente selon laquelle l'augmentation du nombre de réduction augmente la probabilité de trouver un mot de passe.
Même si le pourcentage de succès quand on utilise des mots de passe dans le tableau est plus élevé, on peut voir que la croissance est proportionnelle avec les cas quand on utilise des mots de passe aléatoires.

4.7 Expérimentation sur le succès de crackage en fonction de la taille des mots de passe

Le graphe ci-dessous montre comment la taille des mots de passes influence le pourcentage de succès. Les statistiques ont été obtenus avec des tables de 35 réductions, 250 000 mots de passe et un charset composé des lettres en minuscule.



4.7.1 Observations

On peut observer que l'augmentation des mots de passe entraine une diminution du pourcentage de succès.

Pour cette expérience, nous gardons le même nombre de mot de passe dans la table arc-en-ciel ainsi que le même nombre de réduction. Cette diminution est due au nombre de combinaisons de caractères possibles. Dans notre cas, nous n'utilisons que les 26 lettres minuscules de l'alphabet. Pour des mots de passes de 3 caractères, nous avons 3^{26} combinaisons contre 7^{26} combinaisons possibles pour un mot de passes de 7 caractères.

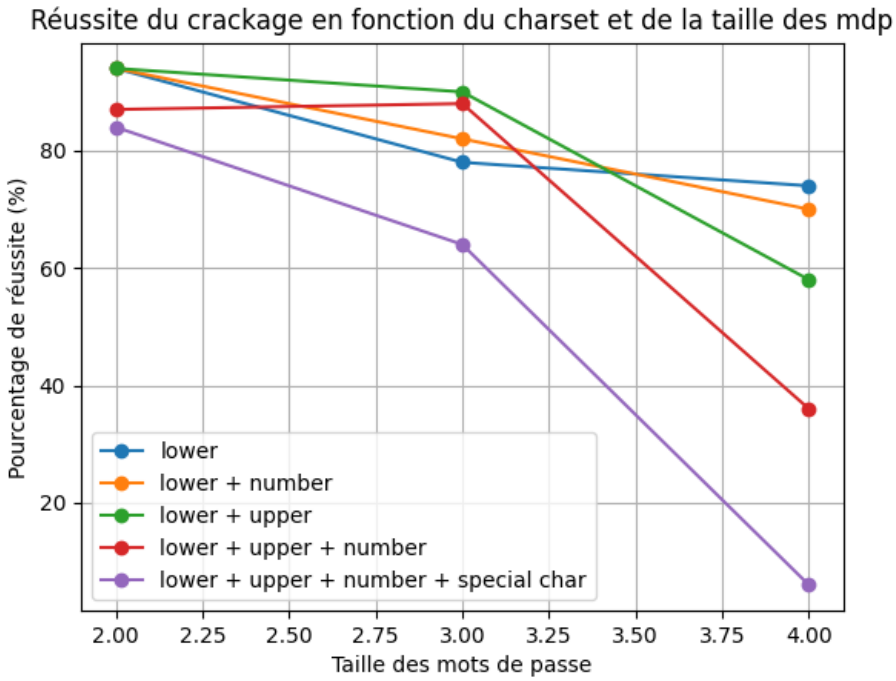
Il est donc logique que le pourcentage de réussite diminue.

4.7.2 Conclusion partielle

On remarque donc qu'il est important de réfléchir au nombre de mot de passe pour un table arc-en-ciel en fonction de la taille des mots de passe recherchés.

4.8 Expérimentation sur le succès d'un crackage en fonction de la variation du charset

Le graphe ci-dessous montre comment les différences de charset et la taille des mots de passe influence le pourcentage de réussite.



4.8.1 Observations

On peut remarquer qu'en ajoutant plusieurs options dans notre charset et en augmentant en même temps la taille de notre mot, on diminue le pourcentage de réussite de notre algorithme de craquage. Comme nous avons déjà expliqué la dépendance entre la taille des mots de passe et le pourcentage de réussite, nous allons nous intéresser à la variation de notre taille de charset et l'effet sur le pourcentage de réussite. Nous prenons comme exemple le cas ou notre charset inclut les majuscules, les minuscules, les chiffres et les caractères spéciaux (soit 107 caractères au total). Pour un mot de passe de taille 2, on aura $107^2 = \mathbf{11449}$ possibilités. Pour un mot de passe de taille 3, on aura **1225043** possibilités et **131079601** possibilités pour la taille 4. Ce graphe utilise une table contenant **250000** mots de passe avec 35 réduction chacune. Dans le cas idéal, nous n'aurons pas de doublon dans notre tableau, donc **8750000** chaînes différentes, ce qui n'est pas possible dans la réalité. A cause des doublons et des conflits présents dans notre table, le nombre total de chaînes sera réduit.

4.8.2 Conclusion partielle

En observant nos calculs, on peut conclure qu'en ajoutant des options dans notre charset, on augmente le nombre de chaînes possibles et consécutivement, on diminue la chance que notre chaîne soit présente.

5 Récapitulatif des expérimentations

Tout d'abord, il a été observé que générer un fichier binaire prend beaucoup moins de temps et d'espace de stockage que de générer un fichier SQL. Cette méthode peut donc être avantageuse pour stocker des données volumineuses.

Ensuite, notre algorithme de hashage GLAMS a été testé par rapport à MD5 et SHA1. Nous avons constaté que GLAMS était plus rapide, mais il avait besoin de beaucoup plus de mémoire.

En ce qui concerne la recherche dans une table, nous avons trouvé que l'utilisation d'un fichier binaire était plus efficace que l'utilisation de SQL en matière de temps. Cela peut également avoir une incidence sur le choix de la méthode de stockage des données.

L'étude a également montré que la taille des mots de passe avait un effet considérable sur la réussite de l'algorithme de crackage. De même, le nombre de réductions utilisées a un effet positif sur la réussite du crackage.

La variation de charset utilisé dans les mots de passe a été trouvée pour avoir un grand effet sur les chances de retrouver le mot de passe. Cela souligne l'importance de prendre en compte les différents charset dans la génération des mots de passe.

Compte tenu des différentes expérimentations faites et des différentes conclusions tirées, rendre efficace la table arc-en-ciel, c'est faire des compromis. Il ne suffit pas de remplir des conditions définies et de se retrouver dans un cas idéal mais plutôt de faire des choix de paramètres selon nos ressources et de créer un cadre idéal et efficace.

6 Pistes d'amélioration du programme

Au cours du projet et avec les enseignements du dernier semestre, nous avons eu plusieurs idées de poursuite potentiel du projet. Cependant, pour des raisons de temps nous n'avons pas eu l'occasion de les implémenter dans le code.

- **Génération :** Pour améliorer et rendre plus efficace la génération des tables arc-en-ciel nous pourrions par exemple utiliser le multithreading. Ce qui diminuerai de manière importante le temps de calcul.
- **Crack :** Pour améliorer la partie crackage, nous pourrions essayer de trier les hash afin de faire des recherches dichotomique (principe de "Diviser pour régner") et ainsi diminuer le temps de recherche
- **Statistique :** Utilisation de données plus importantes pour obtenir des analyses et des résultats plus précis ;
- **Ajout de différentes fonctionnalités :** un nouvel algorithme de craquage, un nouveau format de données.

7 Conclusion

En conclusion, suite aux expérimentations, nous avons remarqué qu'il est important de choisir les paramètres de nombre de mots de passe et de réduction en fonction du charset et de la taille des mots de passe afin d'avoir une recherche efficace.
Cependant, nous avons remarqué que la plupart des services modernes demandent des mots de passe suffisamment robuste pour se protéger de cette technique de crackage.
Ce projet à été pour tout le groupe une très bonne introduction au domaine de la cybersécurité en particulier du coté attaquant.
Nous avons réussi à développer une application permettant de créer, de manipuler et d'utiliser les tables arc-en-ciel munies d'une interface client. Malheureusement, nous n'avons pas pu aller au bout de toutes nos idées en matière d'amélioration et de performance de l'application.

Appendices

A Bibliographie et sitographie

Références

[1] Oechslin, P. (2003). Making a Faster Cryptanalytic Time-Memory Trade-Off. Advances in Cryptology - EUROCRYPT 2003, 617-630

[2] Que sont les Rainbow Tables ?

[3] Precomputation for Rainbow Tables Has Never Been so Fast

[4] Explication de DJB2

[5] How Rainbow Tables work

[6] How to Handle Rainbow Tables with External Memory