Algorithmique et programmation

Rapport de projet

1. Introduction

Question 1

a) Force brute

Soit C le nombre de caractères possible, et M la longueur des mots de passe. Il y a donc $N = C^M$ mot de passe possible.
On suppose ici que le calcul du hash à partir d'un mot de passe, et la comparaison entre deux hash est en temps constant (O(1)).

L'algo est le suivant :

Pour chaque mot de passe on calcul le hash puis on le compare au hash voulu. Si le test n'est pas concluant, on passe au mot de passe suivant. De plus, passer au mot de passe suivant se fait en O(M) (en pire cas mais O(1) amorti en incrémentant le dernier caractère, en propageant la retenue). Si on numérote les mots de passe comme un compteur en base C, l'opération "incrémente le compteur" a en pire-cas coût O(M) (toutes les positions portent), mais sur la totalité des $N = C^M$ incrémentations le nombre total d'opérations de caractère écrites est :

$$\sum_{i=0}^{M-1} C^M / C^i = C^M \cdot \sum_{i=0}^{M-1} C^{-i} = C^M \cdot \frac{1 - C^{-M}}{1 - C^{-1}} = \Theta(C^M)$$

Donc en pire cas on a une complexité en $O(N \bullet M)$ et en complexité amortie on a une complexité en $O(C^M)$.

Pour la complexité spatiale on a :

$O(M)$ stockage du mot de passe
$O(H)$ stockage du hash (calculé et cible)

Donc la complexité spatiale est $O(M + H)$.

b) Base de données pré-calculée

Pré-calcul :

- Pour le pré-calcul on a à chaque étape, le calcul du hash $O(1)$, stockage du hash dans le dictionnaire (ou table de hachage) $O(1)$, puis passage du mot de passe suivant (en $O(M)$ ou $O(1)$ amortis) et tout cela pour les $N = C^M$ mots de passe. Donc l'algo de pré-calcul se fait en $O(M \bullet C^M)$ en pire cas mais en $O(C^M)$ amortis.

Attaque après pré-calcul :

- Après le pré-calcul on cherche juste un antécédent du hash cible et donc une recherche dans la table pré-calculée suffit à trouver UN mot de passe correct, cette recherche se fait en $O(1)$.
- L'espace nécessaire est en $O(C^M)$ car on stocke tous les mots de passe ainsi que leur hash.

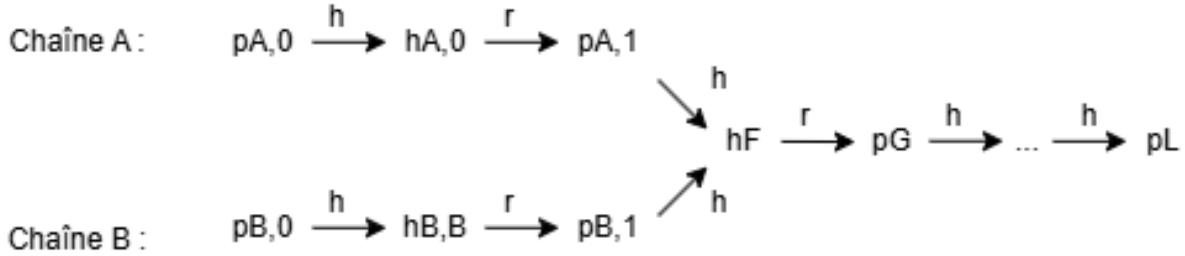
2. Méthode des hash chaînés : compromis temps - mémoire

Question 2

Pour cette méthode, on ne stocke pas les étapes intermédiaires mais uniquement les couples ($pass_{x,0}, pass_{x,L}$) avec L qui correspond à la longueur de la chaîne.
On doit donc être capable de vérifier rapidement si un mot correspond à une fin de chaîne connue.On souhaite stocker une grande liste de couples et la structure de données doit permettre de vérifier rapidement si un mot est un $pass_{x,L}$ connu et de retrouver rapidement le ou les $pass_{x,0}$ qui correspondent (car plusieurs chaînes peuvent se terminer sur la même valeur).
Ainsi, la structure de données la plus adaptée est la table de hachage, on pourra utiliser $pass_{x,L}$ comme clé et $pass_{x,0}$ comme valeur. Grâce à cette structure, pour chaque mot candidat il sera possible de rechercher si ce mot correspond à une fin de chaîne ($pass_{x,L}$) avec une complexité de l'ordre de O(1) ce qui rend la procédure d'attaque relativement efficace.

Question 3

Le candidat n'est pas nécessairement un antécédent du hash attaqué.
En effet, des collisions de hash ou des collisions en raison de la fonction de réduction sont possibles, ce qui ferait que plusieurs chaînes pourraient se rejoindre et se terminer par la même valeur. Ainsi, on se retrouverait dans une situation où deux $pX,0$ partagent le même pX,L.
Le mot de passe candidat pourrait donc être un "faux positif" et pas l'antécédent du hash attaqué. Il est donc nécessaire de vérifier si $h(Pcandidat) = inputhash$



Sur ce schéma, on voit que les deux chaînes A et B partent de mots de passe différents ($pA,0$ et $pB,0$), mais après application de h et de r à plusieurs reprises, elles aboutissent au même résultat hF en raison d'une collision de hash.
À partir de ce point hF, les deux chaînes deviennent identiques et se terminent sur le même mot de passe final pL. Ainsi, si l'on retrouve pL comme fin de chaîne dans notre table et que le mot de passe recherché se situe avant la collision alors il existera deux candidats différents, dont un seul est réellement l'antécédent du hash attaqué. Cependant, si le mot de

passe recherché se situe après la collision alors le candidat sera effectivement nécessairement l'antécédent du hash attaqué.
A noter que la collision peut également arriver lors de l'application de la fonction r, en effet r prend un hash très long et le transforme en un mot de passe avec des possibilités beaucoup plus restreintes, plusieurs hashs différents peuvent donc être réduits vers le même mot de passe ce qui produit le même problème.

Question 4

La fonction de hachage est déterministe donc si un mot de passe est représenté dans une chaîne de la table, on devrait pouvoir le retrouver en attaquant. Ainsi,si l'on ne trouve pas de candidat, cela signifie que le mot de passe n'est pas présent dans la table, c'est-à-dire qu'il ne fait partie d'aucune des chaînes représentées par les couples $(passX, 0,\ passX, L)$.

Question 5

Si $pass_{x,i} = pass_{y,j}$ avec $x \neq y$ alors le reste de la chaîne pour $pass_{x,>i}$ et $pass_{y,>j}$ sera identique comme illustré dans le schéma de la question 3. Ainsi, une partie de l'espace des mots de passe ne sera plus explorée, car deux chaînes censées être différentes se superposent. La conséquence est une perte de couverture et donc une diminution de l'efficacité de la méthode : certains mots de passe ne pourront pas être retrouvés.

3. Méthode des rainbow tables

Question 6

Si $i \neq j$, alors la collision de se propage pas car :
$pass_{x,i+1} = (r_i \, o \, h)(pass_{x,i})$ et $pass_{x,j+1} = (r_j \, o \, h)(pass_{x,j})$
En général (sauf si $r_i(h_{x,i}) = r_j(h_{y,j})$), on a donc $pass_{x,i+1} \neq pass_{y,j+1}$, et par conséquent les suites des chaînes restent différentes.

En revanche, si $i = j$, il y a bien collision, et dès lors les chaînes à partir de i et j pour x et y coïncident jusqu'à L.
Ce cas est problématique pour les mêmes raisons que celles évoquées dans la partie précédente : si $inputhash$ se situe avant la collision, il se peut que le candidat obtenu ne corresponde pas au résultat attendu.

Question 7

Pour que $pass_{x,L} = pass_{y,L}$, il faut qu'à un certain moment les deux chaînes se rejoignent, c'est-à-dire qu'il existe un indice i tel que $pass_{x,i} = pass_{y,i}$.

En effet, par déterminisme des fonctions h et r :

- Si $pass_{x,i} = pass_{y,i}$, alors les chaînes restent identiques jusqu'à la fin : $pass_{x,L} = pass_{y,L}$.

- Si $hash_{x,j} = hash_{y,j}$, alors $pass_{x,j+1} = r(hash_{x,j}) = r(hash_{y,j}) = pass_{y,j+1}$, et on retombe sur le cas précédent.

On peut donc se limiter au calcul des collisions sur les mots de passe, car toute collision sur les hash entraîne nécessairement une collision sur les mots de passe à l'étape suivante.

Soit $X = C^M$ le nombre de mots de passe possibles.

Pour des $pass_{x,0}$ et $pass_{y,0}$ choisis aléatoirement et différents, cherchons la probabilité qu'elles convergent à un moment donné de la chaîne.

Calcul de la probabilité de collision

Approche récursive

Soit A_i l'événement : "les deux chaînes n'ont pas le même mot de passe à l'étape i".

On veut calculer $P(A_L)$.

- Étape initiale : $P(A_0) = 1$ (les mots de passe initiaux sont différents par construction)

- Étape k :

$$P(A_k) = P(\text{pass}_{x,k} \neq \text{pass}_{y,k}) = \left(\frac{X-1}{X}\right)^k = \left(1 - \frac{1}{X}\right)^k$$

car la fonction de réduction r appliquée aux hashs produit une sortie uniforme. Et il faut "éviter" une valeur pour chaque étape i où $0 \leq i \leq k$.

Donc la probabilité exacte que deux chaînes distinctes n'aient pas le même mot de passe final est :

$$P(\text{pass}_{x,L} \neq \text{pass}_{y,L}) = \left(1 - \frac{1}{X}\right)^L = exp\left(L \cdot ln(1 - \frac{1}{X})\right) \approx e^{-L/X} \approx 1 - \frac{L}{X}$$

Car :

$$ln(1 - \frac{1}{X}) \approx -\frac{1}{X} \quad \text{et} \quad e^{-L/X} \approx 1 - \frac{L}{X}$$

pour

$$L \ll X$$

(L négligeable devant X).

Donc la probabilité exacte que deux chaînes distinctes aient le même mot de passe final est :

$$P(\text{pass}_{x,L} = \text{pass}_{y,L}) \approx \frac{L}{X}$$

Cas générale avec N chaine.

On suppose qu'on a déjà k chaîne dans la table. On génère une nouvelle chaîne $k + 1$.

Soit maintenant $i \leq k$.

Par le raisonnement de la Partie 1, la probabilité que la nouvelle chaîne $(k + 1)$ entre en collision avec la chaîne i est :

$$P(\text{collision avec la chaine i}) \approx L/X$$

La nouvelle chaîne doit éviter chacune des k chaînes existantes.

Si on note B_i : "collision avec la chaîne i".

On a alors :

$$P(\text{collision}) = P(B_1 \cup B_2 \cup ... \cup B_k) \approx \sum_{i=1}^{k} P(B_i) \approx k \cdot \frac{L}{X}$$

Ainsi si on a N-1 chaînes et que l'on en insère une N-ème,

On a :

$$P(\text{collision}) \approx (N-1) \cdot \frac{L}{X} \approx \frac{N \cdot L}{X}$$

Car $N - 1 \approx N$ pour N grand.

Application numérique (M=6, L=1000) :

$$P(\text{collision}) \leq 0.5 \Leftrightarrow N \leq 0.5 \cdot \frac{X}{L}$$

Pour $X = 26^6$ $L = 1000$

On a :

$$(\text{collision}) \leq 0.5 \Leftrightarrow N \leq 154,000$$

L'ordre de grandeur du nombre de chaînes qu'on peut facilement insérer est de N ≈ 150,000 chaînes

Avec $N = 100,000$ $L = 100$ $X = 26^6$.

$$P(collision) \approx 100,000 \times 1,000/26^6 \approx 0.32$$

Interprétation : Vers la fin du remplissage de la table, environ 1 tentative sur 3 sera rejetée car le $pass_{x,L}$ sera déjà présent.

Question 8

Implémentations

Avant de discuter des résultats nous devons parler un peu de notre implémentation. Celle-ci fut compliquée, obtenant au début des résultats proches de 1 ou 2%. Nous pensions que notre fonction de réduction originelle (ci-dessous) était à blâmer. Celle-ci fut créée en jouant avec les opérations utilisées dans le hash fournit par le sujet avec les différents paramètres (variation, indice du caractère ciblé, taille de l'alphabet…).

```
void reduction(uint64_t hash,int variation,int allowed_chars_length, char* allowed_chars,
char* pass)
{

    uint64_t modified_hash = hash * (variation + 1);
    int dec = 1+variation%allowed_chars_length;

    for (int i = 0; i < M; i++)
    {
        pass[i] = allowed_chars[modified_hash % allowed_chars_length];
        modified_hash ^= modified_hash >> (dec+i );
    }
    pass[M] = '\0';
}
```

Cependant nous avons par la suite découvert que le problème ne provenait pas de celle-ci mais de notre attaque. Une fois avoir trouvé un hash candidat, l'antécédent testé ne correspondait pas à l'antécédent du hash ciblé. Une simple erreur d'inattention dans notre code nous a donc fait perdre beaucoup de temps.

Une fois cela réglé nous sommes passé à 71.7% de mots trouvés de taille 6 sur les 10000. Pensant que les quelques pourcents restant était lié à notre fonction de réduction encore une fois, nous avons effectué des recherches sur des stratégies de réduction connues afin d'en obtenir une meilleure divisée en deux méthodes :

```
uint64_t avalanche(uint64_t x)
{
    x ^= x >> 33;
    x *= 0xff51afd7ed558ccd;
    x ^= x >> 33;
    return x;
}
```

Cette sous fonction reprend la stratégie du hash MurmurHash3 :
https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp#L81
MurmurHash3 étant connu pour être une méthode de hash rapide et facilement réversible. La stratégie empruntée à celui-ci permet en quelques opérations d'effectuer de gros

changements sur l'ensemble de l'entrée. Plus précisément, cette méthode applique des Xorshift (https://en.wikipedia.org/wiki/Xorshift) avec certaines constantes précises permettant une bonne répartition des valeurs et de grosses conséquences aux opérations avec un coût minimum.

```
void reduction(uint64_t hash, int variation, int allowed_chars_length, char* allowed_chars,
char* pass)
{
   uint64_t state = avalanche(hash ^ ((uint64_t)variation * 0x9e3779b97f4a7c15));


   for (int i = 0; i < M; i++)
   {
      pass[i] = allowed_chars[state % allowed_chars_length];
      state = avalanche(state + i + variation);
   }

   pass[M] = '\0';
}
```

La fonction de réduction en elle-même utilise la fonction avalanche définie plus haut pour altérer efficacement le hash avant la transformation en caractère de l'alphabet. Cependant la fonction de hash utilise aussi quelques méthodes découvertes via recherches comme l'utilisation du xor qui permet une répartition efficace des modifications puisqu'il affecte tous les bits de la valeur d'entrée. De plus c'est un opérateur commutatif et associatif (donc facilement réversible).
La fonction de réduction utilise également la constante 0x9e3779b97f4a7c15 qui est un dérivé du nombre d'or. Cette constante permet une bonne répartition des propriétés originelles de la valeur d'entrée, elle est utilisée dans des variations de MurmurHash3. Plus concrètement son intérêt est dans sa capacité à impliquer chaque bit de la valeur d'entrée, permettant d'accentuer cet effet "d'avalanche" (petit changement, grosses conséquences). Permettant ainsi de réduire les chances de collision puisqu'on s'assure qu'aucune partie de la valeur d'entrée (bits) ne restera inchangée. Mais aussi que chaque partie changée ne sera pas de la même valeur pour chaque réduction.

Mais il se trouve que le problème ne provenait pas de la fonction de réduction mais de notre attaque encore une fois. En effet, nous gérions les cas de collisions entre deux chaînes d'une même table (deux pass0 qui donnent le même passL) en rejetant le passL si il est déjà présent dans la table, toutefois cela ne prenait pas en compte les cas de collisions entre chaînes de différentes tables, il était donc nécessaire de prendre en compte les cas de multiples candidats lors de l'attaque. Une fois ce problème réglé nous obtenons les résultats suivants :

Résultats

Avec les valeurs de l'énoncé et notre nouvelle méthode de réduction nous trouvons un total de 8679 antécédents sur les 10000 (donc 86.79%).

En utilisant notre code de création des tables avec les valeurs définies dans l'énoncé sur un pc de l'ensimag (ensipc312) nous mettons 59secs à générer les tables.
Pour ce qui est de l'attaque, sur le même ensipc celle-ci prend 3min32.
Sur nos pc personnels la génération des tables prend environ 50 secondes et l'attaque entre une et deux minutes.
Avec notre ancienne méthode de réduction nous trouvons un total de 7396 antécédents sur les 10000 (donc 73.96%).
En utilisant notre code de création des tables avec les valeurs définies dans l'énoncé sur un pc de l'ensimag (ensipc312) nous mettons 1m6 à générer les tables.
Pour ce qui est de l'attaque, sur le même ensipc celle-ci prend 5m38.

En comparant les résultats des deux fonctions de réduction on observe une différence non négligeable de 12% de mots de passe en plus trouvé avec la fonction utilisant les méthodes plus poussées. Malgré le fait que la source de nos difficultés originelles n'était pas la fonction de réduction, expérimenter avec une fonction que nous avions fait nous-même et une fonction qui s'inspire de méthodes connues nous a permis de comprendre l'importance de la réduction dans ce processus. Ainsi que de comprendre en quoi la répartition uniforme des modifications par une fonction de réduction est importante pour réduire les collisions.

Malgré cela, les résultats des deux fonctions sont bons. on retrouve environ 86% des mots de passe pour celle optimisée et 73% pour celle d'origine, ce qui est satisfaisant étant donné que nous n'avons stocké que 1 000 000 chaînes pour un espace de recherche de 26^6 mots de passe possibles.
Une piste possible d'amélioration est de faire moins de vérification notamment dans le nombre de collision dans la table elle-même et donc d'améliorer potentiellement significativement la rapidité d'exécution ce qui permettrait d'augmenter le R et donc de contrebalancer le nombre de collision par le nombre de tables.

Question 9

En tâtonnant avec les valeurs de L, N et R, nous arrivons à obtenir :
environ 99% de réussite avec les valeurs suivantes et la fonction de réduction optimisée :
L = 1800
N = 250 000
R = 20

Avec les paramètres :
L = 1500
N = 100 000
R = 15
Nous avons essayé de percer le fichier crackme2025_7.txt, ce qui nous a donné 23% de mots de passe trouvés en 6 min avec notre réduction optimisée.

4. Recherche documentaire

Question 10

De nos jours, il existe différentes manières de se prémunir contre ce genre d'attaques.
La technique la plus courante est le salage dont le principe est de concaténer une chaîne de caractères au mot de passe avant d'appliquer la fonction de hachage.

- Cette chaîne de caractères (sel) peut être fixe, c'est à dire qu'on utilise la même chaîne pour tous les mots de passe, cela force l'attaquant à connaître ce sel fixe pour pouvoir calculer la table.
- Le sel peut aussi être variable ce qui est nettement plus efficace pour se prémunir contre les attaques. Chaque mot de passe reçoit un sel différent, unique et aléatoire pour chaque utilisateur, il n'est pas destiné à être secret et peut être stocké en clair en base de données avec les informations de l'utilisateur, son intérêt est de forcer l'attaquant à recalculer une table pour chaque hash individuel ce qui est trop peu efficace et donc irréalisable.

En complément du salage, on peut utiliser ce qu'il s'appelle un pepper. Celui-ci fonctionne sur le même principe que le sel, il s'agit d'une valeur que l'on ajoute au mot de passe avant de le hacher. Contrairement au sel, le pepper est secret, généralement situé dans le code de l'application et pas dans la base de données, son intérêt est d'empêcher un attaquant ayant accès à la base de données (mot de passe et sels) de recalculer les hashs. Il oblige donc également l'attaquant à avoir accès au code source de l'application ou par exemple aux variables d'environnement en plus de la base de données.

Une autre méthode est de remplacer la fonction de hachage classique (comme sha256 par exemple qui est très rapide) par une fonction de hachage dédiée au hachage de mots de passe, aussi appelée fonction de dérivation de clée (KDF en anglais), dont des exemples sont : Argon2, scrypt, bcrypt ou PBKDF2. La particularité de ces KDF est qu'ils font exprès d'être lents et parfois gourmands en mémoire pour rendre les essais le plus coûteux possible pour l'attaquant. Ces KDF prennent en entrée le mot de passe, le sel unique par compte et des paramètres de coût (nombre d'itérations, mémoire) puis répètent des opérations de hachage de très nombreuses fois. On stocke ensuite le KDF utilisé, les paramètres choisis, le sel et le hash final qui est aussi appelé valeur dérivée ce qui permet de "rejouer" la vérification lors des futures authentifications de l'utilisateur. L'intérêt de cette méthode est également de pouvoir faire évoluer les paramètres au bout d'un certain temps, par exemple en augmentant le nombre d'itérations, les ordinateurs sont de plus en plus puissants et on souhaite donc pouvoir augmenter le "coût" du calcul à mesure que les puissances de calcul augmentent pour rester difficile à attaquer.
Enfin en complément, l'authentification multi-facteurs permet de limiter l'exploitation d'un mot de passe compromis, mais n'est pas une contre-mesure spécifique aux attaques par bases de hashs pré-calculées.

En résumé, le sel unique empêche la réutilisation des tables, le KDF rend chaque essai suffisamment cher pour que le pré-calcul soit irréalisable et le pepper ajoute une couche de sécurité en plus si la base de données est compromise.

Sources :

IONOS - Rainbow tables : explication simple des tables arc-en-ciel
RECOMMANDATIONS RELATIVES À L'AUTHENTIFICATION MULTIFACTEUR ET AUX MOTS DE PASSE
Key Derivation Functions — PyCryptodome documentation
Huntress - What's a rainbow table
Wikipedia - Salage