Algorithmique et programmation 1AA
Projet période 1 (2023)

Ce projet est à réaliser par équipes de 2 étudiant · es (on peut faire une unique équipe de 3 si nécessaire pour ne laisser personne seul), et à rendre (sur teide) pour le **13 octobre à 23h59**, soit trois semaines après la mise à disposition du sujet.

Vous rendrez une archive .tar.gz contenant votre compte-rendu (réponse aux questions) au format pdf (il n'est pas nécessaire d'être trop verbeux ni de passer trop de temps à la mise en forme), des fichiers textes contenant les mots de passe trouvés, et vos programmes en langage C accompagnés d'un Makefile, qui devront pouvoir être compilés (sans erreurs ni warnings) et executés sur les ordinateurs de l'ensimag. Veillez à respecter strictement les spécifications imposées.

La charte contre la fraude proposée par l'Ensimag s'applique. Notamment tout partage ou copie (même partielle) de programmes [1] ou de réponses avec une autre équipe est strictement interdit, ainsi que plus généralement toute aide extérieure.

Ce sujet est disponible sur `http://perso.crans.org/frenoy/ap1aa/projet_periode1.tar.gz`, et sera si nécessaire mis à jour pour répondre à d'éventuels signalements d'erreurs ou demandes de clarification.

1 Introduction

Dans un système informatique moderne, les mots de passe ne sont pas stockés en clair, mais sous forme hachée. Lorsque l'utilisateur fait une tentative de connexion, le mot de passe entré est haché et comparé au hash stocké. Ainsi, une personne ayant accès en lecture à la base de données de mots de passe ne peut pas l'utiliser pour se connecter frauduleusement, à moins d'inverser la fonction de hachage (c'est à dire de trouver un antécédent du hash). Si cette fonction est bien choisie, cette inversion se fait par force brute et requiert donc une puissance de calcul considérable (pour des mots de passe de complexité suffisante).

On s'intéresse ici à des techniques probabilistes permettant de trouver rapidement des antécédents de hashs en utilisant des structures de données pré-calculées.

Pour la suite de l'énoncé, on suppose que les mots de passe sont composés de M lettres minuscules exactement. Sauf précision contraire, les résultats doivent être donnés en fonction de M (et vos programmes doivent pouvoir s'adapter à différentes valeurs de M).

La fonction de hachage attaquée, qu'on notera h dans la suite de l'énoncé, est fournie dans `hash.h` : vous utiliserez impérativement cette implémentation pour l'écriture de vos programmes. Cette implémentation s'appuie sur la bibliothèque openssl, l'argument `-lcrypto` doit donc être donnée au compilateur, et vous aurez peut-être besoin d'ajouter d'autres arguments permettant d'éviter des warnings liés à l'utilisation de certaines fonctions.

Question 1 — *Quel est l'ordre de grandeur du nombre d'essais qu'on doit faire par force brute (c'est à dire en essayant tous les mots de passe séquentiellement) pour trouver l'antécédent d'un hash donné, en moyenne ?*

1. Il vous est en conséquence demandé de ne pas travailler dans un dépôt git accessible à tous en lecture

Si une grande partie des systèmes utilisent une même fonction de hachage standard, on peut imaginer calculer une seule fois et stocker les hashs associés à tous les mots de passe possibles, et utiliser par la suite cette base de données pré-calculée pour attaquer n'importe quel hash sur n'importe quel système.

Question 2 — *Quelle structure de données vous semble adaptée pour stocker ces informations pré-calculées (hashs associés à tous les mots de passe possibles) ? Quelles seront alors la taille de la base de données, et la complexité temporelle de la recherche d'un mot de passe à partir de son hash une fois cette base crée ?*

Question 3 — *Jusqu'à quelles valeurs de M l'utilisation de chacune de ces techniques (force brute et pré-calcul) vous parait-elle réaliste ?*

2 Méthode des hashs chainés : compromis temps - mémoire

Cette technique est un compromis entre le calcul par force brute (pas de pré-calcul et stockage mais la recherche d'un mot de passe est longue) et la création d'une base de données exhaustive (recherche d'un mot de passe rapide une fois la base calculée mais espace nécessaire considérable).

On utilise une fonction de réduction r qui transforme un hash en un nouveau mot de passe [2]. On peut ainsi composer une chaine de mots de passe et de hashs :

$$pass_0 \xrightarrow{h} hash_0 \xrightarrow{r} pass_1 \xrightarrow{h} hash_1 \xrightarrow{r} ... \xrightarrow{r} pass_{L-1} \xrightarrow{h} hash_{L-1} \xrightarrow{r} pass_L$$

En stockant seulement le premier et le dernier mot de passe de la chaine ($pass_0$ et $pass_L$), on peut facilement tester si un hash donné correspond à un mot de passe dans cette chaine : si en appliquant la fonction de réduction une fois au hash donné on obtient $pass_L$, alors le mot de passe cherché est probablement [3] $pass_{L-1}$. Si en appliquant la fonction de réduction puis la fonction de hash puis la fonction de réduction on obtient $pass_L$, alors le mot de passe cherché est probablement $pass_{L-2}$. Plus généralement et plus formellement, on applique les $(r \circ h)^i \circ r$ avec toutes les valeurs de i de 0 à $L-1$ au hash attaqué. Si on obtient $pass_L$, alors le mot de passe cherché est probablement $pass_{L-i-1}$.

Supposons qu'on calcule et stocke au total N chaines similaires à partir de mots de passes choisis aléatoirement :

$$pass_{1,0} \xrightarrow{h} hash_{1,0} \xrightarrow{r} pass_{1,1} \xrightarrow{h} hash_{1,1} \xrightarrow{r} ... \xrightarrow{r} pass_{1,L-1} \xrightarrow{h} hash_{1,L-1} \xrightarrow{r} pass_{1,L}$$

$$pass_{2,0} \xrightarrow{h} hash_{2,0} \xrightarrow{r} pass_{2,1} \xrightarrow{h} hash_{2,1} \xrightarrow{r} ... \xrightarrow{r} pass_{2,L-1} \xrightarrow{h} hash_{2,L-1} \xrightarrow{r} pass_{2,L}$$

$$...$$

$$pass_{N,0} \xrightarrow{h} hash_{N,0} \xrightarrow{r} pass_{N,1} \xrightarrow{h} hash_{N,1} \xrightarrow{r} ... \xrightarrow{r} pass_{N,L-1} \xrightarrow{h} hash_{N,L-1} \xrightarrow{r} pass_{N,L}$$

Pour trouver l'antécédent d'un hash à attaquer *inputhash*, on généralise le raisonnement précédent. On cherche si un des $(r \circ h)^i \circ r(inputhash)$ $\forall i \in [0, L-1]$ est trouvé parmi les $pass_{x,L}$ $\forall x \in [1, N]$. Si on trouve un match, $(r \circ h)^{L-i-1}(pass_{x,0})$ est le mot de passe candidat.

2. Les propriétés attendues de cette fonction sont les mêmes que pour une fonction de hachage : déterminisme (c'est une fonction au sens mathématique), sorties distribuées uniformément dans l'espace d'arrivée, et calcul efficace

3. On explicitera ce *probablement* dans les questions suivantes

Question 4 — *Le candidat est-il nécessairement un antécédent du hash attaqué ? Expliquez précisément pourquoi, en vous appuyant sur un schéma si nécessaire.*

Question 5 — *Quelle structure de données peut-on utiliser pour stocker les couples* $(pass_{x,0}, pass_{x,L})$ *de façon à rendre la recherche précédente relativement efficace ?*

Question 6 — *Si on a* $pass_{x,i} = pass_{y,j}$ *avec* $x \neq y$ *(on appelle cette situation une collision), quelles sont les conséquences pour cette méthode ?*

3 Méthode des rainbow tables

Cette technique améliore la technique précédente en faisant varier la fonction de réduction utilisée à chaque étape. Chaque chaine est ainsi de la forme

$$pass_{x,0} \xrightarrow{h} hash_{x,0} \xrightarrow{r_0} pass_{x,1} \xrightarrow{h} hash_{x,1} \xrightarrow{r_1} ... \xrightarrow{r_{L-2}} pass_{x,L-1} \xrightarrow{h} hash_{x,L-1} \xrightarrow{r_{L-1}} pass_{x,L}$$

L'ensemble des N chaines (toujours représentées par leurs premiers et derniers éléments) est appelé "rainbow table".

L'attaque d'un hash se fait de façon similaire [4] : on calcule l'image de ce hash par les différentes fonctions $\left[\prod_{j=1}^{j=i}(r_{L-j} \circ h)\right] \circ r_{L-i-1}$ $\forall i \in [0, L-1]$, et cherche les résultats obtenus parmi les $pass_{x,L}$ $\forall x \in [1, N]$. Si on trouve un match, $\left[\prod_{j=i+2}^{j=L}(r_{L-j} \circ h)\right](pass_{x,0})$ est le mot de passe candidat.

Question 7 — *Est-ce alors toujours aussi problématique d'avoir* $pass_{x,i} = pass_{y,j}$ *avec* $x \neq y$ *et* $i \neq j$ *? Pourquoi ?*

Question 8 — *Et si on a* $pass_{x,i} = pass_{y,i}$ *avec* $x \neq y$*, que se passe-t-il au niveau des* $pass_{x,L}$ *et* $pass_{y,L}$ *? Est-ce problématique ?*

Pour éviter cette situation, on veillera lors de la création de la table à rejeter toute chaine terminant par un $pass_{x,L}$ déjà présent et à recommencer avec un $pass_{x,0}$ différent.

Question 9 — *Si on choisit deux mots de passe initiaux aléatoires, obtenant les chaines* $(pass_{x,0}, pass_{x,L})$ *et* $(pass_{y,0}, pass_{y,L})$*, quelle est la probabilité d'obtenir une collision entre les mots de passe finaux (c'est à dire que* $pass_{x,L} = pass_{y,L}$*) ?*

Question 10 — *Quel est alors l'ordre de grandeur du nombre de chaines qu'on peut facilement insérer dans une rainbow table en conservant l'unicité des mots de passe finaux ?* [5]

On comprend (et observe par l'expérience) qu'au delà de cet ordre de grandeur, le programme de création de la rainbow table aura de plus en plus de difficultés à trouver des valeurs de $pass_{x,0}$ qui ne créent pas de collision.

On va donc créer plusieurs rainbow tables (notons R le nombre de tables), chacune contenant N chaines, toujours de longueur L. On continue d'imposer l'absence de collision (c'est à dire l'unicité des $pass_{x,L}$) au sein de chaque table, mais pas entre les différentes tables. Il parait cependant judicieux de ne pas autoriser l'égalité de deux $pass_{x,0}$ même pour des tables différentes.

4. Ne laissez pas les indices compliqués vous troubler : essayez sur des exemples simples

5. On attends seulement un ordre de grandeur. Si vous ne parvenez pas à formuler une attente théorique, il est aussi possible de s'appuyer sur une observation empirique.

Question 11 — ***Implémentez tout ça****, pour l'instant avec M=6, R=10, N=100000, L=1000. Ces valeurs sont définies comme constantes pré-processeurs dans le fichier* `hash.h` *fourni et pourront ainsi être modifiées ultérieurement.*

Plus précisément, on veut écrire :

- un programme `rainbow_create` qui écrit les R tables dans des fichiers dont les noms sont donnés en arguments 1 à R (chaque table contient N lignes, chaque ligne contient un couple $pass_{x,0} pass_{x,L}$ où les deux mots de passe sont séparés par un espace). Si un R+1ème nom de fichier est donné en argument, les $pass_{x,0}$ seront lus depuis ce fichier (un par ligne), sinon (seulement R arguments donnés) ils seront générés aléatoirement.
- un programme `rainbow_attack` qui prend en arguments (dans cet ordre) les noms des fichiers contenant les R tables, le nom d'un fichier contenant une liste de hashs à attaquer (un par ligne, comme dans `crackme2023_6.txt`), et le nom du fichier où écrire les mots de passe trouvés (un par ligne, ligne vide si pas d'antécédent trouvé, de sorte que ce fichier ait le même nombre de lignes que celui contenant les hashs à attaquer).
- un programme `hash_many` qui prend en premier argument le nom d'un fichier contenant une liste de mots de passe, et écrit leurs hashs (un par ligne) dans un second fichier dont le nom est donné en second argument. Cette fonction peut servir à tester les autres.

L'ensemble des exécutables demandés doit pouvoir être compilé par un simple appel à la commande `make` sans arguments sur les ordinateurs de d'école.

Quand tout fonctionne, attaquez la liste de hashs donnée dans le fichier `crackme2023_6.txt`. Rendez avec vos sources et vos réponses un fichier `found_6.txt` contenant les mots de passe trouvés. Ce fichier devra avoir le même nombre de lignes que `crackme2023_6.txt` : la ligne n contiendra un mot de passe (si vous avez trouvé un antécédent du hash présent à la ligne n de `crackme2023_6.txt`) ou sera dans le cas contraire (si vous n'avez pas trouvé d'antécédent) laissée vide pour garder la correspondance entre les deux fichiers.

Question 12 — *Pour combien de hashs trouvez-vous un antécédent ? Cela correspond-il à vos attentes ? Commentez librement vos résultats.*

Vous pouvez essayer d'améliorer ces résultats en changeant les valeurs de R, N et L, dans les limites du raisonnable en terme de temps de calcul [6]. Si vous y parvenez, attaquez la liste de hashs donnée dans le fichier `crackme2023_7.txt` qui correspondent maintenant à des mots de passe de M=7 lettres, et rendez alors un fichier `found_7.txt` au même format que précédemment.

Recherche documentaire

Question 13 — *Quelles techniques couramment implémentées de nos jours permettent de se prémunir contre de ce genre d'attaques ?* [7]

6. Vous pouvez utiliser les ordinateurs de l'ensimag, mais restez raisonnable, pensez à la planète...
7. On parle ici de techniques qui permettraient de se prémunir *spécifiquement* contre une recherche d'antécédents de hashs par le biais d'une base de données pré-calculée