ÉCOLE POLYTECHNIQUE DE LOUVAIN

UCLouvain

LSINF 1252 - Systèmes informatiques - 2019

Password cracker

Auteurs

Camberlin	Merlin	0944-17-00
Pisvin	Arthur	3711-17-00

Dernière modification le : 10 avril 2019

Professeur :
Bonaventure O.

Assistant :
De Coninck Q.

Etapes du projet :

1. Architecture à haut niveau
2. Implémentation de l'architecture
3. Ecriture des tests unitaires
4. Remise du rapport
5. Review de deux autres projets

Chapitre 1

Consignes

1.1 Objectif

A partir d'une liste de mots de passe hashés, on souhaite inverser les hashs générés par la fonction `SHA-256`. Parmi la liste de mot de passe originiale, une première sélection devra au préalable être effectuée pour ne sélectionner que les mots de passes ne contenant soit le plus grand nombre de voyelles, soit le plus grand nombre de consonnes. En outre, les mots de passe à l'origine des hash sont constitués uniquement d'**au plus 16 lettres minuscules**.

En pratique, pour trouver l'inverse d'un hash,

- On génère une panoplie de candidats,
- On calcule le hash de chaque candidat,
- On compare le hash généré à celui qu'on essaye décrypter. Si les deux hashs sont identiques, alors on a trouvé le mot de passe originel.

1.2 Fonctionnement de la fonction de hash `SHA-256`

La fonction de hash `SHA-256` génère un hash de 32 bytes (256 bits). La représentation textuelle d'un hash est une concaténation des représentations hexadécimales de chaque byte. Chaque groupe de deux caractères représente un byte du hash sous forme hexadécimale. Ainsi les 256 bits retournés par la fonction de hash s'écrivent avec 64 caractères hexadécimaux.

Il est possible d'obtenir le hash de n'importe quel string avec la ligne de commande suivante dans le terminal.

```
1 echo -n monString | sha256sum
2 dfa8adabf25ccc139116c78ed7b9e3aed15cf5ec55e59a07f95c73d4a5a240e8
```

Listing 1.1 – Commande et retour de la fonction de hash `SHA-256` pour "*monString*"

1.3 Spécification

1.3.1 Exécution

L'éxecutable de notre fichier doit s'appeler `cracker` et doit pouvoir s'utiliser comme suit :

```
1 ./cracker [-t NTHREADS] [-c] [-o FICHIEROUT] FICHIER1 [FICHIER2 ... FICHIERN]
```

Les arguments entourés de [ ] sont optionnels.

- L'argument `-t` spécifie le nombre de calcul à effectuer. Le nombre doit être un entier positif non nul. Si l'argument n'est pas spécifié, alors le nombre de threads vaut pas défault 1.

- L'argument `-c` spécifie que le critère de sélection des mots de passe se base sur le plus grand nombre d'occurence de consonnes. Si l'argument n'est pas spécifié, alors le critère de sélection est le plus grand nombre de voyelles.
- L'argument `-o` indique que la liste des mots de passe candidats doit être écrite dans le fichier `FICHIEROUT` passé en argument. Si l'argument n'est pas spécifié, alors la liste des candidats est écrite par défault sur la sortie standard.
- L'argument `FICHIER1 [FICHIER2 ... FICHIERN]` spécifie au minimum un nom de fichier contenant les hashs des mots de passe. Si l'argument ne spécifie pas d'autres fichiers, alors `FICHIER1` est l'unique fichier contenant les hashs à décrypter.

A titre d'exemple, l'exécution suivante exécute `cracker` avec **4** threads avec le critère de sélection des mots de passe basé sur le nombre d'occurences de **consonnes** avec comme fichier d'input binaire `passwords.bin`.

```
./cracker -t 4 -c passwords.bin
```

A second titre d'exemple, l'exécution suivante exécute `cracker` avec **1** thread avec le critère de sélection des mots de passe basé sur le nombre d'occurences de **voyelles** avec 3 fichiers d'input binaires nommés `pass1.bin`, `pass2.bin`, `pass3.bin`.

```
./cracker pass1.bin pass2.bin pass3.bin
```

1.3.2 Format des fichiers d'input

Chaque fichier d'input sera de type binaire et contiendra les hashs de mots de passe sous forme **binaire**. Sa taille vaudra **toujours un multiple de 32 bytes**. Pour rappel, un hash est stocké sur 32 bytes donc les 256 premiers bits représenteront le premier hash.

1.3.3 Helper

La fonction inversant le hash nous est donnée. Sa signature est la suivante :

```
bool reversehash (const uint8_t* hash1, char* res, size_t len);
```

- Le paramètre `hash1` est un pointeur vers un tableau de 32 bytes représentant un hash SHA-256.
- Le paramètre `res` est un pointeur vers un string où sera écrit l'inverse du hash s'il est trouvé.
- Le paramètre `len` indique la longueur maximale de l'inverse.

La valeur de retour est `true` ou `false` respectivement si l'inverse du hash est trouvé ou pas.

Pour chaque hash,

1.3.4 Sélection des mots de passe

Pour chaque hash, votre programme doit effectuer les deux opérations suivantes :

1. Appeler la fonction reversehash (**coûteux**).
2. Décider si le mot de passe est un candidat.

Pour qu'un mot de passe soit considéré comme candidat, il faut que son nombre d'occurences de consonnes (ou de voyelles) soit supérieur ou égal à ceux des candidats actuels.

1.3.5 Format de sortie

Après avoir traité tous les hashs, votre programme doit écrire une ligne par mot de passe des candidats sur la sortie standard (stdout).

1.3.6 Compilation du programme

Votre projet doit fournir un `Makefile` permettant de compiler votre programme. Le `Makefile` doit contenir les cibles suivantes :

- `make` Produit l'exécutable `cracker` contenant votre programme.
- `make tests` Lance les tests fournis avec le projet.
- `make all` Produit l'exécutable `cracker` et lance les tests.
- `make clean` Supprime tous les fichiers binaires (exécutables, fichiers objets, fichiers intermédiaires,...)

Le programme doit être compilé avec les drapeaux `-Wall -Werror`

1.3.7 Tests

Votre projet doit également fournir une suite de tests montrant le bon fonctionnement de votre programme.

1.4 Rapport

Votre soumission doit contenir un rapport de 4 pages maximum contenant au minimum les informations suivantes :

- Architecture haut niveau du programme
- Choix de conceptions/ d'implémentation que vous désirez mettre en avant qui peut se distinguer des autres projets.
- Stratégie des tests
- Evalutation **quantitative** (nombres et/ou graphes) montrant le bon fonctionnement de votre programme (parallélisation).

1.5 Soumission

La remise de l'implémentation complète du projet est fixée en **S 12**. La soumission se fera via Inginious. Vous devez remettre une archive **ZIP** nommée `projet_numGroupe_nom1_nom2.zip` et devra respecter les formats suivants :

/ Racine de l'archive

Makefile Le `Makefile` demandé

src/ Le répertoire contenant le code source du programme

tests/ Le répertoire contenant les fichiers utiles aux tests

rapport.pdf Le rapport

gitlog.pdf La sortie de la commande `git log -stat`

Chapitre 2

Architecture à haut niveau

2.1 Strucutures de données (représentation des mdps, etc...)

Les mots de passes hashés seront au fûr et à mesure lu et enregistré dans un tableau de bits avant d'être inversé.

Après inversion, cette fois, les mots de passes en clair seront stockés sous forme de tableau de caractères.

Après comparaison, les mots de passe auront toujours la forme d'un tableau de caractères et seront affichés sur la sortie standard ou seront enregistrés dans un fichier.

2.2 Grandes étapes

L'idée de base est d'utiliser deux producteurs-consommateurs.

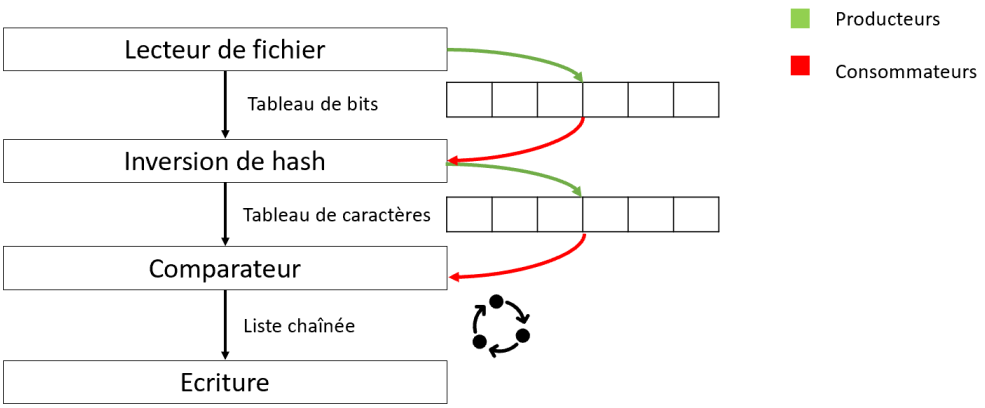

FIGURE 2.1 – Blocs fonctionnels

2.2.1 Lecture de fichier

Les fichiers en entrée peuvent avoir différentes origines. Ils peuvent provenir :

- Soit d'un dispositif de stockage (un disque dur,...)
- Soit du réseau

Concernant la lecture, un thread par **type** de fichier sera exécuté pour sélectionner les mots de passes sous forme binaires par 32 bytes (ou 256 bits). Au fûr et à mesure, les mots de passes sélectionnés seront introduits dans le premier producteur-consommateur. Derrière la structure du producteur-consommateur, se cache un tableau de hashs. Sa taille sera au préalable fixée à deux fois le nombre de threads pour garantir des conditions optimales.

Remarques

1. Il est important de souligner que la taille des fichiers en entrée ne peut pas être prédite à l'avance dans tous les cas.
2. On souhaite exécuter un thread par fichier.Dans la situation de deux fichiers sur disque dur, ce choix d'implémentation nuirait aux performances temporelles du programme. En effet, comme l'emplacement en mémoire des fichiers sur le disque durs sont différents, utiliser un thread par fichier reviendrait à déplacer successivement la tête de lecture vers l'emplacement du fichier 1 et ensuite vers l'emplacement du fichier 2. Or, le déplacement d'une tête de lecture dans différents secteurs d'un disque dur est une opération coûteuse qui prend un temps de l'ordre de la milliseconde.

   Un choix plus judicieux serait d'exécuter un thread par **type** de fichier. Dans dans la situation de deux fichiers sur disque durs, un seul thread ne serait exécuter tandis que dans le situation d'un fichier sur disque dur et un sur réseau, dans ce cas deux threads seraient utilisés.

2.2.2 Inversion

L'opération `reverse` est l'opération coûteuse de notre programme. C'est cette opération la plus gourmande temporellement. Par conséquent, dans l'idéal, il faudrait que cette fonction tourne en permanence en lui donnant en permanence des inputs.

L'inversion est le consommateur du lecteur de fichier. Elle consiste en l'appel parallèle de la fonction `reverse` avec plusieurs threads. Le nombre de threads exécutant la fonction `reverse` parallèlement dépend du nombre de threads indiqué dans la commande.

Une fois les mots de passes traduits, cette section devient un nouveau producteur de tableaux de caractères.

2.2.3 Comparateur

Le comparateur est le consommateur de l'inversion. Une fois les mots de passes traduits, ils sont stockés dans une structude de la forme d'une liste chaînée triée selon la valeur de retour de notre fonction comparaison.

Remarque Une liste chaînée est préférée car ce type de structure ne nécessite pas de connaître à l'avance la taille nécessaire. Au contraire, pour la création d'un tableau et donc son allouement en mémoire, il est nécessaire de connaître sa taille à l'avance (chose que nous ne connaissons pas dans notre projet..) On ne sait pas déterminer le nombre de mots de passe qui seront candiats.

La fonction comparaison compare le mot de passe avec la liste actuelle de candidats pour décider si oui ou non le mot passe actuel est un candidat suivant le nombre d'occurence de voyelles ou de consonnes.

2.2.4 L'écriture

L'écriture est la dernière étape du programme qui écrit la liste de candidats sur la sortie standard ou dans un fichier selon la commande introduite.

2.3 Types de threads entrant en jeu (quelle(s) tâche(s) ils réalisent ?)

Pour l'implémentation, des threads `Posix` seront utilisés. Leur tâche sera d'effectuer la fonction `reverse` en parallèle avec d'autres threads.

2.4 Méthode de communication entre les threads

Pour coordonner les threads entre eux et éviter qu'ils ne violent la zone critique, deux mutex seront utilisés pour chaque producteur-consommateur pour compter le nombre de slots disponibles et ceux remplis.

2.5 Informations communiquées entre les threads

A priori, les threads de calcul ne devraient pas avoir d'informations à communiquer entre eux.

Chapitre 3

Points d'attentions

3.1 Consignes

- Le nombre d'arguments est variable de l'exécutable.
- Ne pas spécifier des arguments lors de l'exécution implique l'utilisation des valeurs par défault.

3.2 Questions

- Est-ce que les mots de passe à l'origine des hash peuvent contenir des chiffres ou les mots de passes originels se limitent à une combinaisons d'au plus 16 lettres ?