Projet : passe ton hack d'abord !

Algorithmique et Programmation 2

Lucas Vade Grelet & Adam Sibe

1. Réponses au hacker

1.1. Est-il raisonnable de tenter une attaque en force brute ?

Pour une attaque brute-force, le nombre de possibilités dépend de la longueur des mots de passe et du jeu de caractères.
Si l'on considère des mots de passe entre 6 et 12 caractères avec 94 caractères ASCII possibles, les possibilités totales sont données par :

$$\text{Possibilités totales} = \sum_{n=6}^{12} 94^n$$

En calculant cette somme, nous obtenons environ :

$$481 \times 10^{15} \quad \text{possibilités.}$$

Tester un tel volume est irréaliste avec les ressources classiques d'un hacker.
Par conséquent, une attaque brute-force complète n'est pas raisonnable

1.2. Peut-on tenter des attaques en force brute partielles ?

une attaque brute-force partielle est bien envisageable en se basant sur des sous-ensembles:

- On dispose de la liste french_passwords_top20000, qui contient les mots de passe français les plus couramment utilisés.
- Ces mots de passe sont en clair. Il suffit de les hacher à l'aide de la fonction fournie hash_password, puis de les comparer aux hachés présents dans les fuites de données.

Cette facon est efficace pour les utilisateurs ayant choisi des mots de passe faibles ou souvent utilisé (comme azerty ou 123456).

1.3. Existe-t-il d'autres pistes pour casser tout ou partie des mots de passe avec les données dont on dispose ?

- **Réutilisation des mots de passe :** Tester si un même login, présent dans différentes fuites, utilise le même mot de passe haché.
- **comparaison entre les logins :** Identifier les logins présents dans plusieurs fichiers et comparer leurs mots de passe hachés pour trouver des correspondances.

2. Explications d'utilisation

Ce rapport décrit les étapes nécessaires pour analyser les données des fuites et tenter de décrypter des mots de passe.

Chaque partie est indépendante, mais elles doivent être exécutées dans l'ordre pour garantir la cohérence des résultats.

L'utilisation des fonctions d'écriture permet de générer des fichiers lisibles pour une analyse plus aisée des résultats.

Les explications détaillées permettent une exécution pas à pas des commandes pour atteindre les objectifs fixés.

2.1. Partie 1 : Fusionner les données

Objectif

Fusionner les informations des fichiers correspondant à plusieurs fuites d'une même application tout en éliminant les doublons (même login et même mot de passe).

Commandes à exécuter

1. Charger le fichier correspondant :

```
#use "./partie1/partie1.ml";;
```

2. Initialiser les données avec :

```
let (depensetout, slogram, tetedamis, depensetouthache) = init_sheet();;
```

Fonctions d'écriture optionnelles

Pour rendre les résultats plus lisibles, vous pouvez exporter les données dans des fichiers texte :

```
write_list_to_file_generic("outputP1/depensetout.txt", depensetout, format
write_list_to_file_generic("outputP1/slogram.txt", slogram, format_fuite()
write_list_to_file_generic("outputP1/tetedamis.txt", tetedamis, format_fuit
```

2.2. Partie 2 : Analyse des logins

Objectif

1. Déterminer si un même login est présent dans plusieurs fuites.
2. Identifier si les mots de passe associés à ces logins sont identiques.

Commandes à exécuter

1. Charger le fichier correspondant :

```
#use "./partie2/partie2.ml";;
```

2. Regrouper tous les logins dans une seule liste :

```
let (logindepensetout, loginslogram, logintetedamis, alllogins) = init_logi
```

3. Déterminer les occurrences par login :

```
let login_occurence = check_occurence_of_login(alllogins, depensetout, slog
```

4. Identifier les logins avec mot de passe identiques dans plusieurs bases de données :

```
let login_and_mdp_in_different_db = find_login_pwd_in_multiple_db();;
```

Fonctions d'écriture optionnelles

Vous pouvez exporter les résultats pour une lecture plus aisée :

```
write_list_to_file_generic("outputP2/login_occurence.txt", login_occurence
```

2.3. Partie 3 : Recherche des mots de passe hachés communs

Objectif

Déterminer si un même mot de passe haché est présent dans plusieurs fuites et savoir à quels logins il est associé.

Commandes à exécuter

1. Charger le fichier correspondant :

```
#use "./partie3/partie3.ml";;
```

2. Rechercher les mots de passe communs :

```
let shpim = search_hashed_passwords_in_multiple_dbs();;
```

Fonctions d'écriture optionnelles

Exporter les résultats dans des fichiers pour faciliter leur lecture :

```
write_list_to_file_generic("outputP3/shared_hashed_passwords.txt", shpim, f
```

2.4. Partie 4 : Décryptage des mots de passe

Objectif

Utiliser la liste des mots de passe en clair pour tenter de décrypter les mots de passe hachés.

Commandes à exécuter

1. Charger le fichier correspondant :

```
#use "./partie4/partie4.ml";;
```

2. Charger les mots de passe en clair depuis le fichier fourni :

```
let mdp_clair = mdp_from_file "./outils/french_passwords_top20000.txt";;
```

3. Hacher les mots de passe en clair :

```
let hash_mdp_clair = hach_db_mdp(mdp_clair);;
```

4. Trouver les mots de passe décryptés :

```
let (depensetout_password_decode, slogram_password_decode, tetedamis_passwo
    init_decode_password(hash_mdp_clair, mdp_clair);;
```

Fonctions d'écriture optionnelles

Exporter les résultats pour chaque base de données :

```
write_list_to_file_generic("outputP4/depensetout_passwords.txt", depensetou
```

3. Organisation du travail

Pour mener à bien ce projet, nous avons réfléchi ensemble à la manière de structurer et de réaliser chaque partie. Voici comment nous avons réparti et organisé le travail :

Réflexion globale

- La réflexion sur la question du hacker a été effectuée à deux.
- La réflexion pour chaque partie a été réalisée en collaboration, à l'exception de la partie 3, où la réflexion a été menée par Adam seul.

Répartition des tâches

- **Lucas Vade Grelet :**

  - partie1 et testpartie1.
  - partie2 et testpartie2.
  - partie4 et testpartie4.
  - main et rapport.

- **Adam Sibe :**

  - partie3 et testpartie3
  - recherche des complexité des fonction appellé dans le main

Suivi et coordination

Nous avons mis en place une feuille de route pour documenter les avancées de chacun. Cependant, nous avons oublié d'indiquer systématiquement qui a écrit quoi.

Collaboration

Grace a notre répartition des tâches bien définie, chaque réflexion et discussion autour des parties a été menée à deux, sauf pour la partie 3, où Adam a travaillé de manière autonome.

Cette organisation nous a permis de progresser efficacement tout en partageant nos idées pour assurer la cohérence globale du projet.

4. Utilisation de données extérieures

4.1. Documentation OCaml

Avant qu'Éric Andrés souligne la nécessité de ne pas utiliser le fonctionnel avancé et les options avancées du module List (fold left, iter, map ,...) , nous avions consulté la documentation officielle d'OCaml :

- https://ocaml.org/manual/5.2/api/List.html

Suite à cette remarque, nous avons retiré les fonctions utilisant du fonctionnel avancé, à l'exception de celles qui lisent et écrivent dans les fichiers texte.

4.2. Assistance avec ChatGPT

Nous avons également consulté ChatGPT dans les cas suivants :

- Lorsqu'une erreur survenait et qu'on ne comprenez vraiment pas sont origine.
- La fonction d'écriture dans un fichier texte.
- Confirmer nos résultats concernant le calcul des possibilités pour l'attaque brute-force.

4.3. Installation des modules OCaml

Pour installer les modules base64 et cryptokit, nous avons utilisé la documentation officielle d'opam :

- https://opam.ocaml.org/doc/Manual.html

5. Complexité des fonctions et des appels principaux

Cette section détaille la complexité en temps et en espace des principales fonctions et appels utilisés dans chaque partie du projet.

5.1. Partie 1 : Initialisation des données

- `let (depensetout, slogram, tetedamis, depensetouthache) = init_sheet()`

  Temps : $O(n^2)$
  Justification : La fonction unique_list parcourt chaque élément de la liste (n) et appelle is_element_in_db qui parcourt la base de données (n).
  Espace : $O(n)$
  Justification : Le résultat est une liste de taille $O(n)$.

- `write_list_to_file_generic("outputP1/depensetout.txt", depensetout, fo`

  Temps : $O(n)$
  Justification : La fonction parcourt les n éléments de la liste.
  Espace : $O(1)$
  Justification : Les éléments sont traités un par un, sans stockage supplémentaire.

5.2. Partie 2 : Analyse des logins

- `let (logindepensetout, loginslogram, logintetedamis, alllogins) = init`

  Temps : $O(n^2)$
  Justification : La fonction merge_no_duplicates utilise unique_list, dont la complexité est $O(n^2)$.
  Espace : $O(n)$
  Justification : Les listes intermédiaires et fusionnées occupent $O(n)$ en espace.

- `let login_occurence = check_occurence_of_login(alllogins, depensetout,`

  Temps : $O(n)$
  Justification : La fonction parcourt chaque login et vérifie son occurrence dans les bases (3 bases au total).
  Espace : $O(n)$
  Justification : Le résultat est une liste des occurrences, de taille $O(n)$.

- `let login_and_mdp_in_different_db = find_login_pwd_in_multiple_db();;`

  Non évaluée : La fonction est laissée en commentaire.

5.3. Partie 3 : Recherche des mots de passe hachés communs

- `let shpim = search_hashed_passwords_in_multiple_dbs();;`

  Temps : $O(n(m+l))$
  Justification : Chaque mot de passe (n) de depensetout est comparé aux bases slogram (m) et tetedamis (l). La recherche des logins associés augmente la complexité.
  Espace : $O(n)$
  Justification : Les résultats sont stockés dans une liste de taille $O(n)$.

5.4. Partie 4 : Décryptage des mots de passe

- `let mdp_clair = mdp_from_file "./outils/french_passwords_top20000.txt"`

  Temps : $O(n)$
  Justification : Lecture de n mots de passe dans un fichier.
  Espace : $O(n)$
  Justification : Tous les mots de passe sont stockés dans une liste.

- `let hash_mdp_clair = hach_db_mdp(mdp_clair);;`

  Temps : $O(n)$
  Justification : Chaque élément (n) est hashé en $O(1)$.
  Espace : $O(n)$
  Justification : Tous les mots de passe hachés sont stockés dans une liste.

- `write_list_to_file_generic("./outputP4/hash_mdp_clair.txt", hash_mdp_c`

  Temps : $O(n)$
  Justification : Écriture de n éléments dans un fichier.

- `let (depensetout_password_decode, slogram_password_decode, tetedamis_pa`
  `init_decode_password(hash_mdp_clair, mdp_clair);;`

  Temps : $O(n)$
  Justification : Appels successifs à check_db_hach, filtrer, et transform_all_tuple pour n éléments.
  Espace : $O(n)$
  Justification : Les résultats intermédiaires et finaux sont stockés dans des listes de taille $O(n)$.

Conclusion sur la complexité

- La complexité en temps varie entre $O(n)$ et $O(n^2)$ selon les opérations effectuées, avec des points critiques dans l'utilisation de unique_list.
- La complexité en espace est principalement $O(n)$, car la majorité des données sont stockées dans des listes proportionnelles à la taille des bases.