Laboratoire 2 : Cracking md5

Auteurs: Kevin Auberson, Vitória Cosmo

Introduction

Le but central de ce projet est le développement d'une application capable de déchiffrer des hash MD5 pour récupérer les mots de passe associés. Le logiciel initial qui nous a été fourni présentait des limitations en termes de performances, principalement en raison de l'absence de concurrence. Notre objectif principal était d'améliorer de manière significative ces performances en mettant en œuvre des calculs parallèles grâce à l'utilisation de threads.

Fonctionnalités du logiciel

- Calcul du hash MD5 d'un mot de passe.
- Recherche du mot de passe correspondant à un hash MD5 donné.
- Utilisation de threads pour accélérer le processus de recherche.
- Utilisation de la programmation concurrente pour améliorer la performance de l'algorithme.

Choix d'implémentation

Afin d'aborder le problème de manière optimale, nous avons ajouté des éléments dans les fichiers *mythread.h* et *mythread.cpp*, tout en effectuant les modifications nécessaires à la fonction *startHacking*.

Notre principal objectif était d'assurer la création, le lancement, et la gestion des threads directement à l'intérieur de cette fonction.

Puis, lors de la création de chaque thread, celui-ci exécute la fonction *workThread* qui s'occupera alors de la recherche du mot de passe correspondant au hash qui lui est passé.

- **Création de threads**

  À l'intérieur de la fonction *startHacking*, nous générons et lançons les threads requis pour le travail concurrent. Ils sont stockés dans un tableau de pointeur de threads de type objet *PcoThread*.

- **Partage d'informations**

  Pour que chaque thread accomplisse sa tâche de manière indépendante, nous avons pris soin de partager les paramètres nécessaires. Nous passons par référence aux threads les paramètres suivants: un pointeur vers l'objet *ThreadManager* qui appelle la fonction *startHacking*, hash à rechercher, le sel, le jeu de caractères.

- **Communication entre les threads**

  La variable *passwordCracked* est introduite pour gérer la synchronisation entre les threads. Lorsqu'un thread trouve le mot de passe, il met à jour *passwordCracked*, ce qui permet aux autres threads de s'arrêter.

La variable *result* est utilisée pour stocker le mot de passe trouvé. L'utilisation de cette variable partagée permet à chaque thread d'être capable d'écrire le résultat final. Elle est initialisée vide pour le cas échéant où l'on ne trouve pas le mot de passe.

La variable *nbToComputeGlobal* représente le nombre de possibilités total de mots de passes à calculer selon le nombre de caractères du mot de passe ainsi que la taille du set de caractères autorisés. Cette variable est utilisée uniquement pour mettre à jour la barre de progression au sein de la fonction *workThread*.

- **Répartition du travail**

  Une partie essentielle de notre implémentation consiste en une répartition efficace du travail entre les threads. Chaque thread exécute en parallèle la fonction *workThread*, c'est ici qu'elle teste les combinaisons de mots de passe dans le but de trouver le mot de passe correspondant au hash MD5 fourni.

  En effet, chaque thread explore le tableau *currentPasswordArray* de manière stratégique pour garantir une distribution équilibrée du travail. Chacun d'entre eux parcourt les éléments de ce tableau en fonction de son *ID* thread et l'incrémentation se fait avec le nombre total de threads. Le nombre d'itérations de chaque thread est determiné par la variable *nbToComputePerThread*. Celle-ci est calculé en fonction du nombre total de threads. Lorsqu'un thread atteint la fin de son sous-ensemble de combinaisons, il incrémente l'index actuel d'où il se trouve dans le tableau de manière à prendre en charge de nouvelles combinaisons.

  L'objectif est d'assurer que chaque thread traite un sous-ensemble différent de combinaisons de mots de passe, répartissant ainsi efficacement la charge de travail et contribuant ainsi à une progression plus rapide de la recherche globale.

- **Terminaison des threads**

  Nous avons implémenté une fonctionnalité de terminaison des threads qui nous a permis d'obtenir le résultat sans attendre la fin de l'exécution de tous les threads. Dès qu'un thread trouve le mot de passe correspondant au hash MD5, la variable statique *passwordCracked* est définie à true, le mot de passe est enregistré dans la variable *result*, et les threads sont arrêtés. Cela a contribué à économiser des ressources en évitant l'exécution inutile de threads supplémentaires. Une fois le résultat retourné, on lance une boucle qui joint chaque thread du vector de threads.

Tests effectués

Nous avons effectué une série de tests pour vérifier les performances et le bon fonctionnement de notre logiciel de crackage de hash MD5. Chaque test était conçu pour évaluer différents aspects de l'application.

Temps de recherche en millisecondes pour le même mot de passe selon le nombre de threads

Nombre de threads	Temps de recherche [ms]
3	11128
10	22422
20	29237

Dans ce tableau, nous avons représenté le temps de recherche du hash du mot de passe à 4 lettres "*KKKK*" en fonction du nombre de threads utilisés dans notre programme.

En observant les données mesurées, nous constatons que le temps de recherche augmente de façon significative à mesure que le nombre de threads augmente, ce qui semble aller à l'encontre de notre intuition.

Cependant, lorsque le nombre de threads augmente considérablement, il est possible que les temps deviennent plus longs. Cette augmentation peut s'expliquer par le fait que la gestion de multiples threads ajoute une certaine complexité. Le programme doit gérer la concurrence entre les threads et effectuer des changements de contexte, ce qui peut entraîner des temps d'attente et des ralentissements.

Temps de recherche en millisecondes avec 3 threads en fonction de la taille du mot de passe

Taille du mot de passe [lettres]	Temps de recherche [ms]
5	1,6
10	6
15	7,3

Ce tableau présente une analyse du temps de recherche en fonction de la taille du mot de passe associé à un hash généré. Les observations démontrent qu'à mesure que la taille du mot de passe augmente, le temps de recherche nécessaire pour le retrouver augmente également. En d'autres termes, les mots de passe plus longs demandent davantage de temps pour être identifiés, ce qui est en accord avec nos prévisions et attentes.

Temps de recherche en millisecondes avec 3 threads pour différents mots classés par ordre alphabétique

Classement	Mot	Temps de recherche [ms]
1	aaaa	4
2	zzzz	7430
3	JJJJ	10758
4	ZZZZ	14972
5	***a	233

Pour ce test, nous avons pris une liste de mots triés par ordre alphabétique puis nous avons calculé le temps de recherche pris par l'algorithme. En général, nous avons observé une tendance où le temps de recherche augmente à mesure que nous progressons dans le dictionnaire, ce qui est cohérent avec la manière dont nous parcourons *currentPasswordArray* (par ordre alphabétique). Cependant, il y a une valeur qui se distingue comme étant particulièrement surprenante, celle associée au mot "***a."

Test de précision de recherche

Nous avons réalisé plusieurs autres tests avec des mots aléatoires en plus des ceux précédemment mentionnés. Lorsque nous avons entrepris de rechercher des mots de passe situés en fin de dictionnaire et comportant plus de 4 caractères, nous avons été confrontés à des temps d'attente considérablement

prolongés, atteignant parfois plusieurs minutes. Dans certains cas, notamment lors de la recherche de mots de passe tels que "FFFGG" ou composés uniquement de caractères spéciaux ("*****"), nous n'avons pas réussi à les trouver du tout.

Conclusion

Ce projet illustre la manière dont l'utilisation du parallélisme peut avoir un impact significatif sur les performances d'une application. Il offre également une démonstration de l'importance de la synchronisation et du partage de données entre les threads pour garantir un fonctionnement correct.

En implémentant le parallélisme à l'aide de threads, nous nous attendions à améliorer considérablement les performances de l'application, réduisant ainsi le temps nécessaire pour cracker un hash MD5.

Toutefois, certains tests nous ont montré les limitations et vulnérabilités de notre algorithme notamment pour les cas de mots de passe à plus de 4 lettres ou placés en fin de dictionnaire. Il est aussi important de noter que l'exécution de l'application peut varier d'un ordinateur à l'autre. Elle dépend de plusieurs paramètres, dont le nombre de coeurs du processeur disponibles. Ces facteurs peuvent influencer la capacité de la machine à effectuer des calculs en parallèle.

Pour améliorer le comportement de notre programme, il serait bien d'étudier la manière dont nous avons géré les ressources partagées. Cela pourrait inclure une synchronisation plus fine des threads, la réduction des temps d'attente ou même l'optimisation de la manière dont les données partagées sont accédées.