Rapport Laboratoire 2 PCO: Cracking MD5

Auteurs: Benoit Delay, Eva Ray

Description des fonctionnalités du logiciel

Le programme que nous avons développé permet de cracker un hash md5 par bruteforce, afin de récupérer un mot de passe. Une application de base nous a été fournie mais elle souffrait de problèmes de performances car elle n'était pas multi-threadée. Pour améliorer son efficacité, nous avons donc intégré une fonctionnalité de multithreading.

Afin d'utiliser le programme, l'utilisateur doit fournir un hash md5, la longueur du mot de passe cherché et le nombre de threads qu'il souhaite utiliser. Le sel est également une option que l'utilisateur peut fournir, si nécessaire.

Lors de l'exécution du programme, une barre de progression est affichée et se remplit graduellement. Cette bare représente le pourcentage de mots de passe potentiels qui ont déjà été testés.

Choix d'implémentation

L'objectif principal de l'implémentation du multi-threading était d'améliorer les performances de l'application, tout en gardant une cohérence dans les variables partagées.

Notre stratégie pour répartir la charge de travail entre les threads est la suivante. Nous considérons le nombre total des mots de passe possibles et nous divisons par le nombre de threads fourni par l'utilisateur. Le résultat représente donc le nombre de mots de passe que chaque thread va devoir tester.

```
nbToCompute = intPow(charset.length(), nbChars);
// On ajoute 1 pour le cas où nbToCompute n'est pas divisible par nbThreads
nbToComputePerThread = (nbToCompute / nbThreads)
                       + (nbToCompute % nbThreads ? 1 : 0);
```

Pour implémenter cette stratégie, nous avons décidé de créer une classe *`TaskThread`* qui contient tous les attibuts et méthodes nécessaires à la définition de la routine qu'un thread doit mettre en place pour procéder au cracking de mots de passe. Nous avons opté pour une classe plutôt qu'une simple fonction globale, afin de profiter des capacités d'encapsulation du C++.

C'est la méthode *`TaskThread::taskHacking`* qui contient la routine de cracking. La première étape est de trouver le mot de passe à partir duquel le thread doit commencer à tester les hash. Pour cela, la méthode reçoit le paramètre *`nbToCompute`* qui représente le nombre de mots de passe que le thread doit tester (i.e. le nombre de hash à générer). A partir de ce paramètre et de l'id du thread, on déduit la position dans le dictionnaire du mot de passe à partir duquel commencer.

```
long long unsigned startPosition = threadId * nbToCompute;
```

A partir de ce mot de passe et jusqu'à avoir atteint *nbToCompute* itérations, le thread va donc calculer le hash du mot de passe et le comparer avec celui fournit par l'utilisateur. S'ils sont identiques, l'attribut *hasFoundPassword* de l'objet *TaskThread* va être actualisé à *true* et le contenu de l'attibut *passwordFound* va être actualisé avec la mot de passe trouvé.

```
if (currentHash == hash){
    passwordFound = currentPasswordString;
    hasFoundPassword = true;
}
```

C'est la classe *ThreadManager* qui s'occupe de gérer les threads. Nous avons ajouté un attribut privé qui permet de stocker les threads lancés.

```
std::vector<std::unique_ptr<PcoThread>> threads;
```

C'est à l'intérieur de la méthode *ThreadManager::startHacking* qu'on va gérer les threads. On commence par lancer les threads et, grâce au getter *isPasswordFound* des objets *TaskThread*, on peut donner l'ordre à tous les threads de s'arrêter lorsqu'un mot de passse a été trouvé. On récupère alors le mot de passe stocké dans l'attribut *passwordFound*, qui sera la retour de la fonction *ThreadManager::startHacking*.

```
if (task->isPasswordFound()) {
    passwordFound = task->getPasswordFound();

    // Demande aux threads de s'arreter
    for (const auto &thread : threads) {
        thread->requestStop();
    }
}
```

La dernière fonctionnalité qui doit être gérée est la barre de progression. Nous avons opté pour un attribut *nbHashComputed* dans la classe *TaskThread* qui contient le nombre de hash calculés par un certain thread. Dans la méthode *ThreadManager::startHacking*, on calcule alors le nombre total de hash calculés, en additionant le nombre de hash calculés par chaque thread. Enfin, on utilise cette information pour mettre à jour la barre de progression.

```
if ((nbTotalHashComputed % 1000) == 0) {
    incrementPercentComputed((double)1000 / nbToCompute);
}
```

Tests effectués

Les tests sont effectués sur la machine virtuelle du REDS, à qui nous avons alloué 4 coeurs. Chaque cas de test a été lancé trois fois et les résultats affichés dans les tableaux ci-dessous sont la moyenne de ces tentatives. Les résultats bruts des tests sont disponibles dans l'annexe à la fin de ce rapport. Il est bon de noter que le résultat des tests dépendent de la machine sur lesquels ils sont lancés mais il nous permettent quand même de dégager une tendance globale.

Pour préparer notre terrain de tests, nous avons d'abord lancé le cracking de "aaaa" qui est le premier mot de passe du dictionnaire. Nous constatons que le warm up time est de 0 à 2 [ms]. Ce n'est donc pas à cette étape que nous perdons en temps d'exécution.

nombre threads	temps [ms]
1	0.67
2	1.33
4	1.67
8	1.33

Test 1: Mot de passe de longueur 4 sans sel

Commençons avec un mot de passe de longueur 4, sans ajouter de sel. Choisissons "abcd" qui est donc plutôt en début de dictionnaire.

nombre threads	temps [ms]
1	370.7
2	564
4	1001
8	1636.33

Faisons un autre test avec le mot de passe "ABCD" qui est plutôt en milieu de dictionnaire.

nombre threads	temps [ms]
1	3442
2	4447
4	3295
8	2952.67

Faisons un autre test avec le mot de passe "!!!!" qui est plutôt en fin de dictionnaire.

nombre threads	temps [ms]
1	7620.33
2	4895
4	3991
8	2499

Nous constatons que la vitesse d'exécution du programme dépend fortement du placement du mot de passe à trouver dans l'espace des mots de passe possibles. En effet, avec un seul thread, le meilleur cas est un mot de passe en début de dictionnaire et le pire cas un mot de passe en fin de dictionnaire.

Dans le meilleur cas, nous perdons du temps en lançant plusieurs threads. Cela est probablement dû au temps de création, de coordination, de changements de contexte dû aux préemptions et de gestion des threads. Il y a problablement une piste d'amélioration de ce côté-là.

Dans le pire cas , il est logique qu'ajouter plus de threads améliore les performances. En effet, bien que le mot de passe reste en fin d'espace même après séparation entre les threads, chaque thread a bien moins de mots de passe à parcourir. Il atteint donc le dernier plus vite.

Dans le cas moyen , on observe aussi une amélioration des performances en ajoutant des threads, dans la pupart des cas. Cependant, cela dépend encore une fois de comment l'espace des mots de passe est réparti entre les threads. Si le mot de passe cherché se retrouve en fin d'un sous-ensemble, alors il sera trouvé moins vite que s'il se retrouve au début. C'est d'ailleurs ce qu'il se passe dans le cas avec deux threads.

En conclusion, nous avons pu amméliorer de manière significative les performances dans la pire cas, améliorer légèrement les performances du cas moyen. Ceci se fait au prix de petites pertes de performance dans le meilleur cas, qui reste toutefois très correcte.

Tous les comportements énumérés ci-dessus sont des comportements attendus. Les résultats des tests sont cohérents.

De plus, lors d'autres essais sur notre VM, nous avons parfois constaté que, dès 4 threads, les performances commencent à diminuer ou stagner. Cela est dû à la configuration de notre VM, à qui nous avons pu allouer 4 coeurs. Cela implique qu'à partir de 4 threads, il commence à y avoir beaucoup de préemptions, ce qui ralentit le programme. De manière générale, on constate que les performances plafonnent lorsqu'on commence à ajouter plus de threads que le nombre disponible sur la machine. C'est un comportement attendu.

Test 2: Mot de passe de longueur 4 avec sel

Testons le mot de passe "abdc" avec le sel "xy".

nombre threads	temps [ms]
1	367.67
2	556.33
4	892.33
8	1775

Nous constatons que les résultats sont extrêmement similaires à ceux observés pour ce même mot de passe lors du test 1. C'est tout à fait logique puisque, une fois que nous avons précisé le sel utilisé à notre programme, le problème revient en fait à cracker un mot de passe de longueur 4 à nouveau. C'est un comportement attendu.

Test 3: Mot de passe de longueur 5 sans sel

Testons avec un mot de passe de longueur 5, sans ajouter de sel. Choisissons "abcde" qui se trouve plutôt en début de dictionnaire.

nombre threads	temps [ms]
1	33428.33
2	48639
4	90511.33

Nous constatons que le temps d'exécution du programme est bien plus élevé qu'avec un mot de passe de longueur 4. C'est logique car avec une longueur de 4, nous testons au plus (66)^4 = 18'974'736 mots de passe, alors qu'avec une longueur de 5, nous testons au plus (66)^5 = 1'252'332'576 mots de passe. En général, la technique de crackage de mots de passe bruteforce est très lente pour de longs mots de passe, ces résultats ne sont donc pas étonnants.

Ceci dit, nous constatons que plus il y a de threads, plus l'exécution du programme est lente, ce qui est le même comportement que pour le test 1 et donc cohérent pour notre programme.

Test 4: Barre de progression

- La barre se remplit de manière cohérente en fonction compte du nombre de threads. Par exemple, lors du cracking du mot de passe "abcd" du test 1, la barre de progression est entre 4 et 5 pourcent lorsque le mot de passe est trouvé avec un thread et entre 8 et 9 pourcent avec deux threads. C'est le comportement attendu puisqu'on a testé deux fois plus de mots de passe.
- La barre affiche bien 100% si aucun mot de passe n'a été trouvé.
- La barre se remplit bien de manière linéaire et régulière.

Conclusion

Nous avons pu constater que le temps que met notre programme à cracker un mot de passe dépend fortement de la position de celui-ci dans le dictionnaire. Ceci étant dit, notre programme réagit comme attendu en fonction de cette position. Nous avons également observé que l'ajout d'un sel n'affectait pas les performances du programme, et que notre programme était peu efficace pour carcker des mots de passe longs.

De manière générale, nous avons pu constater que donner plus de threads à un programme n'implique pas toujours que ses performances s'amméliorent. En effet, cela dépend fortement des capacités de la machine sur lequel le programme tourne, de la préemption des threads et des algorithmes mis en place.

Enfin, nous pouvons dire que nous avons réussi à amméliorer les performances de l'application de base.

Annexe

mot de passe	nb threads	temps 1 [ms]	temps 2 [ms]	temps 3 [ms]	moyenne [ms]
aaaa	1	2	0	0	0,67
	2	0	2	2	1,33
	4	1	1	3	1,67
	8	1	1	2	1,33
abcd	1	354	397	361	370,67
	2	679	485	528	564,00
	4	939	1141	923	1001,00
	8	1555	1668	1686	1636,33
	32	1665	1907	2043	1871,67
ABCD	1	3404	3512	3410	3442,00
	2	4510	4421	4410	4447,00
	4	3307	3221	3357	3295,00
	8	2799	3320	2739	2952,67
!!!!	1	7130	7957	7774	7620,33
	2	4805	5013	4867	4895,00
	4	4482	3839	3652	3991,00
	8	2342	2721	2434	2499,00
xyabcd, sel = xy	1	371	368	364	367,67
	2	551	591	527	556,33
	4	862	859	956	892,33
	8	1860	1748	1717	1775,00
abcde	1	31856	34058	34371	33428,33
	2	54182	42461	49274	48639,00
	4	94403	87421	89710	90511,33