Julien Mühlemann et Cristhian Ronquillo

Labo 2 – Cracking MD5

Description des fonctionnalités du logiciel :

"Cracking MD5" est un logiciel conçu pour décrypter un hash md5 afin de récupérer un mot de passe. À l'origine, il fonctionne en monothread, mais il est nécessaire de le convertir en multithreads pour augmenter la vitesse de décryptage du mot de passe.

Choix d'implémentation :

Nous avons apporté des modifications à la fonction startHacking en déplaçant la section de craquage de hash dans une nouvelle fonction nommée computing. Cette dernière est dédiée au craquage du hash MD5. Afin d'accélérer le processus de craquage, la fonction computing sera exécutée sur plusieurs threads. Chaque thread, opérant indépendamment, se verra attribuer une portion spécifique du mot de passe à traiter. Cette attribution est basée sur le nombre total de threads, assurant ainsi que chaque thread commence là où le précédent s'est arrêté et s'arrête avant le début de la portion assignée au suivant.

Nous avons deux index ; startIndex, endIndex et chunkSize qui est le nombre de hash possibles. Les index sont calculés avant le lancement de chaque thread de la manière suivante:$pour\ k\ de\ 0\ \text{à}\ n-1\ thread$:

$$startIndex = k \cdot chunkSize\ + 1$$

$$endIndex = (k+1) \cdot chunkSize$$

Pour le premier startIndex nous le forçons à 0 et pour le dernier endIndex nous plaçons le nombre de possibilités comme borne de fin. En effet, ceci n'est pas à proprement parler une borne de fin puisque nous commençons à 0. Toutefois nous décrémentons cette valeur lors de son traitement dans la fonction de conversion de base afin de correctement obtenir la borne de fin.

```
77      for (size_t k = 0 ; k < nbThreads ; ++k){
78
79          //Calcul de chaque borne de début et de fin pour chaque thread.
80          long long unsigned int startIndex = (k == 0) ? 0 : k * chunkSize + 1;
81          long long unsigned int endIndex = (k == nbThreads - 1) ? nbToCompute : (k+1) * chunkSize;
```

Cette méthode évite la redondance dans le calcul des hashs et optimise le temps nécessaire pour le craquage. De plus, nous avons introduit une fonction supplémentaire qui convertit une valeur décimale en une base spécifique ; dans notre cas, la base est 66, correspondant au nombre total de caractères possibles dans un mot de passe.

```
 8   //Fonction de convertion de chaque valeur en décimale sur une base N déterminé
 9   //Selon la taille du charset passé en paramètres.
10   QVector<int> toBaseN(unsigned long value, const QString charset, int nChar) {
```

Nous nous sommes assuré également qu'aussitôt qu'un thread trouve le hash correspondant, alors les autres threads ne doivent pas continuer inutilement. Ceci se fait grâce à la condition dans la boucle while.

```
78          *pwd = currentPasswordString;
79          return 0;
80      }
81      while (currentIndex <= endIndex && *pwd == "") {
82
83          for (i=0 ; i < nbChar ; i++){
```

Aussi, lorsque un thread a trouvé un hash valide, il l'affecte au pointeur *pwd et termine immédiatement son execution.

```
92
93          if (currentHash == hashToBeFound){
94              *pwd = currentPasswordString;
95              return 0;
96          }
97
```

StartHaking ---------- ---------- --------- return password

…… PcoThread(1) -> Computing ----------Finish ……

…… PcoThread(2)-> Computing ---------- password Found --Finish …….

…… PcoThread(3) -> Computing ---------- Finish ……

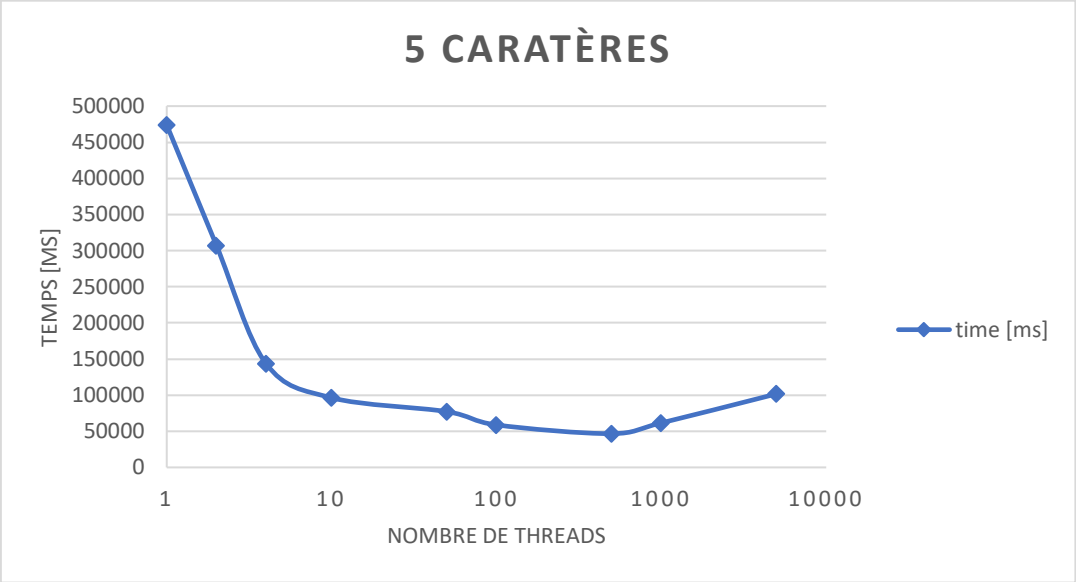
…… PcoThread(N) ->Computing ----------Finish……

Tests effectués :

Lors de nos tests, nous avons observé que le comportement de l'ordonnanceur introduit une variabilité qui peut soit accélérer, soit ralentir le processus de recherche du mot de passe dans certains cas. Cette inconsistance est due à la manière dont l'ordonnanceur attribue les ressources du CPU aux différents threads, influençant ainsi le temps nécessaire pour trouver le mot de passe. Pour illustrer plus clairement ce phénomène, nous présenterons des tableaux détaillant les résultats des tests et les valeurs que nous avons examinées. Nous avons effectué une moyenne sur plusieurs observations concernant le temps que prend notre programme pour trouver le mot de passe, car l'ordonnanceur peut biaiser les mesures.

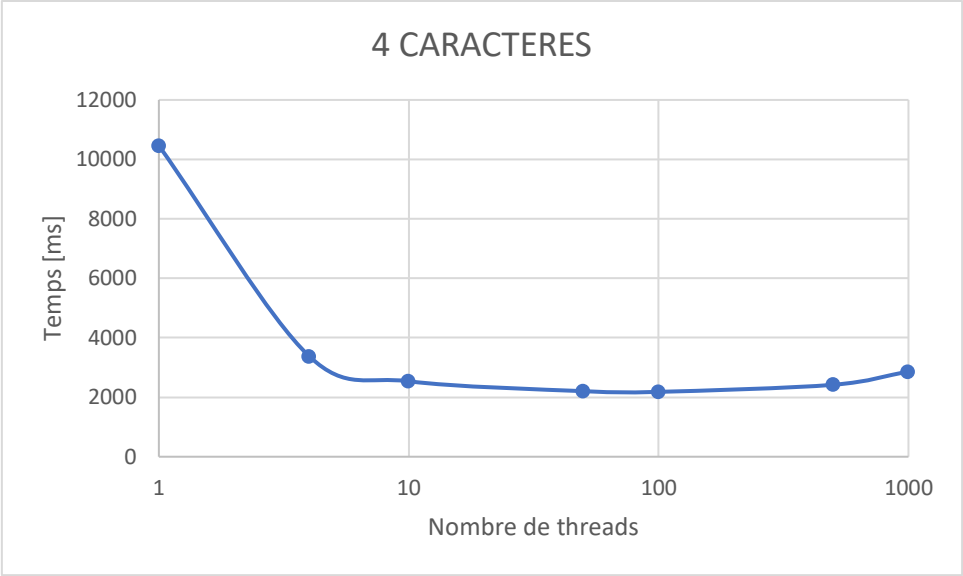
Mot de passe à 5 caractères (E*1az) :

mot de passe	E*1az
threads	time [ms]
1	473868
2	307362
4	143705
10	96331
50	77200
100	58801
500	46713
1000	61569
5000	101691



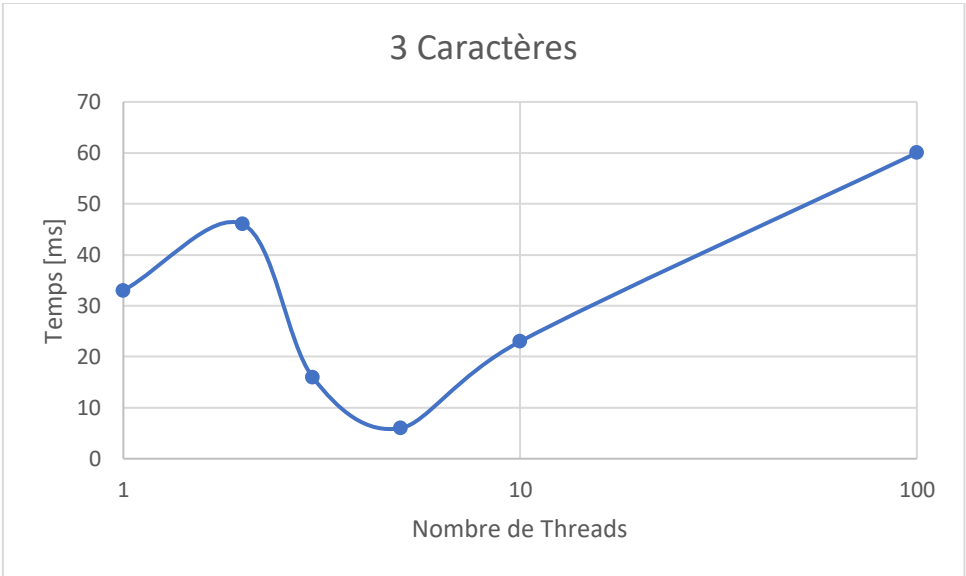
Mot de passe à 4 caractères (A3zb) :

mot de passe	A3zb
threads	time [ms]
1	10462
4	3377
10	2538
50	2205
100	2182
500	2419
1000	2866



Mot de passe à 3 caractères (z$A) :

mot de passe	z$A
threads	time [ms]
1	33
2	46
3	16
5	6
10	23
100	60



Julien Mühlemann et Cristhian Ronquillo

Conclusion :

Sur les graphiques, nous n’observons pas la manière dont nous divisons la portion à traiter pour chaque thread mais influence considérablement le temps de craquage du hash. Cela est particulièrement visible sur le graphique à 3 caractères. Toutefois, cette observation est surtout pertinente pour les mots de passe très courts. À l'inverse, un mot de passe de plus de 3 caractères démontre l'intérêt d'avoir un nombre de threads plus élevé. En effet, l'espace à traiter étant plus conséquent, la portion est mieux répartie entre les threads. Il est à noter qu'il existe un nombre optimal de threads en fonction du nombre de caractères. Lorsque ce seuil est dépassé, augmenter le nombre de threads risque tout simplement de prolonger le temps nécessaire pour trouver le hash correspondant au mot de passe.