Rapport du projet Calcul Parallèle sur GPU
Cassage de hash MD5 par recherche exhaustive par CPU/GPU

Jean-Baptiste Gaeng

February 27, 2019

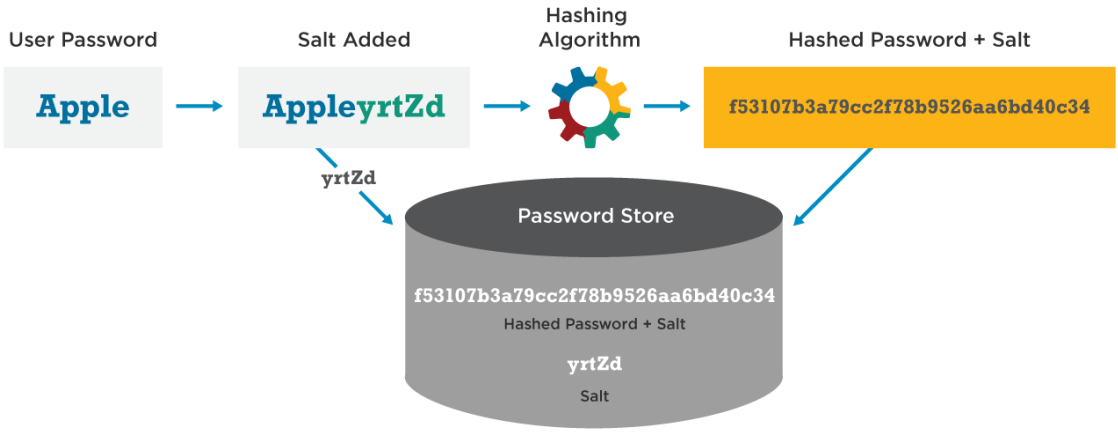

Table des matières

1 Explications théoriques du problème

Cette section s'intéresse à présenter de manière théorique le problème, en s'appuyant principalement sur la thèse de M. Martijn Sprengers "GPU-based password cracking" disponible à l'adresse suivante : https://www.ru.nl/publish/pages/769526/thesis.pdf ainsi que sur le rapport éponyme de M. Marcus Bakker et Roel van der Jagt disponible à l'adresse suivante : https://delaat.net/rp/2009-2010/p34/report.pdf

1.1 Introduction et présentation du problème

Aujourd'hui, de nombreux services et sites internet s'appuient sur des systèmes d'authentification avec nom d'utilisateur et mot de passe pour authentifier les utilisateurs. Pour garantir la sécurité des données des utilisateurs, la quasi-totalité de ces services ne stockent pas les mots de passe des utilisateurs en clair mais stockent un condensé (= un hash) du mot de passe obtenu après l'application d'une fonction de hashage. Aussi, quand un utilisateur s'authentifie sur l'un d'un tel service, un condensé du mot de passe rentré par l'utilisateur est calculé et comparé au condensé stocké par le service.

En raison des propriétés mathématiques de cette fonction de hashage, il est très difficile de retrouver le mot de passe clair à partir du hash. Cependant, certaines propriétés mathématiques sont affaiblies (cf 1.3) en raison de la manière dont les individus choisissent leurs mot de passe. En effet, un mot de passe idéal serait composé de caractères aléatoires mais les individus tendent à choisir des mots de passe courts, ou pas assez complexes, afin de les retenir plus facilement. Afin de pallier à cette perte de sécurité, les systèmes augmentent la complexité des hash stockés en faisant appel à des schémas de hashage cryptographique. Cependant, même de tels systèmes peuvent être mis en défaut aujourd'hui, notamment à cause de l'amélioration des temps de calculs avec l'amélioration rapides des GPU. En effet, les GPU sont très adaptés aux calculs parallèles et donc au cassage de fonction cryptographique.

1.2 Comment attaquer un système cryptographique ?

1.2.1 Les différents types d'attaques

1.2.1.1 Attaque par recherche exhaustive (brute-force)

La méthode la plus simple conceptuellement pour cracker un mot de passe consiste à rechercher toutes les possibilités. Cependant, la plupart des fonctions cryptographiques actuelles ont un très grand domaine de clés ; par exemple, la fonction de hashage cryptographique MD5 à une sortie codée sous 128 bits, ce qui donne un domaine de clefs de 2^{128}. Si on suppose que toutes les sorties sont uniformément réparties, alors la recherche n'est pas réalisable avec le hardware actuel.

Ainsi, pour qu'une attaque de ce type soit réalisable, l'attaquant doit avoir au minimum accès au hash du mot de passe qu'il cherche à cracker. Il cherche alors à retrouver le mot de passe clair correspondant en appliquant la fonction de hashage MD5 à une série de mots de passe clairs puis compare les hash obtenus avec le hash à cracker.

1.2.1.2 Attaque des anniversaires

Cette attaque s'appuie sur le paradoxe des anniversaires. Comme on va le voir en section 1.3, une fonction de hashage associe un domaine de taille dynamique à un domaine de taille fixe, ce qui a pour conséquence l'apparition de collision, c'est à dire l'existence d'antécédents ayant des images identiques.

Dans ce type d'attaque, l'attaquant génère des mot de passe aléatoires et l'applique à une fonction de hashage donnée jusqu'à ce que deux entrées (antécédents) aient la même sortie.

1.2.1.3 Attaque par compromis temps-mémoire

Comme on l'a vu dans le paragraphe 1.2.1.1, certaines méthodes demande beaucoup (trop) de temps de calcul pour être réalisable en pratique. Ainsi, cette attaque vise à réduire le temps de calcul pour casser un système crytographique. Pour ce faire, l'attaquant peut pré-calculer une

partie de la recherche exhaustive et la stocker en mémoire. L'attaque consiste donc à trouver le meilleur compromis entre le coût mémoire de stockage et le temps de calcul.

1.3 Qu'est ce qu'une fonction de hashage ?

1.3.1 Définitions et propriétés

Une fonction de hashage est une fonction mathématique qui associe un domaine d'entrée de taille arbitraire à un domaine de sortie de taille fixe :

$$\begin{array}{rccc} H & : & Z_2^m & \rightarrow & Z_2^n \\ & & x & \mapsto & H(x) \end{array}$$

avec H la fonction de hashage, Z_2 l'ensemble 0,1, m le nombre de bits de l'entrée et n le nombre de bits de la sortie. Pour construire la fonction de hashage, on utilise des combinaisons de fonctions mathématiques, de rotations et de shifts, et des fonctions de compressions. Ceci permet de simuler une grande complexité et un résultat pseudo-aléatoire.

Les propriétés d'une bonne fonctions de hashage sont :

- **Sorties uniformément distribuées** : On aimerait bien que pour chaque antécédent, la fonction de hashage produise une image unique. Or, laplupart du temps le nombre d'antécédents est beaucoup plus grand que le nombre d'image possibles. Par le principe des tiroirs, il y a donc forcément des collisions c'est à dire des antécédents différents qui auront la même image. Pour limiter le nombre de collisions, une bonne fonction de hashage doit avoir des sorties uniformément distribuées, c'est à dire que chaque sortie doit apparaître avec une probabilité $1/2^n$ où n est le nombre de bits de la sortie.
- **Déterminisme** : Pour un antécédent donné, la fonction de hashage doit toujours donner la même sortie quelque soit le temps donné
- **Rapidité** : L'image d'un antécédent par la fonction de hashage doit être rapide à calculer, c'est à dire avoir une complexité linéaire en $O(m)$
- **Résistance à la préimage** : Cette propriété s'assure du fait que la fonction de hashage est bien une fonction non reversible.
  - Première résistance à la préimage : 2tant donnée une fonction de hashage H et un hash h, il doit être asse difficile de trouver un message m tel que $H(m) = h$
  - Seconde résistance à la préimage (aussi appelée résistance à la collision faible) : Etant donné une fonction de hashage H et un message m_1, il doit être assez difficile de trouver un message m_2 différent de m_1 tel que $H(m_1) = H(m_2)$
- **Résistance à la collision** (aussi appelée résistance à la collision forte) : Etant donné une fonction de hashage H, il doit être assez difficile de trouver deux messages différents m_1 et m_2 tels que $H(m_1) = H(m_2)$.

Dans les propriétés précédentes, "assez difficile de trouver" signifie que toute attaque doit avoir un temps assez long pour être considérée ne pas être réalisable en pratique, tant que la protection des données du système est considérée importante.

1.3.2 Le cas de la fonction de hashage MD5

MD5 pour Message Digest 5 est une fonction de hachage cryptographique prenant en entrée un message de taille arbitraire et produit en sortie un condensé (aussi appelé hash, empreinte ou digest) sur 128 bits. Par exemple, si on considère le message suivant "Bonjour" sur 7 octets, il sera représenté en hexadécimal par $426f6e6a6f7572_{16}$. De plus,

$$MD5(Bonjour) = ebc58ab2cb4848d04ec23d83f7ddf985_{16}$$

1.4 Schémas de hashage cryptographique

1.4.1 Définitions et propriétés

Comme expliqué dans la section 1.1, des schémas de hashage cryptographique sont utilisés afin de complexifier les hashs stockés sur un système donné. Un schéma de hashage cryptographique désigne un ensemble d'algorithmes, incluant une ou plusieurs fonctions de hachage.

Communément, avant que le mot de passe en clair ne soit haché, il est généralement complété par ce que l'on appelle un sel. Parce qu'un schéma de hachage de mots de passe doit être une fonction unidirectionnelle statique, un sel est une donnée aléatoire, qui peut être publique, qui est hachée avec le mot de passe afin de diminuer le risque que des mots de passe identiques donnent le même condensé après hashage. Un autre avantage d'un sel est qu'il rend les attaques par dictionnaire plus difficiles, comme on va le voir en section 1.4.2.

Un schéma de hashage crytographique ne doit donc pas être confondu avec une fonction de hashage. En fait, généralement, on applique le schéma de hashage cryptographique en incluant les fonctions de hashage afin d'augmenter la sécurité des données (hashs) stockées.

Un schéma de hashage peut être définit mathématiquement par :

$$\begin{array}{rccc} SHC & : & Z_2^m x Z_2^s & \rightarrow & Z_2^n \\ & & x & \mapsto & SHC(x) \end{array}$$

où SHC désigne le schéma de hashage cryptographique, et s le nombre de bits du sel (les autres notations sont les mêmes qu'en section 1.3). Il faut veiller à ce que m + s < n car si la taille du mot de passe en clair plus celui du sel dépasse la taille de la sortie, cela risque encore de créer des collisions supplémentaires, ce que l'on veut justement éviter.

Ainsi, ce qui est réellement stocké en machine est souvent un hash de la concaténation du mot de passe en clair, et du sel. Par exemple, dans les systèmes UNIX, le schéma de hashage cryptographique s'appelle crypt(). Les systèmes Unix stockent les mot de passe dans un fichier `/etc/shadow` de la manière suivante : $IDFonctionHashage$Sel$HashDuMDP

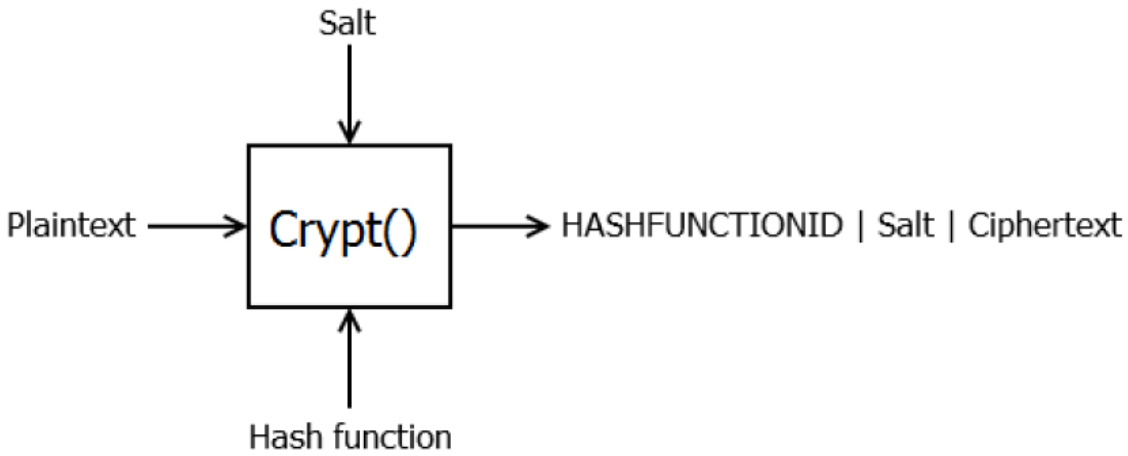

Figure 1: Représentation en "boite-noire" du schéma de hashage des sysèmes Unix

Dans le cas particulier du MD5, voici le schéma de hashage correspondant :

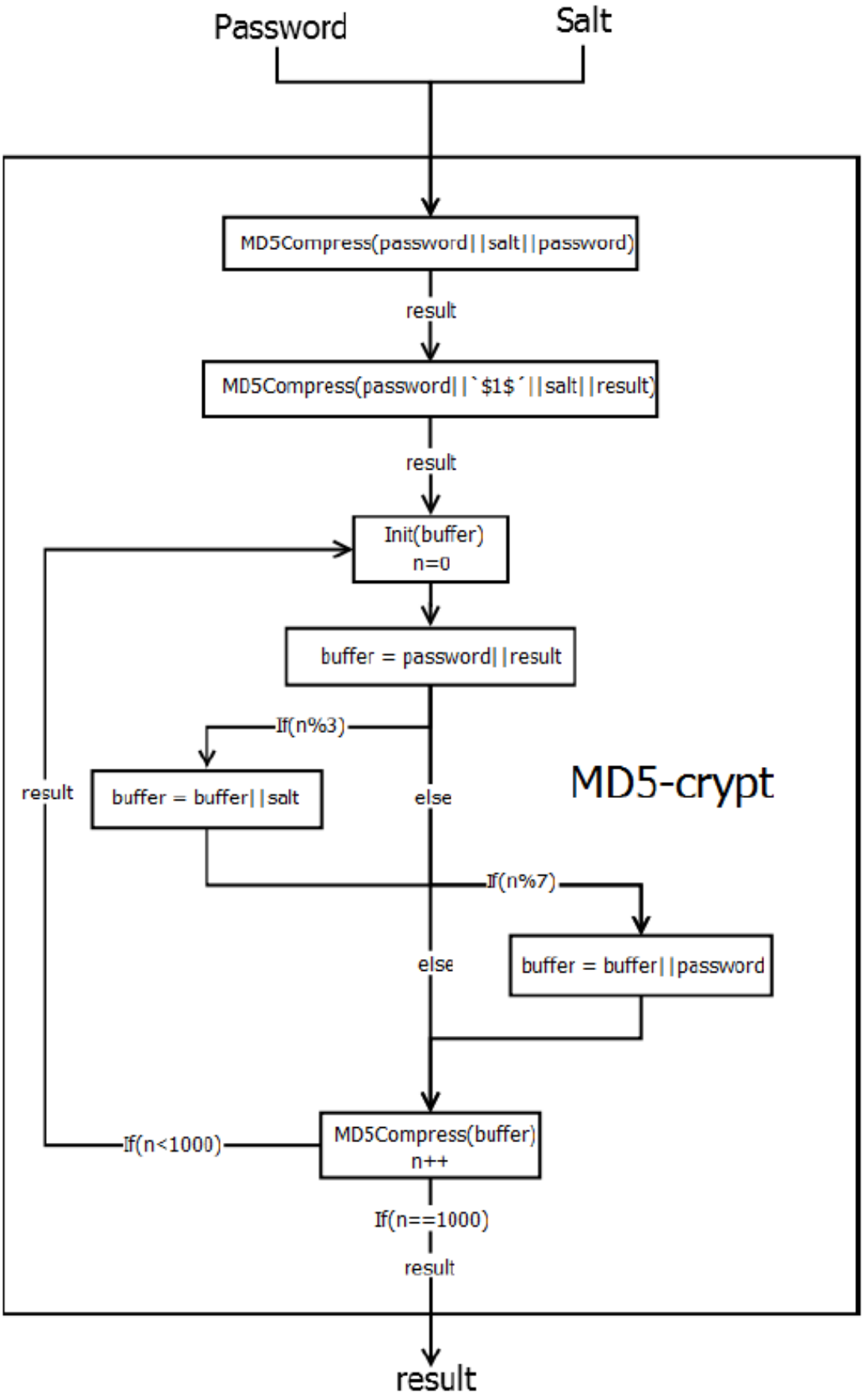

Figure 2: Représentation du schéma de hashage MD5-crypt

Les propriétés d'un bon schéma de hashage sont les suivantes :

- **Utilisation correcte des sels** : L'utilisation de sels permet comme expliqué auparavant d'avoir $H(m_1||s_1) \neq H(m_2||s_2)$ même si $m_1 = m_2$ avec s_1 et s_2 deux sels différents, et $||$ l'opération de concaténation. De plus, l'utilisation de sels rend impossible les attaques précalculées comme les attaques par dictionnaires (cf 1.4.2) ou les compromis temps-mémoire. Cependant, il faut veiller à ce que le nombre de bits du sel soit assez long. En effet, théoriquement $H(m||0)$ et $H(m||1)$ donneront des hash complètement différents en raison de l'effet boule de neige dans la fonction de hashage, mais il est alors possible de précalculer les deux possibilités de hash possibles par avance pour un m donné. Ainsi, la plupart des schémas de hashage actuels utilisent des sels codés sur 64 ou 128 bits.

- **Hashage lent** : Nous avons vu qu'une des conditions d'une bonne fonction de hashage était d'avoir une complexité linéaire par rapport au nombre de bits du message d'entrée. Par contre, pour un schéma de hashage, celui-ci doit avoir un temps de calcul assez lent pour ne pas rendre trop rapide les attaques de type brute-force. Une des méthodes pour augmenter le temps de calculs au sein d'un schéma de hashage est d'effectuer des itérations de la fonction de hashage, c'est ce que l'on appelle l'étirement de clé. La fonction de hashage $H()$ est ainsi remplacée par $H^k()$ où $H^k()$ est calculée en composant la fonction de hashage avec elle-même k fois. Cette technique permet d'augmenter par un facteur linéaire k le temps d'attaque par bruteforce.

- **Politique de renforcement de mots de passe** : Afin de complexifier les hashs produits en sortie du schéma de hashage, on peut rendre plus stricte la politique de mots de passe pour les utilisateurs. Par exemple, en obligeant les utilisateurs d'un service à choisir un mot de passe avec des caractères spéciaux, des lettres majuscules et minuscules, et des chiffres, l'entropie du mot de passe sera plus élevée et le hash en sortie du schéma de hashage plus complexe. On réduit par le même coût la probabilité de collision.

1.4.2 Stratégies d'attaques

Les principales stratégies d'attaques des schémas de hashage sont les suivantes :

- **Attaque par recherche exhaustive** : Dans ce type d'attaque, l'attaquant calcule toutes les possibilités. Si l'on note $SHC(m_1, s_1) = h_1$ avec SHC le schéma de hashage cryptographique, m_1 le mot de passe clair de l'utilisateur stocké, s_1 un sel donné, alors l'attaquant essaye toutes les possibilités de mot de passe m afin d'en trouver un tel que $SHC(m, s_1) = h_1$. L'attaquant sait alors que dans ce cas $m = m_1$. Si on suppose que les hashs produits en sortie par le SHC sont distribués uniformément, alors la probabilité de trouver le bon mot de passe en calculant toutes les possibilités est égale à 1. Cependant cette méthode est limoté par le temps de calcul, et si le domaine de mot de passe M est trop grand, ce type d'attaque n'est plus faisable. Par exemple, si l'on considère un mot de passe de taille maximale 9 pouvant être écrits avec les 94 caractères ASCII imprimables, alors la taille du domaine de mot de passe $|M| = N$ sera égal à :

$$N = \sum_{k=0}^{9} 94^k \approx 2^{59} \tag{1}$$

  Le temps de calcul prendrait 18 ans pour un ordinateur capable de calculer 1 milliard de mot de passe par secondes, avec une espérance mathématiques de 9 ans selon le principe des anniversaires, ce qui n'est pas faisable en pratique.

- **Attaque par dictionnaires** : Cette attaque utilise le fait que les individus utilisent souvent des mots de passe courant et facile à mémoriser. Ainsi, des dictionnaires de mot de passe peuvent être construits, par exemple à partir d'anciennes banques de mot de passes qui ont fuitées sur internet. Cela permet aux attaquants de limiter leur recherche en itérant sur moins de mots de passe que la recherche exhaustive.

- **Attaque par reconnaissance d'empreintes** : Ce type d'attaque utilise des modèles statistiques appris sur des banques de mot de passes, afin de retrouver des patterns ou empreintes spécifiques dans les mots de passe. Cette technique tire profit du fait que les individus doivent se souvenir de leur mot de passe d'une manière ou d'une autre, et utilise souvent les mêmes moyens mnémotechniques ou d'obfuscations pour le choix de leurs mot de passe. Exemple :

- Moyen mnémotechique : **D**on **J**uan et **G**abriel **o**nt **r**egardé **N**etflix **s**ur **l**eur **c**anapé neuf $\longrightarrow$ DJ&GorNslc9
- SuperMotDePasse $\longrightarrow$ SüP3rM0t2Pà$$e
- Exemple pattern spécifique : esxcfr (caracères proches au clavier)

Les attaquants entraînent donc des modèles statistiques prenant en compte la fréquence d'apparition des caractères, la longueur des mots de passe, et la reconnaissance de ces patterns spécifiques.

2 Implémentations CPU et GPU de cassage de hash MD5 par recherche exhaustive

Pour notre projet, nous nous sommes intéressés au cas le plus simple de crackage de mot de passe, celui de la recherche exhaustive. La fonction de hashage considérée est le MD5. Nous ne considérons pas non plus dans cette section les schémas de hashage cryptographique, mais cherchons simplement à cracker des hashs obtenu après application de la fonction de hashage MD5.

Pour mesurer la performance des différents algorithmes, on calcule le nombre de hashs/s (H/s) candidats proposés.

2.1 Hardware utilisé

Dans la suite, nous avons implémenté nos programmes et effectué nos tests sur deux machines.
La première machine (que nous allons désigner comme machine 1 dans la suite) contient un CPU Intel(R) Core(TM) i7-2670QM CPU @ 2,20 GHZ, ainsi qu'un GPU Nvidia Geforce 560M. Nous avons installé CUDA et OpenCL sur cette machine. La deuxième machine (machine 2) contient quant à elle un CPU Intel(R) Core(TM) i5-3340M CPU @ 2,70 GHZ ainsi qu'un GPU AMD FirePro M6000. C'est cette machine qui contient la distribution Kali Linux.
Les deux machines ont des GPU de prix et de performances similaires. En ce qui concerne le CPU, celui de la machine 2 est légèrement plus cher et plus récent.
Pour des spécifications détaillées, voir en Annexe A 4.

2.2 Un mot sur la génération des mots de passe et des hashs MD5

Dans le cadre de nos implémentations, nous avons écrit un script de génération de mot de passe ainsi que de leurs hashs MD5 correspondants. Les mots de passe produit sont donc plus complexes que dans la réalité car chaque caractère est choisi uniformément dans le domaine des caractères imprimables. Le temps de crackage d'un hash ainsi obtenu doit donc être vu comme un majorant du temps attendu pour cracker un mot de passe de longueur équivalente dans la réalité.

Le code du script est disponible en Annexe et dans le dossier du projet.

2.3 Implémentation CPU

2.3.1 Recherche exhaustive (brute-force) naïf

La première implémentation que nous avons faite consiste en une recherche exhaustive naïve par bruteforce. Cette implémentation permet d'avoir des valeurs de temps et de performance de référence et nous permettra de quantifier précisément l'efficacité des algorithmes parallèles plus poussés dans la suite.

Le script que nous avons écrit (disponible en Annexe et dans le dossier du projet) ouvre un fichier texte contenant les hash MD5 (un hash par ligne et au format le plus courant hexadecimal). Pour chaque hash, nous itérons sur tous les mots de passe possibles, calculons pour chacun son hash MD5, et le comparons au hash MD5 à cracker.

Voici les résultats que l'on obtient :

charset longueur (nb_caractères)	[a-z,A-Z]	[a-z,A-Z,0-9]	[a-z,A-Z,0-9,@... !]
1	$1.1 * 10^{-4}$ s	$1.3 * 10^{-4}$ s	$1.0 * 10^{-4}$ s
2	$4.7 * 10^{-3}$ s	$2.4 * 10^{-3}$ s	4.9 10-3 s
3	0.43 s	0.32 s	0.59 s
4	34.3 s	33.8 s	39.2 s
5	52,2 min	58,7 min	60,4 min

Figure 3: Tableau des temps nécéssaires au crackage des hashs, en fonction de la longueur du mot de passe en clair et du domaine de caractères. Chaque temps est calculé sur une moyenne de 10 mots de passe générés à l'aide de notre script de génération

Comme on peut le constater sur le tableau précédent, le temps de calcul varie de manière exponentielle à mesure que l'on ajoute des caractères au mot de passe.

Cela est en accord avec la formule 1, puisqu'il s'agit d'une somme géométrique de raison 94.

Ainsi, pour un mot composé de 6 cracatères, on peut s'attendre d'après nos tests à un temps de crackage d'environ

$$Temps(MDP\_6\_caracteres) = 60 * 94 = 94\, heures = 3.9\, jours$$

et pour un mot de passe de 7 caractères

$$Temps(MDP\_7\_caracteres) = 3.9 * 94 \approx 367\, jours$$

ce qui est déjà très elevé. Dès 7 caractères, la méthode du bruteforce naïf n'est donc plus utilisable en pratique.

En ce qui concerne les performances de l'algorithme, nous avons mesuré une moyenne de 950 kH/s. Cette valeur nous indique donc que pour être efficace, un algorithme de cassage de hash MD5 sur notre architecture devra proposer au minimum 950 000 hashs candidats par seconde.

2.3.2 Recherche exhaustive parallèle en utilisant OpenMP

Pour ce cas, nous avons utilisé OpenMP afin de tenter de paralléliser les calculs correspondants à la génération des hashs candidats. Le code est disponible en Annexe et dans le dossier du projet. C'est une version adaptée et améliorée d'un code qui est disponible à l'adresse suivante : https://rosettacode.org/wiki/Parallel_Brute_Force

2.3.2.1 Explication du code

Le script comprend :

- Fonction `matches` : vérifie si deux hashs convertis en tableau d'octets sont égaux
- Fonction `StringHashToByteArray` : convertit un hash au format hexadécimal (encodage des caractères supposé en ASCII) en tableau d'octets
- Fontion `printResult` : imprime le mot de passe clair du hash lorsqu'il est cracké, ainsi que le nombre de hashs générés au total (nombre de tentatives)
- Fonction `computeCombination` : fonction récursive qui parcourt toutes les combinaisons possibles pour une longueur de mot de passe donnée. Le domaine de caractères est les 95 caractères imprimables ASCII dont l'encodage va de 33 à 127.
- Fonction `main` : fonction principale qui utilise OpenMP pour paralléliser les calculs correspondants à la boucle for calculant toutes les combinaisons possibles

2.3.2.2 Utilisation de l'outil

Les commandes pour utiliser le programme sont les suivantes :

```
gcc -o bruteforce-md5-openmp bruteforce-md5-openmp.c -fopenmp -lssl -lcrypto
export OMP_NUM_THREADS=4
./bruteforce-md5-openmp
```

`OMP_NUM_THREADS` peut être optimisé selon l'architecture sur laquelle le programme tourne. Le programme contient encore quelques bugs et n'imprime pas toujours le résultat sur la sortie standard. Il faut donc parfois le lancer plusieurs fois.

En faisant une moyenne sur différents mots de passe, nous obtenons une performance moyenne sur notre machine de 7,9 MH/s.

2.3.3 Recherche exhaustive avec multiprocessing

La deuxième implémentation consiste à essayer d'utiliser du multiprocessing pour effectuer des calculs en parallèle et ainsi réduire le temps de crackage des hashs. Nous avons trouvé une implémentation disponible à l'adresse suivante : https://github.com/aaronjwood/cracker.

Le code utilise la bibliothèque `multiprocessing` de Python. En effet, on ne peut pas vraiment faire de multithreading en Python car le GlobalInterpreterLock de Python interdit l'éxécution du bytecode Python en même temps par plusieurs threads. Ainsi, le programme tente de paralléliser le travail en découpant le charset en plusieurs sous ensembles, chaque processus travaillant sur un sous ensemble différent.

Mais nous n'obtenons pas des résultats satisfaisants avec l'outil. En effet nous mesurons une performance de 39 kH/s soit une fréquence moins grande que dans notre implémentation par bruteforce naïve. Nous avons supposé que cela venait du fait que la mise en place du multiprocessing était bien plus coûteuse que le gain éventuel que cette méthode pourrait faire gagner en vitesse de calcul. Quoi qu'il en soit dans la suite nous n'avons plus utilisé le langage Python pour tenter d'implémenter du calcul en parallèle.

2.3.4 Utilisation de l'outil John the Ripper

Un autre outil que nous avons eu l'occasion de tester s'appelle John the Ripper. L'outil est disponible de base dans la distribution Kali Linux. D'après la documentation de l'outil, lorsque l'on simule une attaque par recherche exhaustive, John the Ripper peut paralléliser les calculs en utilisant OpenMP, mais aussi en utilisant du multi-processing avec l'option –fork.

Voici les commandes que nous avons utilisées pour notre attaque :

```
john --format=raw-md5 --incremental hashs.txt
```

L'argument `--incremental` permet d'effectuer une recherche exhaustive. `hashs.txt` désigne notre fichier contenant les hashs MD5 à cracker.

La sortie de l'outil nous indique le nombre de passwords par seconde. En faisant une moyenne sur quelques exécutions, nous trouvons sur notre machine une performance d'environ 28,1 MH/s. L'accélération par OpenMP permet donc d'augmenter grandement la performance de l'algorithme.

Cette performance augmente encore quand on ajoute l'option `--fork` qui permet de faire du multi-processing.

```
john --format=raw-md5 --incremental --fork=2 hashs.txt
```

Avec la commande ci-dessus, on obtient une performance en moyenne égale à 40 MH/s.

Remarque : Lors de nos tests, nous avons cependant constaté que l'outil prenait beaucoup de temps pour casser un hash provenant d'un mot de passe clair contenant des caractères spéciaux

(parfois plus de temps que notre implémentation avec du bruteforce naïf). Selon la documentation de l'outil, ceci peut venir du fait que John the Ripper ne propose pas toutes les combinaisons possibles dans l'ordre. Il sélectionne un ordre des mots de passe qui tente de maximiser le temps de cassage. Ainsi, John va par exemple d'abord essayer des mots de passe simples à 8 caractères (sans caractères spéciaux) avant les mots de passe peu probables et plus complexes de 6 caractères.

2.3.5 Utilisation de l'outil Hashcat version CPU

Hashcat est l'outil le plus avancé à l'heure actuelle pour le cassage de hash. Il supporte un grand nombre de fonction de hashage et de schémas de hashage, et permet de lancer des attaques de types très variés. Il est également installé de base sur la distribution Kali Linux.

L'outil utilise aussi OpenCL pour accélérer les calculs sur CPU. Nous ne nous sommes pas plongés dans les détails de son implémentation mais nous nous sommes contentés de tester la performance de l'outil sur notre machine avec la commande suivante :

```
hashcat -b -m 0
```

L'option `-b` permet d'effectuer un benchmark, l'option `-m 0` permet de renseigner le type de hash (0 pour MD5).

La sortie du becnhmark permet de connaître la vitesse en terme de hash/seconde :

```
Speed.#1.........: 108.4 MH/s (19.10ms) @ Accel:1024 Loops:1024 Thr:1 Vec:8
```

Les termes `Accel` et `Loops` correspondent à des réglages liées à la charge de travail sur le device considéré. Des valeurs plus élevées peuvent être plus efficaces lors de tentatives de force brute, alors que des valeurs plus basses ont tendance à être plus efficaces lors d'attaques par dictionnaires. Ainsi, on arrive à une vitesse de 108,4 MH/s.

2.4 Implémentation GPU

2.4.1 Accélération de la recherche exhaustive avec OpenCL : Oclcrack

2.4.1.1 Mise en place de l'outil

Pour ce cas, nous nous sommes basés sur une implémentation disponible à l'adresse suivante : https://github.com/sghctoma/oclcrack. Voici comment sont organisés les fichiers composants le projet :

- HashStore.cpp : Implémente la classe Hastore, qui est dédiée au stockage des hashs, à leurs comparaisons et à la mise en place du domaine de caractères.
- MD5.cl : Implémente le code kernel du programme, à savoir le calcul des hashs.
- OCLCrack.cpp : Implémente le cracker. Contient les méthodes d'initialisation du contexte OpenCL et la création des différents buffers (notamment pour le domaine de caractères et pour les hashs crackés). La fonction crack quant à elle s'occupe de boucler sur toutes les longueurs possibles de mot de passe. Pour une longueur donnée, elle calcule le nombre de hashs qui vont être générés, puis boucle sur toutes les combinaisons possibles de mot de passe, en appelant le code kernel.
- Timer.cpp ; Gère le calcul du temps pendant l'éxécution du programme
- main.cpp : Programme principal, parse les arguments du programme et crée le cracker.
- genkernelstring.sh : Script bash permettant de générer le fichier header correspondant à MD5.cl

Le site web de l'auteur n'est plus disponible mais il est toujours accessible via le Web Cache. Il est donc possible d'avoir une documentation minimale à l'adresse suivante : http://web.archive.org/web/20150911201941/http://sghctoma.extra.hu/index.php?p=entry&id=11

De plus, le code étant optimisé pour une architecture CUDA selon l'auteur, nous avons dû porter le code sous Windows afin de l'utiliser sur notre machine 1 contenant OpenCL et une carte NVIDIA. Nous avons donc ajouté quelques fichiers afin par exemple de trouver des équivalents aux méthodes Linux qui utilisent `sys/time.h`. Le projet Visual Studio se trouve dans l'archive du projet. Il contient un exécutable compilé pour notre machine mais est utilisable sur d'autres architectures également. Pour mettre en place OpenCL sous l'environnement Windows, nous avons suivi les conseils disponibles à l'adresse suivantes : https://medium.com/@pratikone/opencl-on-visual-studio-configuration-tutorial-for-the-confused-3ec1c2b5f0ca

2.4.1.2 Utilisation de l'outil

Une fois le projet compilé, nous avons utilisé l'outil à l'aide de la commande suivante :

```
oclcrack.exe --verbose --charset=abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
    ->    STUVWXYZ0123456789!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ hashs.txt
```

L'option `--charset` permet de définir le domaine de caractères, car par défaut l'outil ne prend pas en compte les caractères spéciaux. `--verbose` permet d'avoir une sortie plus détaillée notamment d'avoir le temps d'éxécution et le nombre de hashs générés.

Sur notre machine 1, on arrive à des performances moyennes de 70.1 MH/s.

2.4.2 Accélération par CUDA : BarsWF Bruteforce

Nous avons utilisé la version binaire Windows de cet outil sur la machine 1, téléchargeable sur le site de l'auteur à l'adresse suivante : https://3.14.by/en/md5/. Les fichiers sources sont quant à eux téléchargeables à l'adresse suivante : https://3.14.by/forum/viewtopic.php?f=8&t=1333&p=8907

L'éxécution simple du binaire permet de générer un hash MD5 d'un mot de passe clair aléatoire que l'outil tente de casser. Voici un exemple de sortie que l'on obtient sur notre machine :

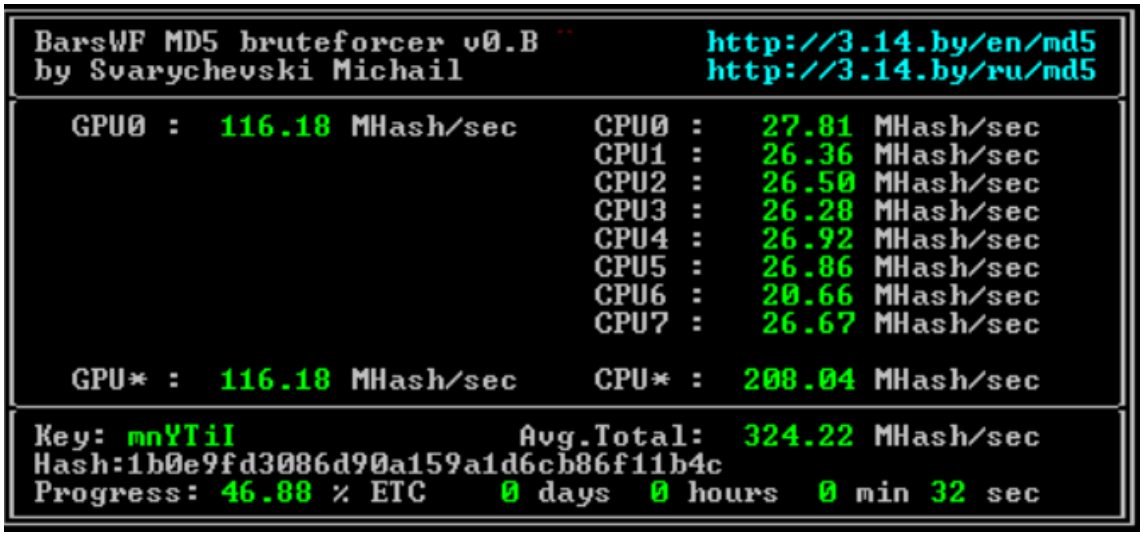

Figure 4: Un exemple de sortie obtenu avec l'outil BarsWF MD5 Bruteforcer sur la machine 1

Comme on peut le constater sur la figure précédente, BarsWF partitionne bien le calcul en parallèle sur les 8 coeurs du CPU et également sur le GPU. L'outil atteint une fréquence moyenne de 324 MH/s.

2.4.3 Utilisation de Hashcat version GPU

Nous avons également utilisé Hashcat en mode GPU pour comparer l'efficacité de cet outil avec les autres. Nous avons utilisé les mêmes commandes que dans la section 2.3.5.

La sortie obtenue est la suivante :

```
Speed.#1.........: 1021.1 MH/s (79.84ms) @ Accel:256 Loops:128 Thr:256 Vec:1
```

Performance de l'algorithme : 1021.1 MH/s.

3 Comparaisons globale des performances des outils CPU et GPU

3.1 Graphe des performances pour les différents outils

Les figures suivantes récapitulent les résultats obtenus :

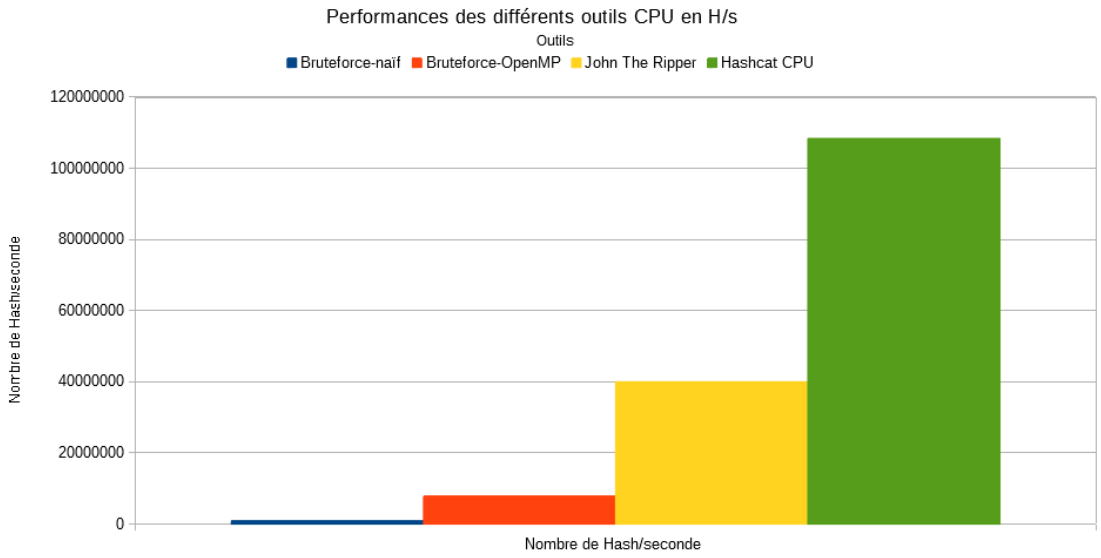

Figure 5: Graphe récapitulatif des performances pour les implémentations CPU

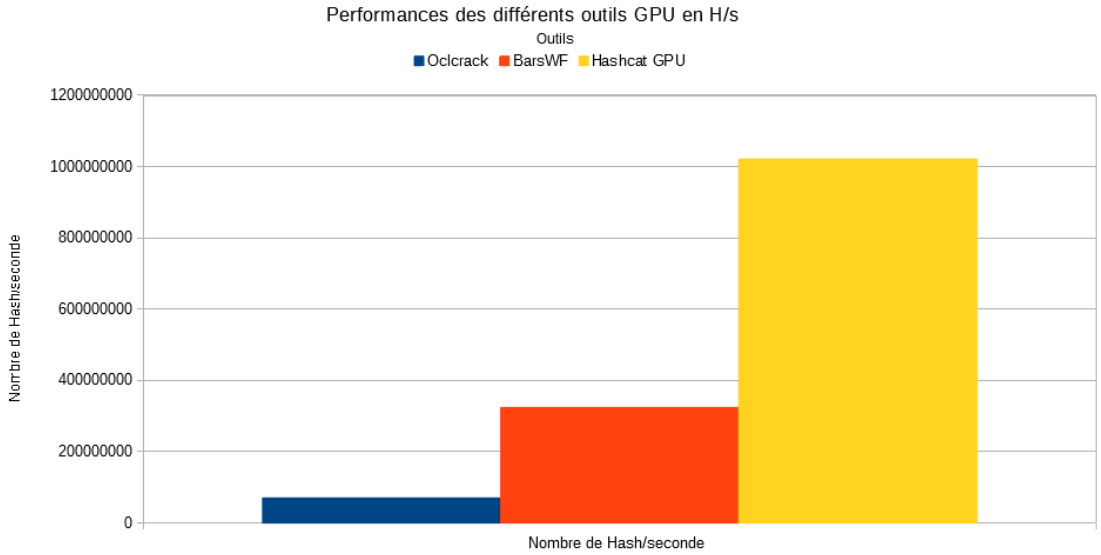

Figure 6: Graphe récapitulatif des performances pour les implémentations GPU

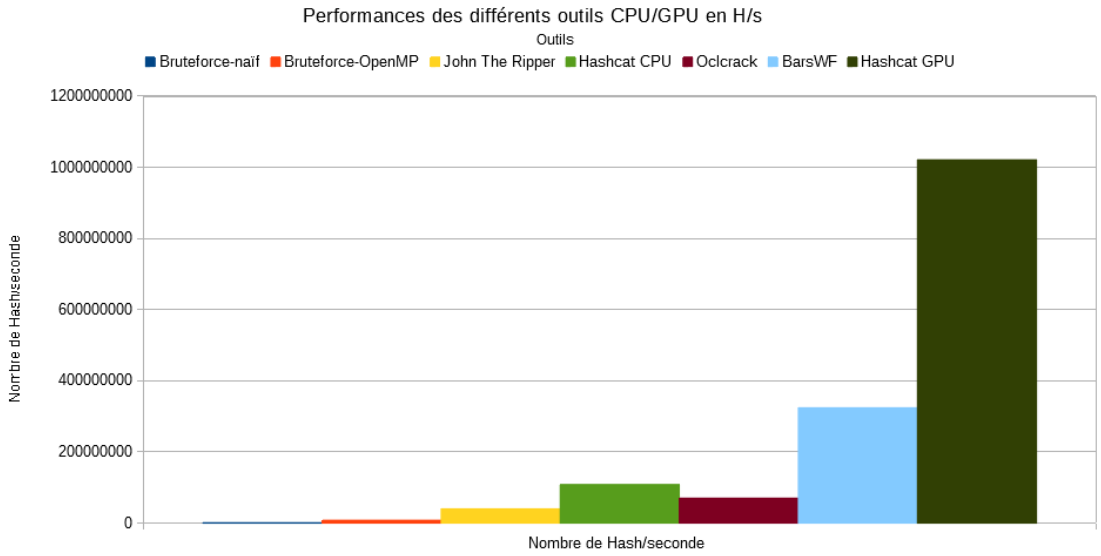

Figure 7: Graphe récapitulatif des performances pour les implémentations CPU/GPU

	CPU				GPU		
Algorithmes/Outils	Bruteforce-naïf	Bruteforce-OpenMP	John The Ripper	Hashcat CPU	Oclcrack	BarsWF	Hashcat GPU
Performance moyenne en hash/sec (H/s)	950 000	7 900 000	40 000 000	108 400 000	70 100 000	324 000 000	1 021 100 000
Gain de performance % Bruteforce-naïf	x1	x8,3	x42	x114	x73	x341	x1074

Figure 8: Tableau récapitulatif des performances pour les implémentations CPU/GPU et gain de performance (donc de temps) des différents outils par rapport au bruteforce naïf

Pour les implémentations CPU, l'utilisation de OpenMP pour pralléliser le calcul des hashs permet de multiplier déjà les performances par 8. Mais ce sont les implémentations OpenCL des outils John The Ripper et Hascat qui permettent vraiment de gagner en performance. L'outil le plus rapide Hashcat CPU parvient à générer 10,8 MH/s soit 42 fois plus que l'algorithme de bruteforce naïf.

En ce qui concerne les implémentations GPU, elles sont toutes bien plus performantes que les implémentations CPU mise à part Oclcrack qui est moins rapide que Hashcat CPU. Peut-être que cela vient du fait que Oclcrack ne parvient pas à utiliser tous les coeurs du GPU pour effectuer ses calculs. En ce qui concerne les deux autres outils GPU restants, BarsWF utilisant l'accélération CUDA arrive à multiplier le nombre de hash générés par seconde par 341 par rapport au bruteforce naïf. L'implémentation qui atteint les meilleures performances sur notre machine 2 est Hashcat GPU avec une fréquence de 1021 MH/s soit 1074 fois plus que le bruteforce naïf.

Avec de telles performances sur des machines peu coûteuses comme les nôtres, les implémentations GPU arrivent à cracker des hashs de mots de passe complexes (94 caractères impirmables ASCII) de 6 et 7 caractères en quelques minutes.

4 Conclusion

Après avoir introduit théoriquement le sujet, nous avons vu au cours de ce projet les différentes manières d'implémenter le cassage de hash MD5 par recherche exhaustive : d'abord avec le CPU naïvement ou en parallélisant les calculs avec OpenMP ou OpenCL, puis avec le GPU en parallélisant les calculs à l'aide de OpenCL et CUDA.

La méthode d'attaque par recherche exhaustive est censée être la plus coûteuse et la plus longue. Pourtant avec les performances que l'on obtient, sur des machines dont le hardware a plus de 5 ans maintenant, il est possible de casser facilement des hashs de mots de passe complexes de 6-7 caractères. Même si aujourd'hui le MD5 est considéré comme peu fiable cryptographiquement car sensible aux collisions, il reste tout de même utilisé dans certaines applications.

De plus, avec les avancées rapides de la puissance de calcul des GPU et selon la loi de Moore, même les fonctions de hashage récentes comme le SHA-512 seront de plus en plus vulnérables d'ici quelques années. Il est donc important de choisir ses mots de passe avec un nombre de caractères élevé, dans l'idéal supérieur à 8, avec lettres majuscules et minuscules, chiffres et caractères spéciaux.

ANNEXE A : Spécifications précises du hardware utilisé

4.1 Machine 1

4.1.1 Spécifications générales

- CPU : Intel(R) Core(TM) i7-2670QM CPU @ 2,20 GHZ
- GPU : NVIDIA Geforce GTX 560M
- Système d'exploitation : Windows 7

De plus, nous avons installé OpenCL et CUDA sur la machine.

4.1.2 Spécifications Intel Core i7 2670QM

Propriétés générales

- Nombre de coeurs : 4
- Nombre de threads : 8
- Fréquence de l'horloge (GHz) : 2,20
- Fréquence Max Turbo (GHz) : 3,10
- Smart Cache (MB) : 6
- Jeu d'instructions : 64-bit
- Extensions du jeu d'instructions : Intel® AVX

Propriétés de la mémoire CPU

- Taille de mémoire maximale (GB) : 32
- Type de mémoire : DDR3 1066/1333
- Nombre de canaux mémoire : 2
- Bande Passante mémoire max (GB/s) : 21,3

4.1.3 Spécifications Nvidia Geforce GTX 560M

Propriétés générales

- Nombre de coeurs CUDA : 192
- Fréquence de l'horloge graphique (MHz) :
- Fréquence de l'horloge processeur (MHz) : 1550
- Texture fill rate (milliard/sec) : 36,8

Propriétés de la mémoire GPU

- Fréquence de l'horloge mémoire (MHZ) : 1250
- Memory Interface : GDDR5
- Memory interface width : Jusqu'à 192 bit
- Bande Passante mémoire (GB/s) : Jusqu'à 60

4.2 Machine 2

4.2.1 Spécifications générales

- CPU : Intel(R) Core(TM) i5-3340M CPU @ 2,70 GHZ
- GPU : AMD FirePro M6000
- Système d'exploitation : Windows 7 et Debian (Kali Linux) sur une VM

De plus, nous avons installé OpenCL sur la machine.

4.2.2 Spécifications Intel Core i5-3340M

Propriétés générales

- Nombre de coeurs : 2
- Nombre de threads : 4
- Fréquence de l'horloge (GHz) : 2,7
- Fréquence Max Turbo (GHz) : 3,4
- Smart Cache (MB) : 3
- Jeu d'instructions : 64-bit
- Extensions du jeu d'instructions : Intel® AVX

Propriétés de la mémoire CPU

- Taille de mémoire maximale (GB) : 32
- Type de mémoire : DDR3/L/-RS 1333/1600
- Nombre de canaux mémoire : 2
- Bande Passante mémoire max (GB/s) : 25,6

4.2.3 Spécifications AMD FirePro M6000

Propriétés générales

- Nombre de coeurs : N/A
- Fréquence de l'horloge graphique (MHz) : 800
- Fréquence de l'horloge processeur (MHz) : N/A

Propriétés de la mémoire GPU

- Fréquence de l'horloge mémoire (MHZ) : 4500
- Standard Memory Config : N/A
- Memory interface width : N/A
- Bande Passante mémoire (GB/s) : N/A

ANNEXE B : Eléments de code

4.3 Algorithme de génération uniforme de mot de passe et de hash MD5

Script de génération de mots de passe et de leur hash MD5 correspondants :

```
#!/usr/bin/env python3

#Auteur : Jean-Baptiste Gaeng

import string
import random
import argparse
import hashlib

def password_generator(size=5, charset=string.ascii_letters +
↪  string.punctuation + string.digits):
    """
    Générateur de string aléatoires de taille donné dans un domaine de
↪  caractère donné (charset)
    """
    return ''.join(random.choice(charset) for _ in range(size))

def main():
    #Parsing des arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("-nb", "--nombre_mdp", help="Nombre de mots de passe a
    ↪  generer", type=int, default="5")
    parser.add_argument("-l", "--longueur_mdp", help="Longueur des mots de
    ↪  passe", type=int, default="5")
    parser.add_argument("-ch", "--charset", help="Ensemble des caracteres
    ↪  composant les mots de passe", default = string.ascii_letters +
    ↪  string.punctuation + string.digits)
    args = parser.parse_args()

    encoding = "ascii"
    file = open("passwords.txt","x")
    file2 = open("hashs.txt", "x")

    #Génération des mdps et des hashs MD5 correspondants
    for _ in range (args.nombre_mdp):
        password = password_generator(args.longueur_mdp, args.charset)
        hash = (hashlib.md5(password.encode(encoding))).hexdigest()
        file.write(password + "\n")
        file2.write(hash + "\n")

if __name__ == "__main__":
    main()
```

4.4 Implémentations CPU

4.4.1 Recherche exhaustive (brute-force) naïf

Script implémentant la recherche exhaustive naïve :

```
#!/usr/bin/env python3

```

```python
#Auteur : Jean-Baptiste Gaeng

from hashlib import md5
from time import time
from string import printable
from itertools import product, count
from binascii import unhexlify

def passwords(encoding):
    """
    Iterateur sur tous les mdp possibles
    """
    chars = [c.encode(encoding) for c in printable]
    for length in count(start=1):
        for pwd in product(chars, repeat=length):
            yield b''.join(pwd)

#Itere sur tous les mdp possibles, et compare son hash apres application

def crack(search_hash, encoding):
    """
    Itere sur tous les mdp possibles, et compare son hash MD5 au hash à
    ↪ cracker
    """
    for pwd in passwords(encoding):
        if md5(pwd).digest() == search_hash:
            return pwd.decode(encoding)

def main():
    encoding = 'ascii'  # utf-8 pour support unicode

    #Ouverture du fichier contenant les hashs à cracker
    #Format des hashs dans le fichier : hexadecimal
    f = open("hashs.txt", "r")
    lines = f.readlines()
    lines = [line.rstrip("\n") for line in lines]
    f.close()

    total_time = 0
    mean_cracked_time = 0

    #Parcourt des hashs
    for hex_password_hash in lines:

        #Conversion des hashs de l'hexadecimal au binaire
        binary_password_hash = unhexlify(hex_password_hash)

        #Crackage
        start = time()
        cracked = crack(binary_password_hash, encoding)
        end = time()
        total_time = total_time + end-start
        print(f"Mot de passe cracké : {cracked}")
        print(f"Temps: {end - start} secondes.")

    mean_cracked_time = total_time / len(lines)
    print(f"Temps total : {total_time}")
    print(f"Temps moyen par mdp cracké : {mean_cracked_time}")
```

```python

if __name__ == "__main__":
    main()
```

4.4.2 Recherche exhaustive parallèle en utilisant OpenMP

```c
//Auteurs : Alexandre GAENG, Jean-Baptiste GAENG

// $ gcc -o bruteforce-md5-openmp bruteforce-md5-openmp.c -fopenmp -lssl
 ↪  -lcrypto
// $ export OMP_NUM_THREADS=4
// $ ./bruteforce-md5-openmp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <openssl/sha.h>
#include <openssl/md5.h>
#include <time.h>

typedef unsigned char byte;
#define NB_CHARACTERS 4 // Nombre de caractères du mdp
int NB_ATTEMPTS = 0;

int matches(byte *a, byte* b) {
    for (int i = 0; i < 16; i++)
        if (a[i] != b[i])
            return 0;
    return 1;
}

byte* StringHashToByteArray(const char* s) {
    byte* hash = (byte*) malloc(16);
    char two[3];
    two[2] = 0;
    for (int i = 0; i < 16; i++) {
        two[0] = s[i * 2];
        two[1] = s[i * 2 + 1];
        hash[i] = (byte)strtol(two, 0, 16);
    }
    return hash;
}

void printResult(byte* password, byte* hash) {
    char sPass[NB_CHARACTERS+1];
    memcpy(sPass, password, NB_CHARACTERS);
    sPass[NB_CHARACTERS] = 0;
    printf("%s => ", sPass);
    for (int i = 0; i < MD5_DIGEST_LENGTH; i++)
        printf("%02x", hash[i]);
    printf("\n");
    printf("Nombres de tentatives : %i \n", NB_ATTEMPTS);
}

```

```
49 void computeCombination(int p, byte* password, byte* hashByteArray, int k){
50
51     if(p==2){
52         for(password[1] = 33; password[1] < 127; password[1]++){
53             byte *hash = MD5(password, k, 0);
54             NB_ATTEMPTS = NB_ATTEMPTS + 1;
55
56             if (matches(hashByteArray, hash)){
57                     printResult(password, hash);
58
59             }
60         }
61     }
62
63     if(p>2){
64         for(password[p-1] = 33; password[p-1] < 127; password[p-1]++){
65             computeCombination(p-1, password, hashByteArray, k);
66         }
67     }
68
69 }
70
71 int main(int argc, char **argv)
72 {
73
74 #pragma omp parallel
75     {
76
77 #pragma omp for
78         for (int a = 0; a < 94; a++)
79         {
80             byte password[NB_CHARACTERS] = {33 + a};
81             byte* hashByteArray =
               ↪  StringHashToByteArray("f71dbe52628a3f83a77ab494817525c6"); //
               ↪  Le hash correspond à "toto"
82             computeCombination(NB_CHARACTERS,password,hashByteArray,NB_CHARACTERS);
83             free(hashByteArray);
84     }
85 }
86 return 0;
87 }
```