

Protection des données dans les environnements distribués (Cloud)

RAPPORT DE FIN DE PROJET

29 juin 2017

Encadrants

Katarzyna KAPUSTA

Gérard MEMMI

Sommaire

- I. Analyses et tests
 - 1. Automatisation des tests
 - 2. Analyse et affichage des résultats
 - 3. Tests de performance
 - 4. Tests de corrélation
- II. Adaptation de l'algorithme
 - 1. Encodage d'images BITMAP
 - 2. Implémentation de la redondance
- III. Dispersion
 - 1. Dispersion des fragments sur un ensemble de serveurs de l'école
 - 2. Automatisation de l'encodage et de la dispersion
 - 3. Dispersion des fragments sur des hébergeurs clouds privés
 - 4. Traçage des paquets

Résumé

Les solutions d'hébergement de données gagnent de plus en plus en popularité en permettant aux entreprises et particuliers de réduire leurs coûts. Se pose alors le double problème de la protection des données hébergées et de leur résilience. Ce rapport analyse et compare plusieurs méthodes de protections de données par les moyens de fragmentation dans des environnements distribués, notamment le Cloud. Entre autre, il inclut des tests de sécurité et de performance d'un algorithme de fragmentation avec redondance transformant une donnée en n fragments, où k d'entre eux sont nécessaires pour la reconstruction de la donnée initiale. De plus, il présente une façon de fragmenter, qui partitionne l'information en deux fragments: un fragment confidentiel de petite taille et un fragment plus large ne contenant pas d'information sensible. Finalement, un outil de dispersion entre plusieurs serveurs/Clouds publiques est présenté.

Introduction

La protection des données est généralement assurée par l'encryptage des fichiers. Les algorithmes utilisés utilisent souvent un schéma utilisant des clés de protections, publiques ou privées. La question de l'hébergement des clés est aussi un problème.

C'est de cette façon que sont principalement protégées les données sensibles stockées dans des environnements distribués. Sur ce type de stockage se pose d'autres difficultés, comme l'intégrité des machines de stockage. Si un intrus possède l'accès à un de ces serveurs, il peut accéder aux fichiers entiers qui y sont déposés. Cela facilite grandement son travail pour subtiliser l'information.

Ce projet inspecte une nouvelle approche à ces problématiques, la fragmentation [5]. L'idée est de séparer le fichier à protéger en plusieurs parties, qui seront encodées récursivement. Ainsi, ce sont les fragments en eux-mêmes qui jouent le rôle de clé, il n'y a plus de réelle gestion de clé à effectuer. Les fragments sont ensuite dispersés sur le cloud, de façon à ce que tous les fragments ne se retrouvent pas physiquement sur le même serveur. Ainsi, une personne mal intentionnée ayant accès à cette machine n'aura toutefois pas accès au fichier, qui nécessite d'autres fragments pour être décodé.

Nous allons ainsi étudier et utiliser un algorithme de fragmentation inspiré de l'algorithme Shamir Secret Sharing [6], analyser ses performances et proposer un script automatisant les opérations de dispersion et de récupérations des fichiers sur des environnements distribués.



Analyses et tests

1. Automatisation des tests

Nous avons commencé par une phase d'analyse du code fourni et de compréhension de son fonctionnement (<https://goo.gl/tZRXok>). Afin de faciliter la mise en œuvre de tests et de s'appropriier les fonctions fournies, nous avons implémenté une classe TestMaster dans le package tests.

Cette classe comprend une méthode main prenant en arguments les arguments nécessaires à l'encodage d'un répertoire :

- fragmentSize
- inputDirectory
- outputDirectory
- tries

Tous les fichiers contenus dans le répertoire source sont alors encryptés et les fragments sont enregistrés dans le dossier de sortie. Chaque encodage se fait pour plusieurs valeurs du paramètre k (ici 2 et 10) et tries fois pour chacune de ces valeurs.

La particularité de cette classe est de produire un fichier "instructions" contenant la liste des fichiers "en clair" ainsi que d'un de leurs fragments, et ce pour chaque distribution de paramètres différente. Ce fichier stocke l'adresse des fichiers ainsi que les grandeurs consignées/mesurées (taille, k, temps d'encodage, ...)

Le paramètre tries permet d'effectuer plusieurs tests d'implémentation afin de faire une moyenne des temps d'exécution.

2. Analyse et affichage des résultats

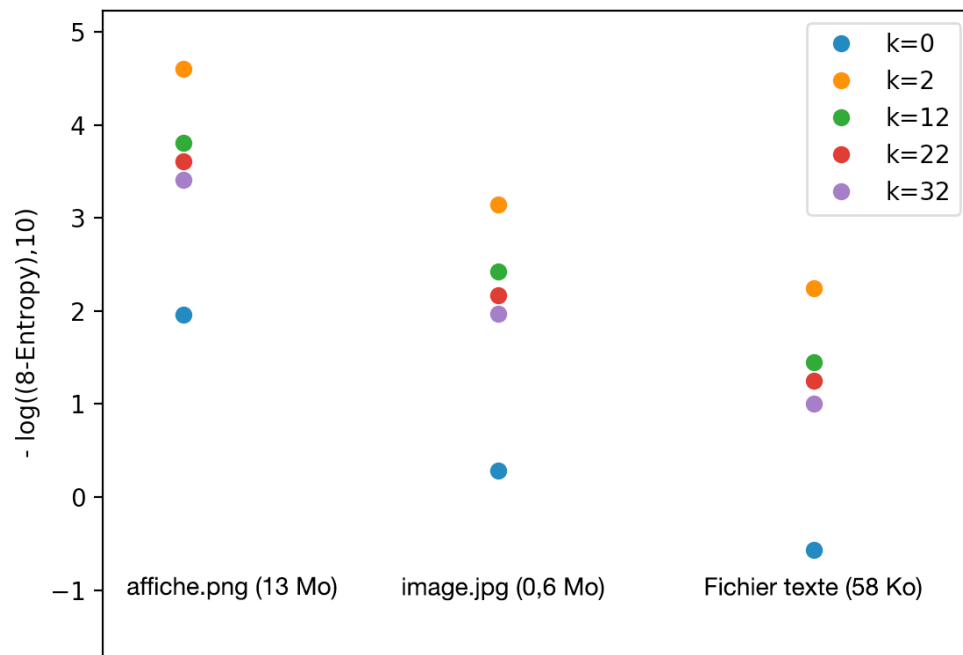
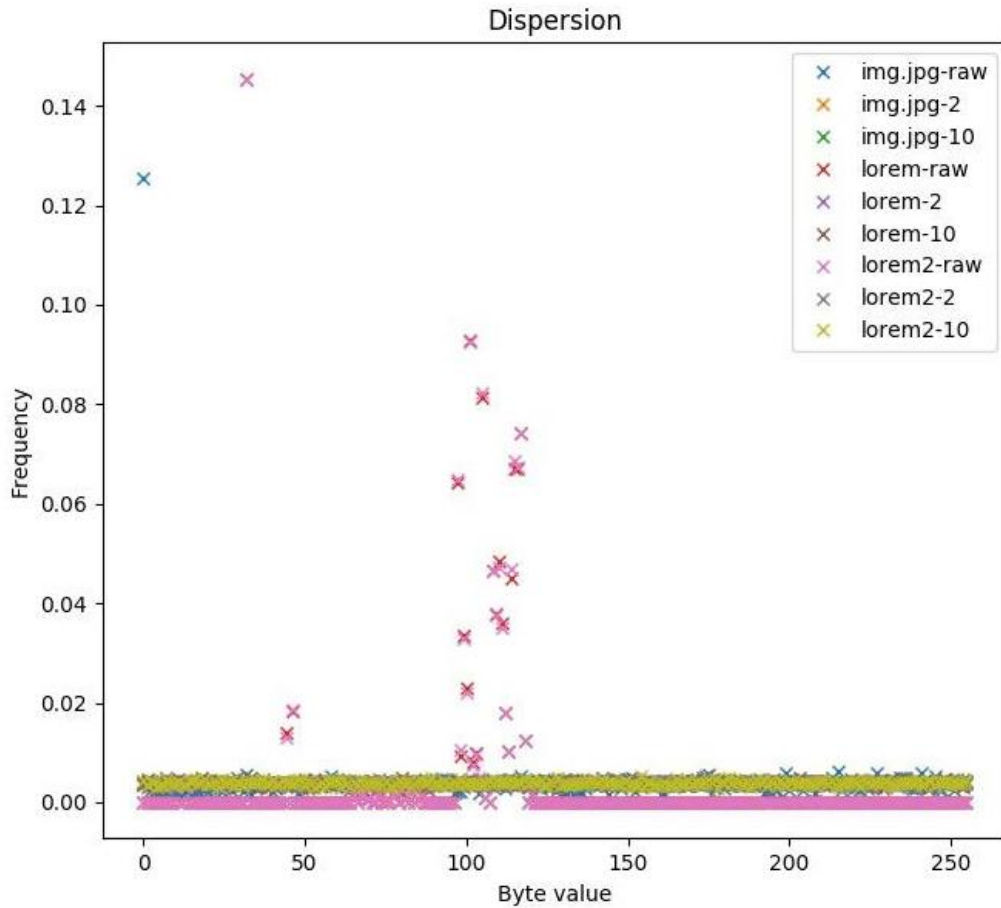
Le fichier d'instructions généré précédemment permet à un script Python de connaître une liste de fichiers qu'il va analyser. Chaque fichier est représenté par une classe TestData. Les informations pertinentes sont extraites et affichées grâce à la bibliothèque matplotlib.pyplot. Les informations relevées sont les suivantes :

Dispersion La fréquence d'apparition de chaque octet (0 - 255). Pour les fichiers textes en clair, les fréquences des octets représentant des lettres sont très supérieures, et les autres fréquences sont inexistantes. Pour les fichiers cryptés, tous les octets ont une fréquence proche de 1/256.

Entropie A partir de la dispersion précédemment calculée, nous pouvons déterminer l'entropie par octet de chaque fichier, par la formule suivante :

$$H = \sum_i f_i \cdot \log(f_i)$$

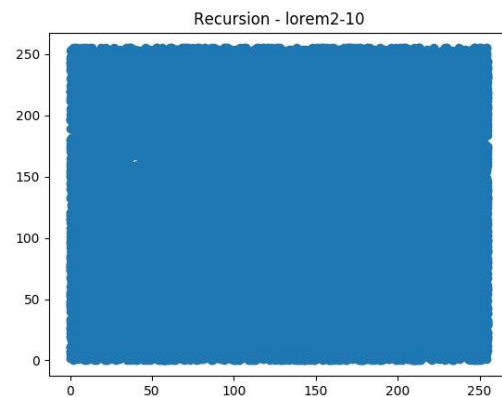
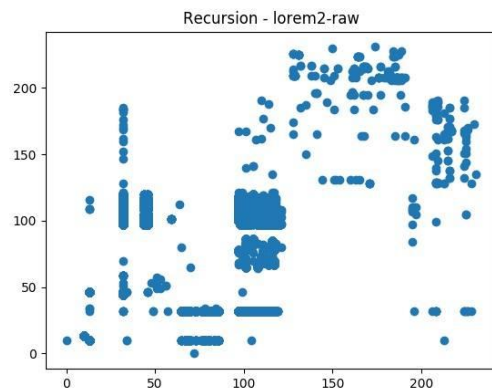
Les fichiers cryptés ont une entropie très proche de 8, qui est la valeur maximale obtenue théoriquement pour une distribution uniforme des octets. Les fichiers non encryptés ont une entropie plus faible. On représente ici l'opposée du logarithme de la différence entre l'entropie d'un seul fragment choisi au hasard et l'entropie maximale théorique. Plus le point est haut dans le graphique, plus l'entropie du fichier est proche du maximum.



Réccurrence Les graphiques suivants représentent la valeur de l'octet $n+1$ en fonction de l'octet n . A noter que l'affichage 2D ne rend pas compte du nombre de points par position sur le graphe.

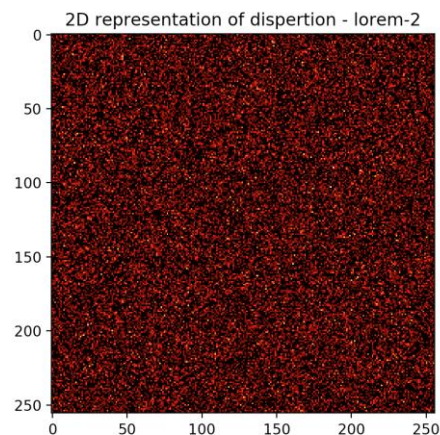
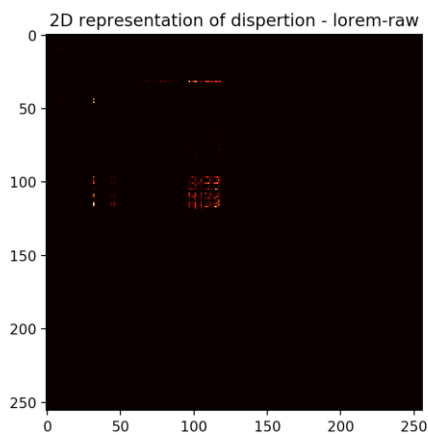
Pour un fichier en clair, on remarque facilement la forte corrélation entre ces octets. Par exemple, la grande barre à gauche résulte du fait qu'après un espace, on recommence très souvent par une lettre.

Le groupement important de points au centre représente toutes les lettres qui se suivent dans le corps du texte. Pour un fichier crypté, cette corrélation n'existe plus.



Récurrance 2D par luminosité:

Afin d'améliorer la représentation de la récurrence, nous intégrons des couleurs liées au nombre d'occurrences de l'événement: $x = \text{octet}[i]$ & $y = \text{octet}[i+1]$. L'affichage de ce graphique est cependant un peu plus long que le précédent mais il permet d'avoir une meilleure représentation de la corrélation entre la valeur de 2 octets successifs.

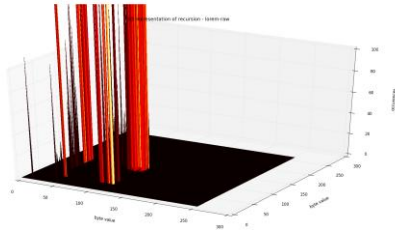


Récurrance 3D:

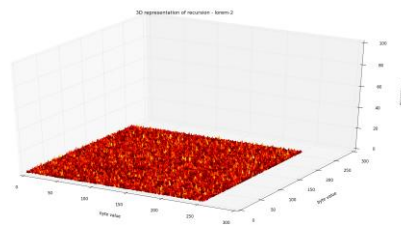
On représente ici la fonction à 2 variables: $\sum_i \text{Indicatrice}(\text{octet}[i] = X, \text{octet}[i+1] = Y)$

L'affichage du graphique représentant cette fonction est cependant beaucoup plus long que les 2 graphiques précédents. Cette affichage permet une représentation plus esthétique de la récurrence, cependant le temps d'affichage limite son intérêt.

Lorem original



Lorem encodé avec k=2



Temps d'encodage Les derniers graphes représentent le temps d'encodage en fonction de la taille du fichier et en fonction de k, tous deux linéaires. Ce test sera détaillé en détail dans la partie suivante.

3. Tests des performances

Dans cette partie, nous allons comparer le temps d'exécution des fonction d'encodage et de décodage selon que le multithreading soit activé ou pas et selon le type d'ordinateur utilisé.

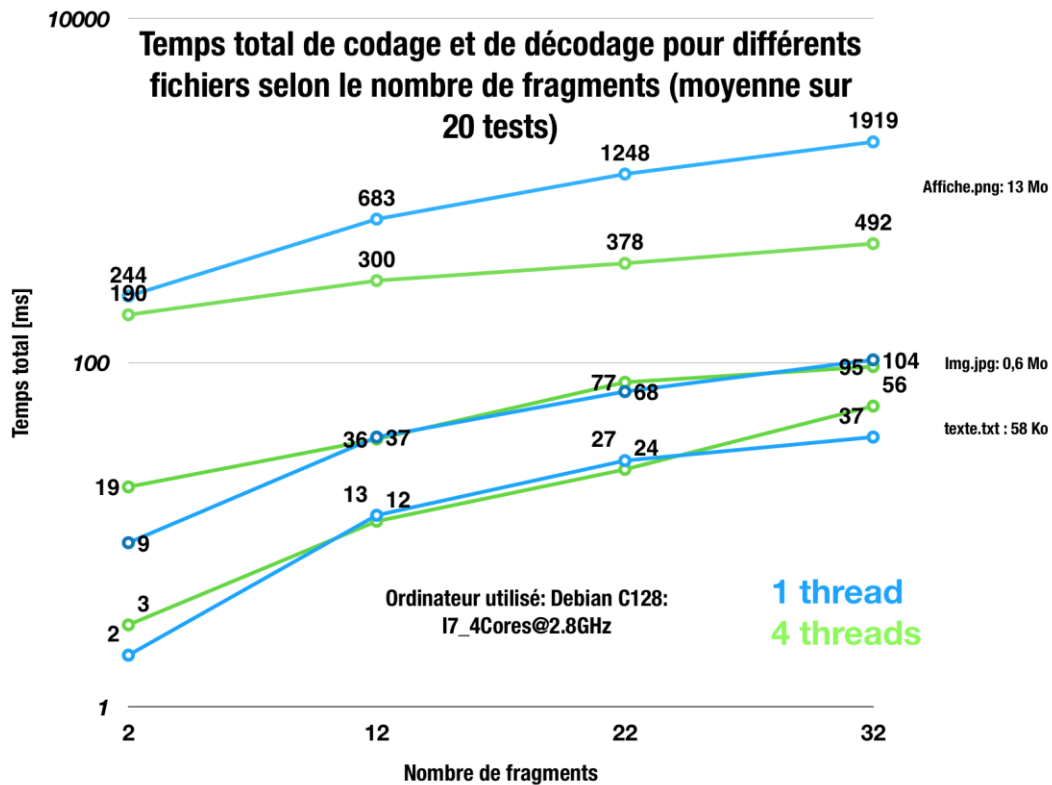
Paramètres:

- En entrée: 4 fichiers de tailles différentes (lorem.txt: 70 kB, lorem2.txt: 98 kB, img.jpg: 602 kB, Affiche_theatre_2017.png: 14 000 kB)
- Chaque fichier est encodé puis décodé. On mesure le temps total de calcul nécessaire à cette opération.
- On effectue cette opération 4 fois par fichier pour des valeurs de k différentes: $k \in \{ 2, 12, 22, 32 \}$.
- Le décodage peut utiliser ou non le multithreading. L'encodage est toujours réalisé sur un seul thread.
- Le calcul est effectué soit sur l'ordinateur local (Debian C128 équipé d'un processeur 8 threads à 2,8 GHz) ou sur le serveur lame15 (Fedora équipé de 4 processeurs de 8 coeurs cadencés à 2,39 GHz).

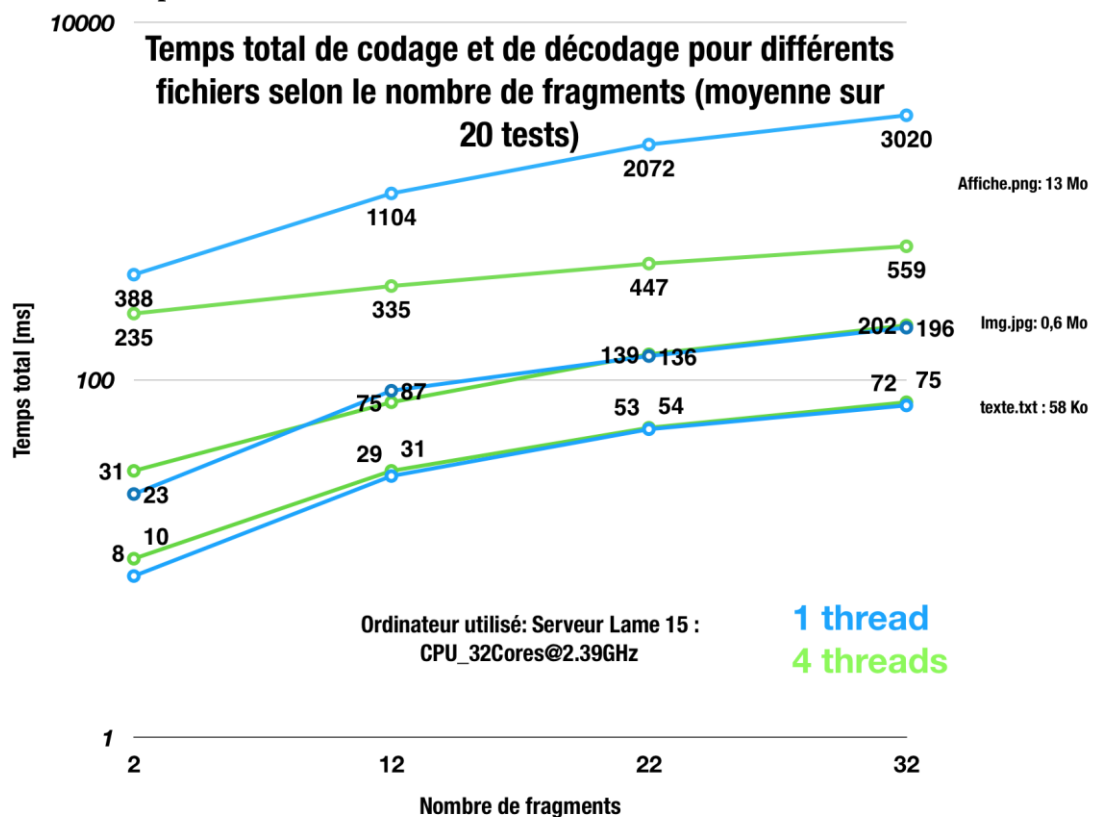
Résultats :

Ce programme est effectué 20 fois et on prend le temps moyen d'exécution en supprimant la première valeur (Qui est parfois 10 fois supérieure aux autres valeurs). Puis, on affiche ce temps moyen d'exécution:

Calculs effectués en Local par l'un des ordinateurs de la salle C128:



Calculs effectués par le serveur lame15 :



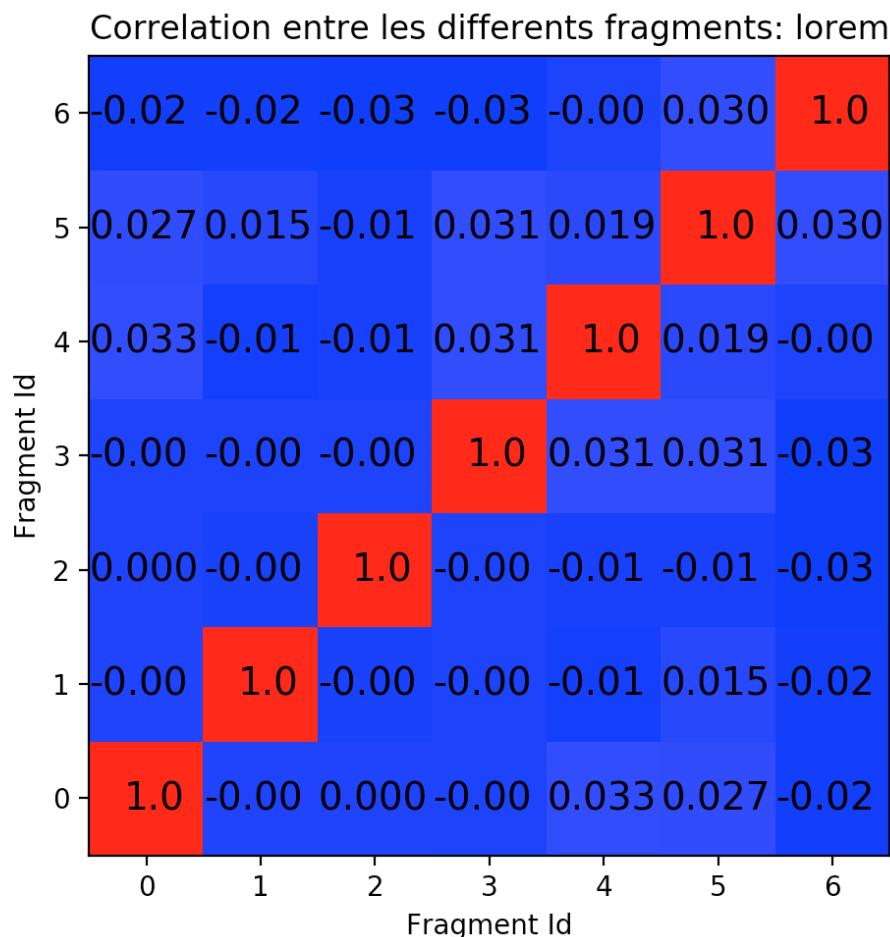
L'utilisation du multithreading nous permet de gagner un facteur supérieur à 4 (jusqu'à 5.4 pour l'affiche avec $k=32$) dans le temps moyen d'exécution pour le fichier affiche (Qui pèse 13 Mo).

Nous remarquons aussi que le temps d'exécution sur le serveur est plus long que le temps d'exécution en local, cela est en partie due au fait que l'encodage doit être réalisé sur un seul thread et que la cadence du processeur de lame 15 est inférieure à celle du serveur local (2,39 contre 2,8 GHz)

4. Tests de corrélations

On mesure le coefficient de corrélation entre les octets de chaque fragment. Un coefficient de corrélation proche de 0 empêche de deviner un fragment à partir d'un autre. Ici, le test a été réalisé pour le fichier texte, qui a été fragmenté en 7 fragments.

Les tests ont été automatisés au maximum, l'utilisateur doit seulement lancer un fichier .jar en ligne de commande avec les arguments de son choix. Un programme java calcule les fragments et les exporte dans un dossier output. Puis, le programme java ouvre un terminal pour lancer l'exécution d'un script Python qui va récupérer les fichiers dans le dossier Output et les analyser pour calculer les différents coefficients de corrélation. Une fenêtre Matplotlib s'ouvre alors avec le graphique affiché ci-dessous. Enfin, le programme java appelle un deuxième script python qui va vider le dossier output.

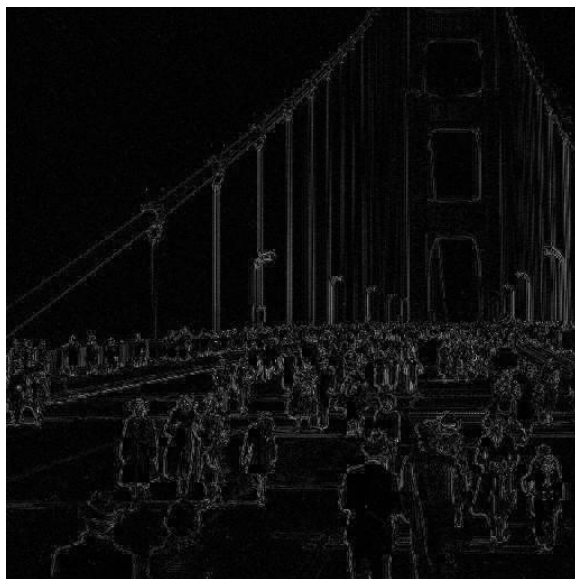


Adaptation de l'algorithme

1. Encodage d'images BITMAP

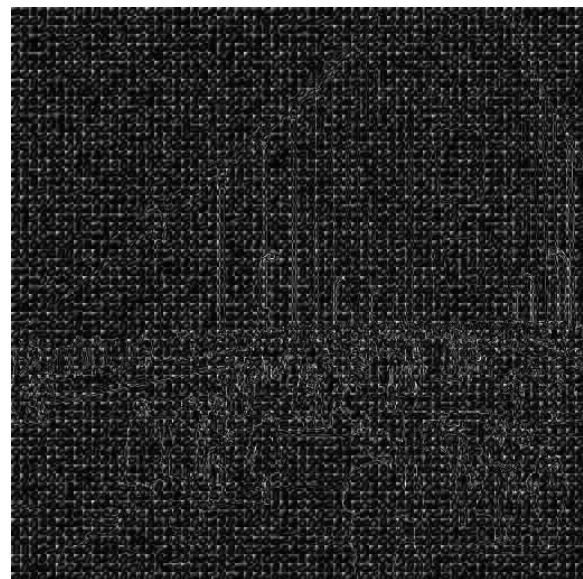
Il s'agit ici d'utiliser l'algorithme de fragmentation et d'encryption pour des fichiers images. Or l'efficacité de la fragmentation peut dépendre du type de fichier. En s'inspirant de la méthode décrite dans [7], nous avons implémenté un programme Java permettant d'encrypter les images bitmap. Ce programme est dans la classe ImageEncoder du package tests.

A l'aide de la bibliothèque *jTransforms* [2], le programme effectue une DCT 2D 8x8 sur une image noir et blanc fournie en entrée. Les 6 premiers coefficients de chaque carré 8*8 de l'image sont concaténés dans un tableau d'octets, encodé et fragmenté par le programme. Ces coefficients correspondent aux basses fréquences, ont sont remplacés soit par un 0 soit par des valeurs aléatoires. Ils sont ensuite stockés en clair dans un fichier. Seuls ils ne sont guères utilisables, sauf si l'image comporte de forts contrastes pour ses contours, comme le montrent les images suivantes.



Fréquences basses remplacées par des 0

Les contours de l'image ressortent beaucoup, on peut donc retrouver l'image originale assez facilement



Fréquences basses remplacées aléatoirement

En utilisant des valeurs aléatoires, tout est brouillé. L'image est nettement moins reconnaissable.

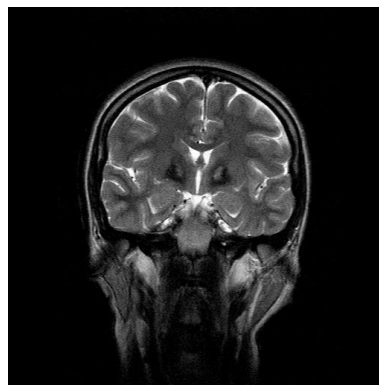
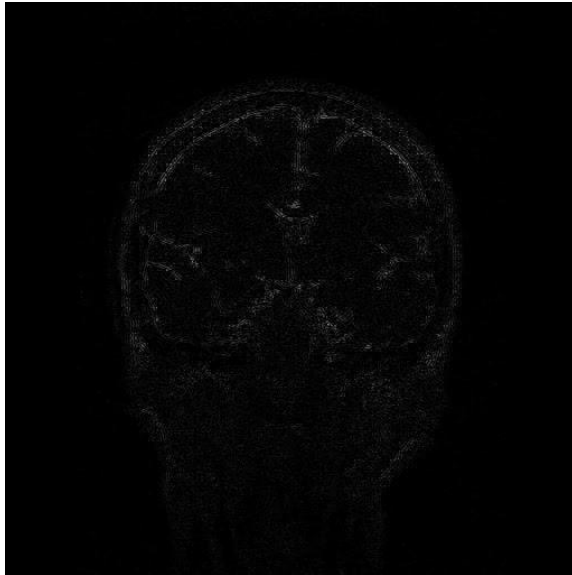
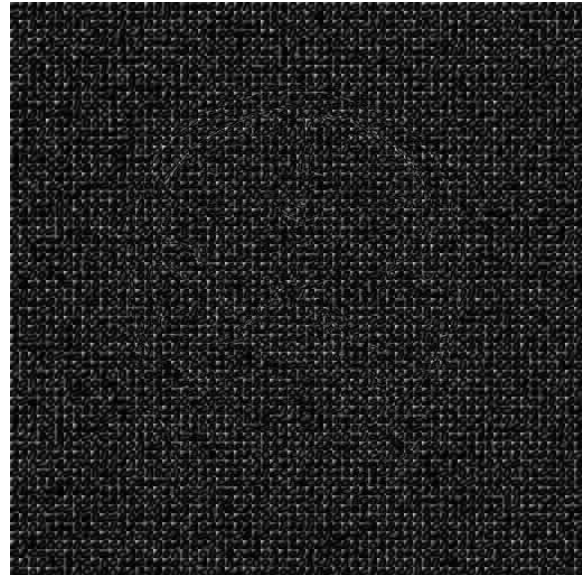


IMAGE DÉCODÉE

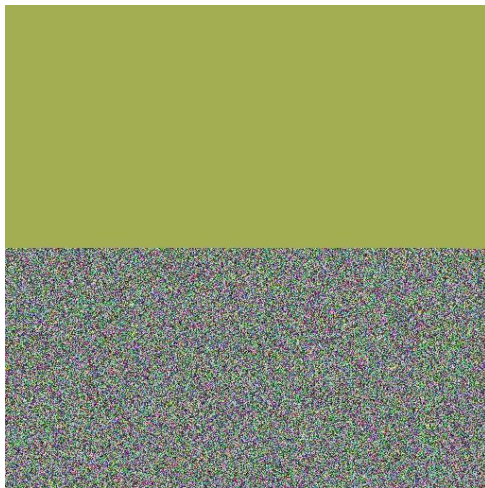


Fréquences basses remplacées par des 0



Fréquences basses remplacées aléatoirement

L'utilisation de l'algorithme de fragmentation sur l'image entière donne des fragments totalement indépendants :



De telles images sont obtenues en simulant une structure de fichier *bmp*. Le header du véritable fichier *bmp* original écrase les premiers octets du fichier, et l'on rajoute des 0 jusqu'à obtenir la taille adéquate.

Pour la démonstration, le programme affiche au fur et à mesure les images calculées.

Remarques Nous pouvons nous interroger sur les garanties de sécurité offertes par cette méthode. Certes l'image brouillée par des valeurs aléatoires pour les basses fréquences est inutilisable à l'oeil nu. Mais, aux erreurs de calcul près, il est possible de ré-effectuer une DCT 8x8 sur l'image et remettre à 0 les valeurs de basses fréquence. Il est donc possible d'obtenir de nouveau les contours de l'image, qui restent exploitables mais dans une moindre mesure. La protection fournie par la fragmentation des basses fréquences n'est cependant pas compromise. Un nombre de fragments inférieur à k ne suffit pas à retrouver l'information, et les fragments en eux-mêmes sont inexploitable, comme le montre l'image ci-dessus. Une façon d'augmenter la protection repose dans la manipulation du nombre de coefficients basses (par exemple 8 au lieu de 6) - en prenant plus de coefficients nous obtenons une image brouillée même sans remplacer les fréquences basses par des nombres aléatoires.

2. Implémentation de la redondance

Afin de parfaire l'algorithme que nous utilisons pour l'encodage et le décodage de fragments, nous avons ajouté de la redondance dans les fragments et de la correction d'erreurs à l'aide du code de Reed-Solomon. Nous avons utilisé le code source de la librairie Java du même nom, développée par Backblaze [3]. Afin de faire fonctionner le programme, il faut désormais spécifier un entier n supérieur à k , qui indique le nombre total de fragments à utiliser.

I. Dispersion

1. Dispersion des fragments sur un ensemble de serveurs de l'école

Une fois les données fragmentées, il faut les disperser sur des machines différentes afin que la fragmentation soit utile et efficace. Pour cela, nous avons mis au point un script Python effectuant cela en répartissant les fichiers contenus dans un dossier sur les serveurs de l'école.

Les serveurs en questions sont ceux du département InfRes de Télécom. Nous utilisons les serveurs lameXX.enst.fr. Les échanges de fichiers se font via le protocole SCP, sous une connexion SSH. Cela est faisable en Python par le module paramiko [4].

A chaque fichier est attribué aléatoirement un hôte. Les fichiers sont ensuite envoyés dans le répertoire /tmp/ des hôtes concernés. La carte des fichiers envoyés est sauvegardée dans le répertoire du script. Afin de pouvoir stocker plusieurs groupes de fragments indépendants, à chaque envoi est attribué un identifiant (16 caractères alphanumériques) qui est nécessaire pour utiliser la fonction de téléchargement. On note aussi la taille du fichier total qui a été fragmenté avant d'être envoyé, car elle est nécessaire pour reconstruire le fichier avec l'algorithme Java (initialisation de la variable lastFragmentSize).

Une fonction clean permet aussi de vider le fichier "locations" et supprime tous les fichiers uploadés par l'utilisateur qui s'est loggué sur les différents hôtes.

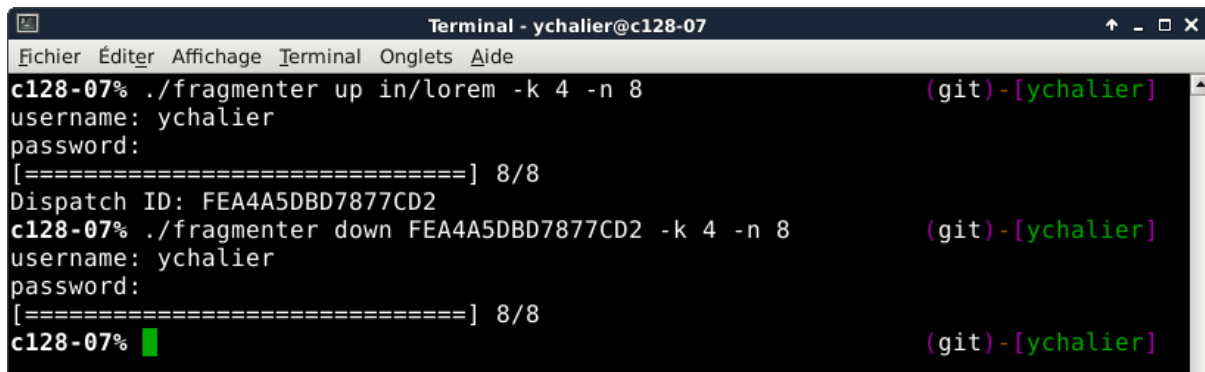
2. Automatisation de l'encodage et de la dispersion

Afin de simplifier l'utilisation des scripts implémentés précédemment, nous avons créé deux exécutable JAR pour l'encodage et le décodage d'un fichier. Les données nécessaires sont passées par arguments, si souhaité. Des valeurs par défauts ont été définies afin de simplifier les exécutions de tests.

Ensuite, un script Python fragmenter.py réunit le tout. Il se charge, à partir du chemin d'un fichier passé en argument, de l'encoder, le fragmenter et le disperser sur les hôtes sélectionnés (jusqu'à alors les serveurs de l'école).

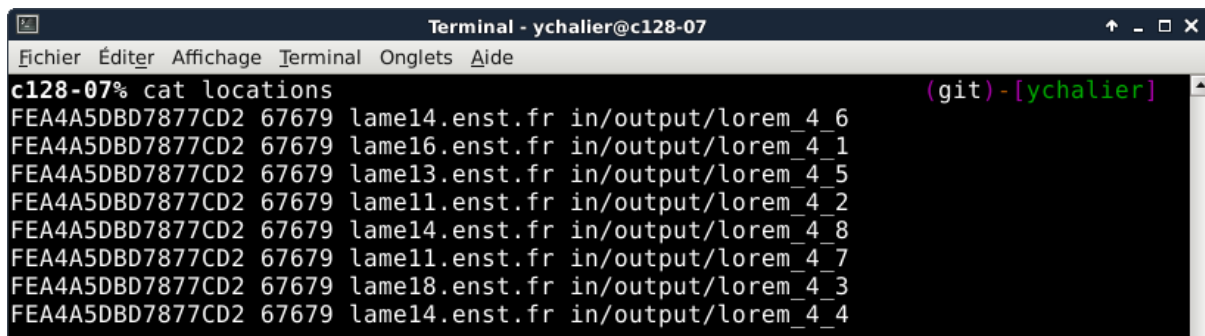
L'identifiant de l'envoi est retourné. Il permet, une fois passé en argument, d'automatiquement télécharger les fragments et les décoder pour reformer le fichier initial.

Des problèmes de versions et de compatibilité avec différents environnements ont cependant été rencontrés lors du développement, aussi nous avons réalisé un exécutable de ce fichier fragmenter.py, à l'aide du module Python pyInstaller.



```
Terminal - ychallier@c128-07
Fichier Éditer Affichage Terminal Onglets Aide
c128-07% ./fragmenter up in/lorem -k 4 -n 8 (git)-[ychallier]
username: ychallier
password:
[=====] 8/8
Dispatch ID: FEA4A5DBD7877CD2
c128-07% ./fragmenter down FEA4A5DBD7877CD2 -k 4 -n 8 (git)-[ychallier]
username: ychallier
password:
[=====] 8/8
c128-07% (git)-[ychallier]
```

Capture d'écran du terminal lors de l'upload puis du download d'un fichier texte



```
Terminal - ychallier@c128-07
Fichier Éditer Affichage Terminal Onglets Aide
c128-07% cat locations (git)-[ychallier]
FEA4A5DBD7877CD2 67679 lame14.enst.fr in/output/lorem_4_6
FEA4A5DBD7877CD2 67679 lame16.enst.fr in/output/lorem_4_1
FEA4A5DBD7877CD2 67679 lame13.enst.fr in/output/lorem_4_5
FEA4A5DBD7877CD2 67679 lame11.enst.fr in/output/lorem_4_2
FEA4A5DBD7877CD2 67679 lame14.enst.fr in/output/lorem_4_8
FEA4A5DBD7877CD2 67679 lame11.enst.fr in/output/lorem_4_7
FEA4A5DBD7877CD2 67679 lame18.enst.fr in/output/lorem_4_3
FEA4A5DBD7877CD2 67679 lame14.enst.fr in/output/lorem_4_4
```

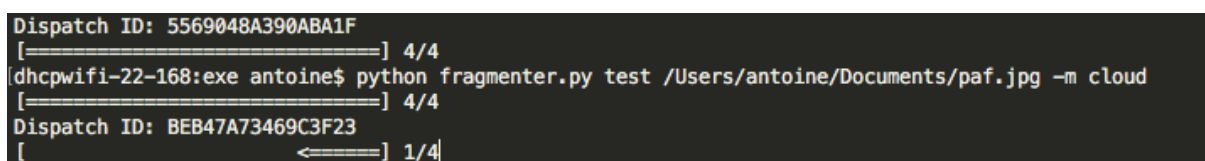
Capture d'écran du terminal lors de l'affichage du fichier locations, contenant l'emplacement des fragments

3. Dispersion sur un ensemble d'hébergeurs clouds privés

Aujourd'hui, de l'avis de nombreux experts, les cloud privés sont les plus sécurisés. Nous avons donc mis au points un processus de dispersion permettant d'envoyer les fragments sur différents cloud. Pour cela nous avons utilisé une librairie adapté [1].

Nous utilisons des comptes développeurs Dropbox, Google Drive, Yandex et Backblaze, couplées à des applications PAF-Cloud, qui permettent de réaliser des échanges les plus sécurisés possibles.

L'exécutable va choisir des clouds au hasard et disperser les fragments dans ces derniers, pour que le fichier ne puisse pas être reconstruit. Chaque dispersion est associée à un ID. Pour récupérer les fragments, il suffit de connaître cet ID. Les fragments seront alors récupérés, décryptés et assemblés pour reconstruire le fichier d'origine.



```
Dispatch ID: 5569048A390ABA1F
[=====] 4/4
dhcpcwif-22-168:exe antoine$ python fragmenter.py test /Users/antoine/Documents/paf.jpg -m cloud
[=====] 4/4
Dispatch ID: BEB47A73469C3F23
[=====] 1/4|
```

4. Traçage des paquets

Pour assurer une sécurité encore plus importante, nous avons également suivi le chemin emprunté par un fragment jusqu'à un datacenter DropBox situé à San Francisco.

```
tracert to dropbox.com (162.125.248.1), 64 hops max, 52 byte packets
 1 gw-radio1x (137.194.22.254) 10.990 ms 1.251 ms 8.223 ms
 2 asrix (137.194.4.240) 1.295 ms 1.195 ms 1.009 ms
 3 ge-5-3-7.edge5.paris1.level3.net (212.73.200.45) 1.582 ms 1.433 ms 1.417 ms
 4 * * *
 5 * * *
 6 * * *
 7 185.45.10.155 (185.45.10.155) 12.266 ms 9.259 ms 9.192 ms
 8 162.125.248.1 (162.125.248.1) 9.489 ms 8.873 ms 8.855 ms
```

En analysant la route ci-dessus, nous pouvons voir que le paquet sort par le routeur de Telecom, avant d'arriver à un autre routeur parisien. Après il emprunte un chemin inconnu avant d'arriver au USA. L'objectif de cette démarche est de vérifier qu'un intermédiaire de type "Man in the middle" ne récupère pas tous les fragments. Pour que cette recherche soit plus pertinente, il faudrait lancer simultanément un ping vers le datacenter en parallèle qui ne passerait pas par le réseau local de Telecom. La comparaison des deux routes permettrait de détecter plus facilement une anomalie. Par exemple, on pourrait envoyer simultanément des paquets sur le réseau de l'école et sur le réseau cellulaire via un partage de connexion. Au moins, cela permettra de différencier les premiers routeurs. Afin de différencier les derniers routeurs, il faut que les destinations des paquets soient différents, par exemple en envoyant les fragments sur différents hébergeurs. Pour plus de sécurité, on pourrait répartir les fragments en différents ensembles, puis envoyer chaque ensemble vers un hébergeur différent et avec une route initiale différente. Et vérifier avec un traceroute que les ensembles de paquets ne passent pas par le même routeur. Si ce n'est pas le cas, alors on est dans une situation de forte sécurité.

Conclusion

A mesure que le stockage de données sur Cloud prend de l'ampleur, leur sécurité est de plus en plus importante. Devant la quantité de données, un cryptage systématique de l'ensemble des fichiers ne permettra pas de répondre aux besoins. De plus, la fiabilité de ces méthodes dépend seulement de la sécurité et de la force de la clé choisie. Enfin, cela ne protège pas contre les attaques de types Ransomware, comme le monde a pris conscience récemment.

L'objectif était donc double:

- trouver une méthode d'hébergement minimisant l'effort de gestion des clés tout en assurant une sécurité élevée.
- assurer la résilience des fichiers hébergés

Les résultats observés à l'aide de notre méthode sont concluants:

- L'algorithme que nous avons mis en place permet de stocker les fragments soit sur des serveurs privés (dans ce cas sur des serveurs de l'école) soit sur un ensemble de cloud privés (Dropbox, Google Drive, Yandex, Backblaze), ce qui permet d'assurer la résilience du stockage.
- Les performances temporelles sont suffisantes pour pouvoir envisager l'utilisation de ces algorithmes sur des ordinateurs personnels.

Ces algorithmes offrent une solution performante, sécurisée et novatrice pour le stockage des données sensibles sur des serveurs distribués.

Bibliographie

- [1] Librairie de dispersion sur les cloud: <https://github.com/ncw/rclone>
- [2] Librairie DCT: <https://github.com/wendykierp/JTransforms>
- [3] Librairie Reed Solomon Backblaze: <https://github.com/Backblaze/JavaReedSolomon>
- [4] Librairie pour la communication ssh: <https://github.com/paramiko/paramiko>
- [5] K. Kapusta, G. Memmi, and H. Noura. *Poster: A keyless efficient algorithm for data protection by means of fragmentation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pages 1745–1747, New York, NY, USA, 2016. ACM.*
- [6] K. Kapusta and G. Memmi, "Data protection by means of fragmentation in distributed storage systems," *2015 International Conference on Protocol Engineering (ICPE) and International Conference on New Technologies of Distributed Systems (NTDS), Paris, 2015, pp. 1-8.*
- [7] G. Memmi, K. Kapusta, and H. Qiu, "Data Protection: Combining Fragmentation, Encryption, and Dispersion", *Cyber Security of Smart Cities, Industrial Control System and Communications (SSIC), Shanghai, 2015, pp. 1-9.*