



Stéganographie

Mise en place de techniques stéganographiques sur des fichiers BMP

Rapport rédigé & produit par : **Kévin MOREAU & Laurent GRAFF**

Professeur responsable : **Éric FILIOL**

École Nationale Supérieure d'Ingénieurs de Bretagne Sud

Apprenti - Ingénieur Expert Cyberdéfense

2^{ème} année

Promotion

AIRBUS

Table des matières

1.	<u>RAPPEL DES OBJECTIFS</u>	<u>2</u>
2.	<u>FONCTION D'INSERTION D'UN MESSAGE CLAIR SANS POSITIONNEMENT ALEATOIRE</u>	<u>3</u>
3.	<u>FONCTION D'INSERTION D'UN MESSAGE CLAIR AVEC POSITIONNEMENT ALEATOIRE</u>	<u>5</u>
4.	<u>FONCTION DE CHIFFREMENT DU MESSAGE PUIS INSERTION ALEATOIRE</u>	<u>6</u>
5.	<u>FONCTION D'EXTRACTION D'UN MESSAGE</u>	<u>7</u>
6.	<u>FONCTION DE DETECTION D'UN MESSAGE</u>	<u>8</u>
7.	<u>LES TFP ET TVP</u>	<u>12</u>

1. Rappel des objectifs

L'objectif est de tester différents aspects de la méthode dite *LSB* sur des images *BMP*.

Elle illustre parfaitement le principe général de la stéganographie.

Dans un premier temps vous constituerez un ensemble d'images *BMP*.

Ensuite vous implémenterez les fonctions suivantes dans un unique programme qui devra pouvoir recevoir un certain nombre d'arguments (CLI ou GUI).

La programmation sous forme de librairies est fortement recommandée.

- *Fonction d'insertion avec ou sans sélection aléatoire des pixels (option 1 et 2).*
- *Une fonction de chiffrement recevant une clef et traitant le clair avant insertion (option 3)*
- *Une fonction d'extraction (option 4).*
- *Une fonction de détection (option 5) pour déterminer si une image donnée en argument contient ou non un contenu dissimulé.*

Une fois votre programme terminé, validé et testé, vous mènerez les expériences suivantes :

- *Insertion de données en faisant varier le taux stéganographique, sur plusieurs images.*
- *Tests de détection sur ces différentes images avec votre propre fonction de détection/analyse.*
- *Vous construirez un graphique (type courbe ROC) pour évaluer les vrai positifs et les faux positifs en fonction du taux stéganographique.*

Vous écrirez un rapport de 10 pages maximum (les codes seront envoyés sous forme d'une archive avec le rapport) contenant vos résultats. La date limite de rendu est fixée au **15 avril**.

Le rapport et les codes source sont à envoyer sur l'adresse eric.filiol@univ-ubs.fr.

2. Fonction d'insertion d'un message clair sans positionnement aléatoire

Avant de commencer l'étude, il est important de rappeler la composition d'un bitmap car c'est grâce à elle que nous allons pouvoir sélectionner les champs de bits à modifier pour cacher un message donné.

Tout d'abord l'entête d'un fichier bitmap :

```
zsh> xxd 1.bmp > 1.hex ; cat 1.hex
```

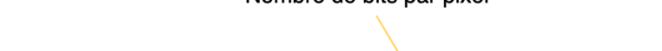
L'entête d'un fichier bitmap est décrite par les champs suivants :

	Signature	Taille du fichier	Réservé	Offset de l'image	
00000000:	424d	f6d4 0100	0000 0000	3600 0000	2800 BM.....6...(. 00000010: 0000 c800 0000 c800 0000 0100 1800 0000 00000020: 0000 c0d4 0100 c40e 0000 c40e 0000 0000 00000030: 0000 0000 0000 9249 bcec 8f99 3fc0 01feI....?.. 00000040: e5ee 96d2 c248 108d fbc3 6678 fdda dc30H....fx...0 00000050: d85e 5862 2829 4be5 8083 8009 da17 5dbe .^Xb()K.....].

Figure 1 - Composition d'un fichier BMP

Les champs juste après correspondent à l'entête de l'image (taille de l'entête image, largeur de l'image, hauteur, nombre de plans, **nombre de bits par pixel**, type de compression, taille de l'image, résolution horizontale / verticale, nombre de couleurs de la palette).

Nombre de bits par pixel



The diagram shows a 6x8 grid of hex values. An arrow points from the text 'Nombre de bits par pixel' to the value '0100' in the first row, seventh column. The values in the first row are 424d, f6d4, 0100, 0000, 0000, 3600, 0100, and 2800. The values in the second row are 0000, c800, 0000, c800, 0000, 0100, 1800, and 0000. The values in the third row are 0000, c0d4, 0100, c40e, 0000, c40e, 0000, and 0000. The values in the fourth row are 0000, 0000, 0000, 9249, bcec, 8f99, 3fc0, and 01fe. The values in the fifth row are e5ee, 96d2, c248, 108d, fbc3, 6678, fdda, and dc30. The values in the sixth row are d85e, 5862, 2829, 4be5, 8083, 8009, da17, and 5dbe.

00000000:	424d	f6d4	0100	0000	0000	3600	0100	2800	BM.....6...(. 00000010:	0000	c800	0000	c800	0000	0100	1800	0000 00000020:	0000	c0d4	0100	c40e	0000	c40e	0000	0000 00000030:	0000	0000	0000	9249	bcec	8f99	3fc0	01feI....?.. 00000040:	e5ee	96d2	c248	108d	fbc3	6678	fdda	dc30H....fx...0 00000050:	d85e	5862	2829	4be5	8083	8009	da17	5dbe	.^Xb()K.....].
-----------	------	------	------	------	------	------	------	------	----------------------------	------	------	------	------	------	------	------	------	--------------------	------	------	------	------	------	------	------	------	--------------------	------	------	------	------	------	------	------	------	----------------------------	------	------	------	------	------	------	------	------	-------------------------------	------	------	------	------	------	------	------	------	----------------

Figure 2 - Composition d'un fichier BMP ; data

C'est notamment l'information du nombre de bits par pixel qui va nous intéresser. En effet, cette information nous permet de déterminer le LSB d'un pixel directement à partir de sa taille. Ainsi, maintenant que l'on sait que les images générées sont composées de 24 bits par pixel ($18_{(16)} = 24_{(10)}$) il est facile d'implémenter un programme capable de modifier les LSB de chaque pixel de l'image.

Le reste correspond au corps de l'image.

Il est donc important de retenir que chaque pixel est donc représenté par 3 octets ($3 \times 8\text{bits} = 24\text{ bits}$).

Concernant l'implémentation en python, il va falloir extraire chaque pixel du corps de l'image pour les décomposer en bits et modifier le bit de poids faible de chacun des octets.

Pour commencer, nous allons convertir l'ensemble des caractères composant le message à cacher en valeur binaire que l'on pourra insérer par la suite :

```
[!] Conversion du message a cacher 'thebestpassword' en binaire : 011101000110100001100101011000100110010101110011011101000111000001100001011110011011101110111101110111011001001100100
```

Figure 3 - Conversion du message en binaire

Une fois le message converti en valeurs binaires, on peut désormais insérer ces derniers dans les données de l'image.

Cette partie présente une insertion d'une chaîne de caractère non chiffrée sans positionnement aléatoire dans l'image.

Ainsi, on va modifier les premières valeurs des pixels de l'image pour y insérer notre image.

Nous savons que le message doit être inséré dans les LSB de chaque couleur de chaque pixel. Pour faciliter la manipulation des données, nous allons utiliser la librairie Pillow.

Cette librairie permet l'extraction facile des données d'une image grâce à la fonction `getdata()`. Cette fonction renvoie un tableau contenant l'ensemble des pixels sous la forme de couleur RGB :

Pixel = (int(R), int(G), int(B))

Pour pouvoir y insérer nos bits de message, il faut convertir l'ensemble des données récupérées depuis l'image en données binaires.

A partir de là, il ne nous reste plus qu'à ajouter notre message sous la forme binaire.

Pour ce faire, on vient modifier le dernier bit pour chaque couleur R,G,B de chaque pixel.

La fenêtre de modification est définie par la longueur du message à insérer. Une fois que la totalité des bits du message a été insérée, on complète le tableau par les anciennes valeurs de l'image.

Ce tableau final est donc légèrement modifié visuellement car nous avons modifié que les bits de poids faible. La différence de couleur sera quasiment imperceptible.

L'insertion dans l'image s'effectue avec la librairie pillow.

Une fois la nouvelle image générée, on peut constater les maigres changements en analysant les valeurs RGB des premiers pixels ; dépendant des 0 ou des 1 qui ont été insérées, la valeur RGB va être incrémentée (0 → 1) ou décrétementée (1 → 0) :

L'image de gauche présente l'image originale tandis que celle de droite représente la nouvelle :

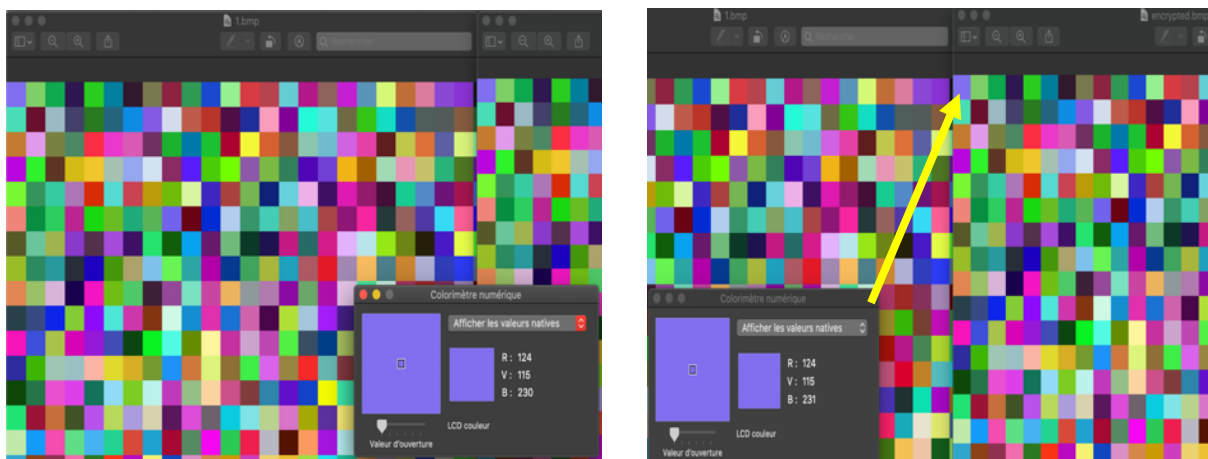


Figure 4 - Comparaison image après LSB

3. Fonction d'insertion d'un message clair avec positionnement aléatoire

La partie précédente présentait une fonction d'insertion à une position bien précise : le début de l'image. La détection de ce genre d'insertion est donc très facile. L'idée de cette partie est de présenter une fonction d'insertion d'un message clair avec un positionnement aléatoire.

Bien entendu, le but étant de cacher un message et donc de le retrouver par la suite, le positionnement aléatoire doit donc être maîtrisé par une graine qui sera générée par l'utilisateur.

Cette graine permettra, dépendant de sa valeur, de positionner le message à un endroit précis de l'image.

Le fonctionnement reste le même que précédemment, nous souhaitons cacher un message dans l'image mais cette fois-ci à un endroit aléatoire défini par la graine :

L'utilisateur est invité à rentrer une graine sous forme d'un message (clé stéganographique). Cette graine est ensuite utilisée dans la fonction d'ajout de message que l'on adapte en fonction de ce nouveau paramètre.

Une attention particulière à porter et que, pour pouvoir extraire le message, il sera nécessaire de connaître sa taille, ainsi le début de chaque message inséré se verra attribué d'un *header* comprenant la longueur du message. Cette fonction prend en paramètre la valeur binaire du message à insérer et retourne la chaîne binaire finale à importer dans l'image.

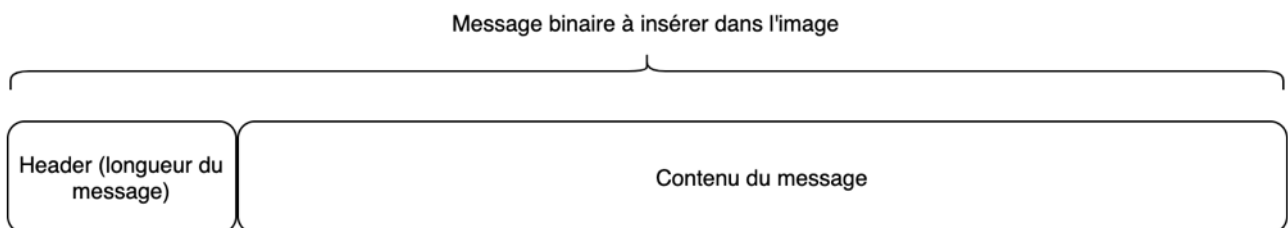


Figure 5 - Composition du message final à insérer

Ce détail sera utilisé lors de l'extraction, nous y reviendrons plus tard.

4. Fonction de chiffrement du message puis insertion aléatoire

Cette partie détaille l'insertion aléatoire d'un message chiffré. L'algorithme de chiffrement utilisé est l'algorithme AES.

Cet algorithme symétrique fonctionne avec une clé de chiffrement instanciée par l'utilisateur.

Ainsi, une première clé sera générée pour le chiffrement, et la deuxième, la clé stéganographique (graine) utilisée pour le positionnement :

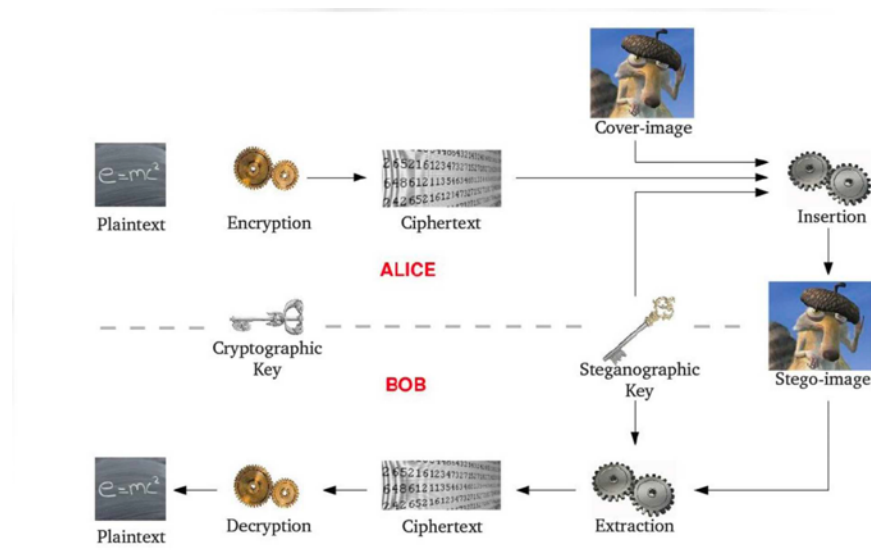


Figure 6 - Principe de fonctionnement général

5. Fonction d'extraction d'un message

Cette partie détaille l'extraction d'un message chiffré.

Pour pouvoir extraire et traiter notre message chiffré dans une plage immense de pixels, nous allons avoir besoin des deux clés précédemment générées.

La clé de chiffrement va être générée à partir d'un input utilisateur puis hashée ce qui nous donne, par exemple :

Clé de chiffrement :

`5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8`

Pour rappel, la clé stéganographique permet de retrouver la position du message dans l'image en fonction de l'index *i* correspondant au numéro de pixel.

La clé stéganographique étant la même au chiffrement qu'au déchiffrement, la graine générée par une fonction random avec la même clé donnera la position de départ sur l'image permettant de déchiffrer le message.

A partir de là, le header que l'on a inséré au début de notre chaîne binaire lors de l'insertion va être utilisé pour déterminer la longueur du message et pour indiquer à quel moment le message a été totalement découvert en fonction de ce paramètre.

Nous avons décidé de construire volontairement le *header* sur deux octets. Ainsi, la longueur maximale du message à cacher est de 2^{16} soit *65536 caractères*.

Premièrement, il est essentiel de construire la table des nombres aléatoires générée par la graine et ce, pour la longueur totale du *header* au minimum. Afin d'éviter les doublons lors de la génération (même si cela arrive rarement), une plage plus large que la taille du header a été construite (3x plus grande).

Ensuite, une fois la table initiale générée, on retire les doublons et on positionne les 16 premiers éléments du tableau de nombres aléatoires générés en index de la liste de l'ensemble des couleurs composant l'ensemble des pixels de l'image.

Ces 16 premiers éléments correspondant au header, la fonction va récupérer l'ensemble des LSB de chaque couleur ciblée, les assembler puis convertir l'ensemble en décimal pour connaître la valeur du header.

Une fois le header déterminé et connu, on réitère la même opération que ci-dessus (génération de la suite du tableau de nombres aléatoires, suppression des doublons, indexation, etc.) pour déterminer le message à récupérer.

Une fois le message chiffré récupéré, il suffit de le déchiffrer avec la clé de chiffrement connue.

6. Fonction de détection d'un message

Cette partie du TP fut la plus complexe à nos yeux. Le principe de détection que l'on a choisi repose sur le principe du SPA (Single Pair Analysis) qui représente pour nous la manière la plus efficace pour détecter si un message est caché dans une image ou non, via la technique d'insertion par LSB, comme vu précédemment.

Le principe du SPA repose, lors de l'insertion d'un message, sur la création et l'augmentation d'un biais statistique reposant sur des probabilités autour des LSB.

L'idée est relativement simple, on analyse deux pixels voisins, notés (a, b) . On concentre notre étude, dans un premier temps sur un seul canal de couleur, prenons le canal rouge pour commencer, et prenons une image BMP classique :

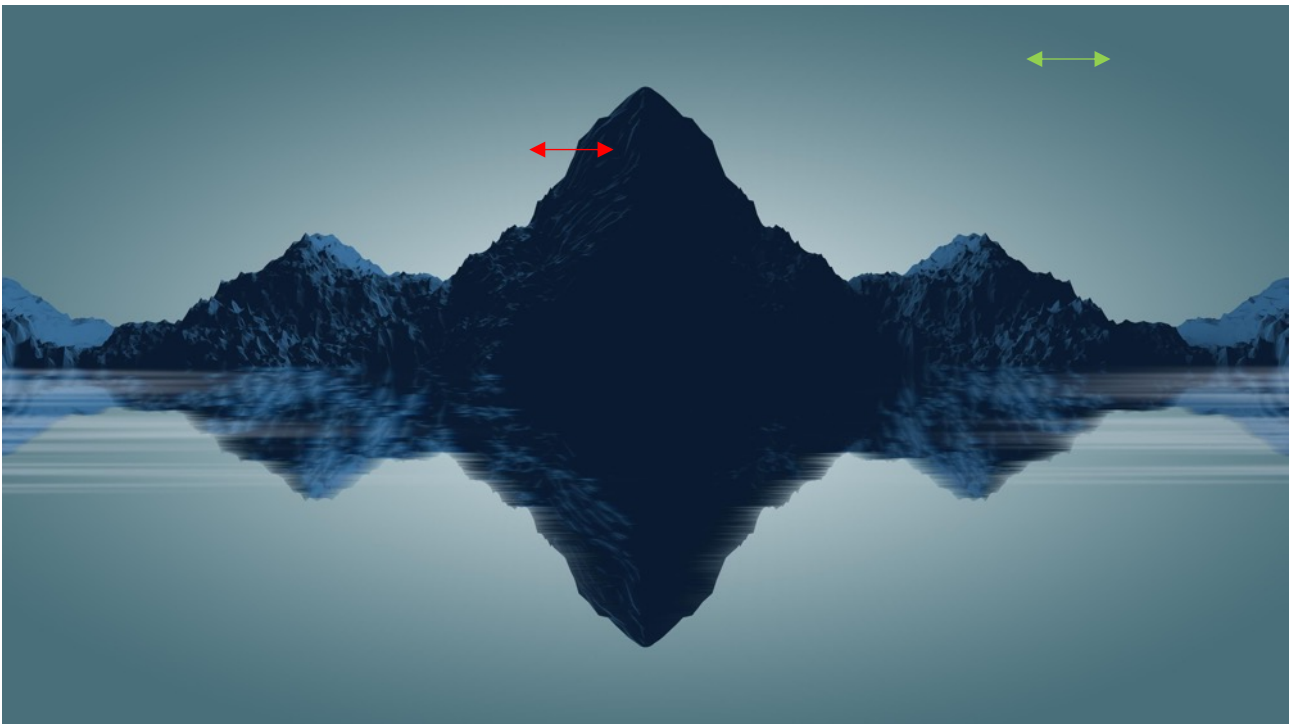


Figure 7 - Visualisation d'une transition forte / faible

Pour comprendre le principe, il est préférable de commencer par jeter un coup d'œil sur la composition même et la tendance des pixels de l'image.

On remarque dans la plupart des images que les changements de couleur entre deux pixels voisins, sont assez minimales, comme le montre la flèche verte. Ce type de transition entre deux pixels compose la majeure partie des transitions de l'image.

La flèche rouge, quant à elle, montre un changement brutal possible entre deux pixels voisins. Comme on peut le voir, ces transitions sont, à vue d'œil beaucoup moins nombreuses que les transitions précédentes, représentées en vert dans l'image (ici, seulement une plage est représentée par transition, il en existe bien plus)

Si on effectue un zoom sur une plage de transition verte (faible), voilà ce qu'on peut voir entre deux pixels voisins :

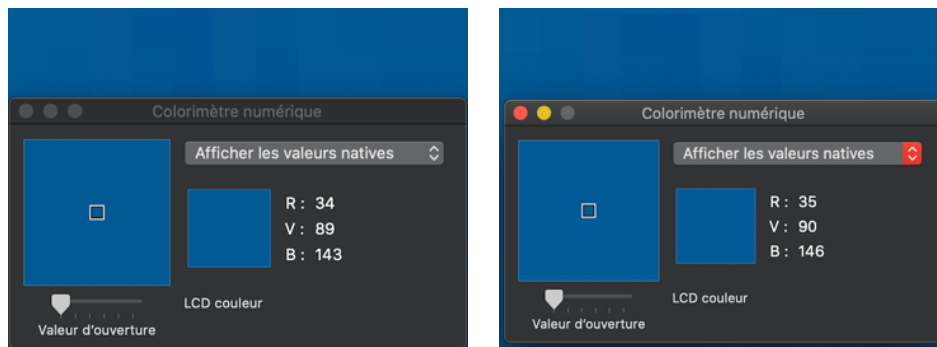


Figure 8 - Démonstration d'une transition faible

Prenons le pixel de gauche comme étant le pixel a , et le pixel de droite comme étant le pixel b .

Comme on peut le voir ici, l'une des propriétés fondamentales pour l'analyse SPA est montrée ici. Étant donné que la transition de pixel est « douce », les valeurs RGB sont légèrement différentes entre le pixel a et le pixel b .

Dans ce cas, on remarque que $a < b$.

Maintenant, concentrons-nous sur les valeurs des LSB, pour rappel, si une valeur décimale et impair, alors le LSB vaut 1 et si elle est paire, elle vaut 0

On déduit donc que $lsb(a) = 0$ (pour le canal rouge) et que $lsb(b) = 1$ (toujours pour le canal rouge).

Lors d'une insertion LSB, dépendant du bit qui sera inséré (soit un 0, soit un 1), la valeur du LSB va être modifiée ou va rester la même.

Par exemple, si le bit du message que l'on insère est un 1 dans $lsb(b)$, la valeur finale de $lsb(b)$ va rester la même. À l'inverse, si le bit du message que l'on insère est toujours un 1 mais qu'on le positionne dans $lsb(a)$, celui-ci va passer à 1 et sa valeur va augmenter.

On en conclut une propriété intéressante [1] :

- $lsb(x) = 0$ (état initial) → Insertion LSB → $lsb(x)$ augmente ou reste le même
- $lsb(x) = 1$ (état initial) → Insertion LSB → $lsb(x)$ diminue ou reste le même.

Autour de cette propriété, il faut également savoir que si $a < b$, étant donné que les valeurs RGB des pixels sont très proches, alors :

$P(lsb(a) = 0, lsb(b) = 1)$ est très élevé (lorsque l'image n'est pas encore modifiée),
cf. figure ci-dessus

Par conséquent, étant donné la propriété [1] après insertion d'un message par LSB, on en conclut que :

$P(lsb(a) = 0, lsb(b) = 1)$ **diminue**

$P(lsb(a) = 1, lsb(b) = 0)$ **augmente**

Enfin, dans le cas où $a > b$, on retrouve les probabilités inverses :

$P(lsb(a) = 0, lsb(b) = 1)$ **augmente**

$$P(\text{lsb}(a) = 1, \quad \text{lsb}(b) = 0) \text{ diminue}$$

C'est ainsi que, à travers l'insertion d'un message caché par LSB, nous allons pouvoir étudier, quantifier et déterminer un biais statistique aux vues des définitions précédentes.

Pour cela, nous allons étudier chaque pair de pixels voisins de l'image soupçonnée et nous allons calculer le nombre de fois les probabilités ci-dessus vont augmenter, et le nombre de fois où elles vont diminuer.

Nous allons stocker dans X les probabilités qui augmentent, et dans Y les probabilités qui diminuent. Dans une image normale, on constate que $\|X\| \approx \|Y\|$:

Bien entendu, les explications ci-dessus traitaient d'un seul canal, il va de soi qu'il faut appliquer le principe sur les deux autres canaux (bleu et vert).

A cause du biais statistique créé par l'insertion LSB on constate que $\|X\| < \|Y\|$.

Ainsi, plus le message est long, plus X diminue et Y augmente.

La différence entre les deux valeurs est ensuite quantifiée afin d'obtenir un taux de change estimé à travers l'équation quadratique suivante (l'intégralité des sources est disponible en Annexe[1]) :

$$0 = ax^2 + bx + c$$

Avec :

$$\begin{aligned} a &= (2 \cdot Z) \\ b &= 2(2 \cdot X - \text{width}(\text{height} - 1)) \\ c &= (Y - X) \end{aligned}$$

Équation du second degré oblige, on calcule :

$$\Delta = b^2 - 4AC \text{ avec } \Delta > 0$$

Enfin, on détermine nos 2 solutions telles que :

$$\begin{aligned} \beta_1 &= \frac{-b + \sqrt{\Delta}}{2a} \\ \beta_2 &= \frac{-b - \sqrt{\Delta}}{2a} \end{aligned}$$

β_1 et β_2 représente les taux de change. Pour la détection sur un canal de couleur, nous récupérons la valeur la plus petites des deux. C'est cette valeur qui représentera le taux de change final pour le canal de couleur donné.

On applique la même chose sur les deux autres canaux (vert et bleu).

On se retrouve donc avec 3 taux de change pour chacun des canal :

$$[\beta_R ; \beta_G ; \beta_B]$$

Pour récupérer un taux de change final noté β_{final} , on effectue la moyenne des sommes de chacun des taux de change ci-dessus, soit :

$$\beta_{\text{final}} = \frac{\beta_R + \beta_G + \beta_B}{3}$$

Enfin, ce taux de change final sera comparé au seuil de détection. Si le taux de change final est supérieur au seuil de détection défini dans le programme, alors on suppose que des données ont été cachées dans l'image. Dépendant de la taille de l'image, le seuil de détection peut être ajusté. Ici, on retient un seuil de **0.015%** pour une image de **660 x 440** pixels. On peut tester la détection avec l'image *test.bmp* fournie dans le dossier du projet.

7. Les TFP et TVP

Cette partie détaille la notion de faux positifs (FP) et vrais positifs (VP).

Le problème avec le taux de change calculé par la méthode SPA, c'est qu'il est très sensible lors de transitions fortes entre deux pixels voisins. Ainsi, une image comme celle-ci est plus complexe à traiter avec notre mode de détection, car les transitions fortes entre pixels voisins sont beaucoup plus présentes :



Figure 9 - Image possédant beaucoup de transitions fortes

NB : pour rappel, sur une image de taille 660 x 440, on avait déterminé un seuil à 0.015 qui fonctionnait plutôt bien, ici la taille étant réduite (400x300), le seuil de détection normal doit être aux alentours de 0.010 pour que la détection fonctionne bien.

Si on applique notre algorithme de détection sur cette image, le taux de change est très élevé sans même avoir de message caché au sein de celle-ci :

```
Tableau des taux de changes minimaux pour chaque canal (RGB) : [0.020262298423612927, 0.012323389556048127, 0.029160893143030622]  
Taux de change final estimé de l'image suspectée : 0.02058219370756389  
Seuil de détection choisi : 0.015  
Détection réussie, des données ont été insérées dans l'image
```

Ainsi, sur cette image, $\beta_{final} = 0.020$ est supérieur à notre seuil de détection initial de 0.015 alors qu'aucune donnée n'a été insérée à l'intérieur. (le seuil est sur calibré par rapport à la taille de notre image (voir NB ci-dessus), mais on le garde en guise d'exemple).

Pour le coup, pour un échantillon de 100 images semblables à celle-ci, on voit que le taux de faux positifs (TFP : image détectée comme cachant un message alors que ce n'est pas le cas) va augmenter avec une valeur de seuil à 0.020.

8. Test d'efficacité de l'algorithme de détection SPA

Cette partie présentera des courbes ROC présentant l'évolution des TFP (Taux de faux positifs) et des TVP (Taux de vrais positifs) d'un changement de seuil sur un échantillon de 100 images.

Notre échantillon d'images est constitué de la façon suivante :

- 100 images (/jpg/101.jpg → /jpg/200.jpg) contiendront chacune un message caché par insertion LSB.
- 100 images (/jpg/1.jpg → /jpg/100.jpg) ne contiendront aucun message et seront tout à fait normales.

Nous allons passer notre algorithme de détection sur les 100 images contenant un message caché.

- A noter que si l'image *contenant un message caché sera détecté* par l'algorithme, alors le taux de vrais positifs (*TVP*) pour ce groupe *augmentera*.
- A l'inverse, si l'image *contenant un message caché ne sera pas détecté* par l'algorithme, alors le taux de faux négatifs (TFN) pour ce groupe *augmentera*.

Idem pour le groupe « normal », nous allons passer notre algorithme de détection sur les 100 images ne contenant pas de message caché :

- Si l'image *ne contenant pas de message caché sera détecté* par l'algorithme, alors le taux de faux positifs (TFP) pour ce groupe *augmentera*.
- Si l'image *ne contenant pas de message caché ne sera pas détecté* par l'algorithme, alors le taux de vrais négatifs (TVN) pour ce groupe *augmentera*.

Ces affirmations faite, voilà à quoi doit ressembler une courbe ROC représentant notre algorithme de détection :

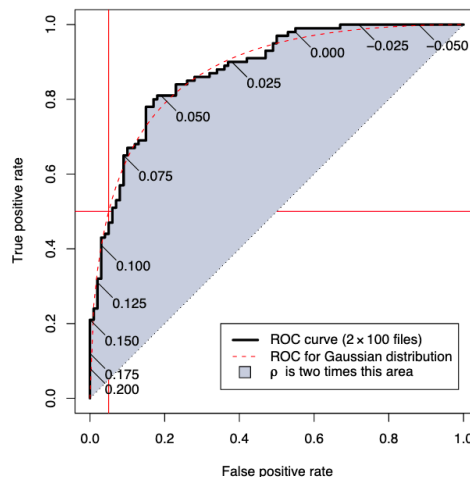


Figure 10 - Exemple d'une courbe ROC

Comme nous le voyons, plus le seuil de détection (représenté par les points relevés sur la courbe ci-dessus) diminue, plus le taux de vrais positifs augmente, ainsi que le taux de faux positifs.

En effet, imaginons le scénario suivant :

10 images sont considérées comme détectées car leur taux de change est égal à 0.016, pour un seuil à 0.015. (0.16 0.015 donc détection OK), peu importe si l'image contient un message ou pas.

10 images sont considérées comme non détectées car leur taux de change est égal à 0.012 pour un seuil à 0.015 ($0.012 < 0.015$ donc détection *NOK*), peu importe si l'image contient un message ou pas.

Si l'on baisse le seuil de 0.015 à 0.010, les 10 images précédemment non détectées vont à présent être détectées car $0.012 > 0.010$.

Les images ne contenant pas de message caché qui sont contenues dans cette plage vont être détectées ($0.012 > 0.010$) à l'instar de ceux contenant un message.

Ainsi, le taux de faux positifs (TFP) va augmenter tout comme le taux de vrais positifs (TVP), ce qui explique qu'avec un taux ~ 0.0001 (voir même 0), toutes les images seront détectées (les vrais positifs, tout comme les faux), et donc le $TFP = TVP = 1$.

Le taux de vrais négatifs (TVN) et le taux de faux négatifs (TFN) ne seront pas utilisés pour la génération de la courbe ROC finale.

Cette conclusion explique l'allure de la courbe que l'on retrouve implémentée dans notre programme Python :

Pour comprendre cette courbe, il faut savoir que les images sélectionnées ont une taille de 700 x 500 pixels. Nous avons opté pour un taux stéganographique de l'ordre de 0.25. Ainsi la taille du message est calculé de la façon suivante (pour arriver à ce taux) :

$$\text{longueurMessage} = \frac{\text{bpp} \times \text{largeurImage} \times \text{hauteurImage} \times 3}{8}$$

A des fins d'optimisation et de choix volontaire de notre part pour que le message respecte bien le taux stéganographique défini, nous avons décidé de retirer la fonction de chiffrement ainsi que l'ajout du header, ces dernières ajoutant des bits non désirés en plus dans notre message.

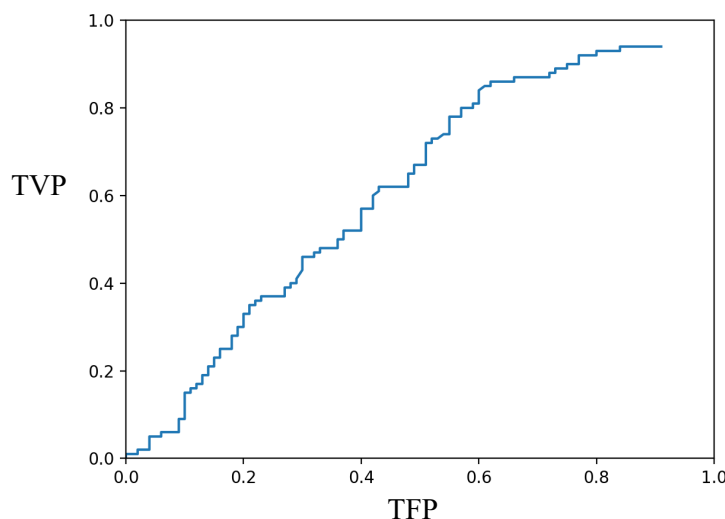


Figure 11 - Courbe ROC générée avec notre échantillon (200 images)

Cette courbe présente donc la tendance générale des taux de faux positifs (TFP) et taux de vrais positifs (TVP) au fur et à mesure que le seuil est proche de 0.

Comme on le voit, la définition annoncée précédemment est remarquable sur cette figure : la valeur des TFP et TVP se rejoint à 1 lorsque le seuil est proche de 0 car toutes les images sont détectées comme cachant un message (ce qui n'est bien entendu par toujours le cas).

9. Conclusion

Ce TP nous a permis d'en apprendre beaucoup plus sur la stéganographie en particulier dans la composition d'une image, d'un pixel et de l'insertion / extraction par méthode LSB.

Par ailleurs, le choix de s'être tourné vers une méthode de détection SPA nous a permis de comprendre les notions d'analyse assez complexes de prime abord.

Des améliorations restent à apporter dans notre programme final (comme l'optimisation lors de l'insertion LSB par méthode aléatoire, qui prend du temps pour les 100 images sélectionnées pour un long texte donné).

Cela est dû au fait qu'on régénère la même table aléatoire pour les 100 images ce qui prend un temps considérable. Nous pouvons, par exemple, la générer une seule fois lors de la première image, et l'utiliser pour les 99 autres.

Le seul petit point noir étant le manque de documentation ainsi que la nécessité d'un temps assez considérable pour pouvoir réaliser ce TP de manière convenable.

Enfin, nous avons toutefois pris du plaisir à mener à bien ce projet malgré sa difficulté.

**Merci pour votre lecture,
Kévin MOREAU et Laurent GRAFF**

Annexe :

- Fridrich, Jessica. *Steganography in Digital Media: Principles, Algorithms, and Applications*. Cambridge University Press, 2010. https://books.google.fr/books?id=wcAZ-QEthqkC&pg=PA185&lpg=PA185&dq=change+rate+lsb&source=bl&ots=Val9nSFyu8&sig=ACfU3U0LG0vK5hywZv2sh3_8dIWF5mGo6w&hl=fr&sa=X&ved=2ahUKewjVh-D0tp_pAhVGOhoKHc8wDuMQ6AEwDnoECACQAQ#v=onepage&q=change%20rate%20lsb&f=false

Algorithm 11.1 Sample Pairs Analysis for estimating the **change rate** from a stego image. The constant γ is a threshold on the test statistic $\hat{\beta}$ set to achieve $P_{FA} < \epsilon_{FA}$, where ϵ_{FA} is a bound on the false-alarm **rate**.

```
// Input  $M \times N$  image  $\mathbf{x}$ 
// Form pixel pairs
 $\mathcal{P} = \{(\mathbf{x}[i, j], \mathbf{x}[i, j + 1]) | i = 1, \dots, M, j = 1, \dots, N - 1\}$ 
 $x = y = 0; \kappa = 0;$ 
for  $k = 1$  to  $M(N - 1)$  {
     $(r, s) \leftarrow k$ th pair from  $\mathcal{P}$ 
    if  $(s \text{ even} \ \& \ r < s) \text{ or } (s \text{ odd} \ \& \ r > s) \{x = x + 1;\}$ 
    if  $(s \text{ even} \ \& \ r > s) \text{ or } (s \text{ odd} \ \& \ r < s) \{y = y + 1;\}$ 
    if  $(\lfloor s/2 \rfloor = \lfloor r/2 \rfloor) \{\kappa = \kappa + 1;\}$ 
}
if  $\kappa = 0$  {output('SPA failed because  $\kappa = 0$ '); STOP}
 $a = 2\kappa; b = 2(2x - M(N - 1)); c = y - x;$ 
 $\beta_{\pm} = \text{Re}((-b \pm \sqrt{b^2 - 4ac})/(2a));$ 
 $\hat{\beta} = \min(\beta_+, \beta_-);$ 
if  $\hat{\beta} > \gamma$  {
    output('Image is stego');
    output('Estimated change rate = ',  $\hat{\beta}$ );
}
```

- ROC Curves for Steganalysts, Andreas Westfeld,
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.5156&rep=rep1&type=pdf>