L'ART DU CAMOUFLAGE INFORMATIQUE

Tristan Colombo, Relecture Léo Ducas-Binda et Christian Surace

De tous temps les hommes ont eu besoin de dissimuler des informations, de communiquer en s'assurant qu'il n'y ait pas d'interception. Les systèmes étaient forts ingénieux et avec l'informatique ils sont devenus plus ou moins simples à mettre en place.

Transmettre des informations à l'insu de l'ennemi est une des bases de toute stratégie militaire. Imaginez une tentative d'encerclement lancée dans *Age of Empires* (nostalgie...) et que votre adversaire soit au courant de ce mouvement... Vous allez vous retrouver face à une horde de lanciers ! C'est donc tout naturellement dans l'histoire des guerres que l'on retrouve le plus de stratagèmes permettant de communiquer sans que cela ne se voit. Je commencerai cet article par quelques exemples historiques avant d'étudier la dissimulation d'informations dans des fichiers informatique en analysant deux algorithmes représentatifs du domaine.

1 D'un point de vue historique

Les exemples sont nombreux et je ne ferai ici qu'une synthèse des moyens les plus marquants.

1.1 Les esclaves

Commençons par les grecs puisque nous leurs devons le mot stéganographie, de « steganos » caché et « graphein » l'écriture (d'après le dictionnaire Larousse). L'historien Hérodote (484 à 445 avant J.-C.), relate plusieurs exemples de communication cachée (Livre V - Terpsichore). Le premier de ces exemples remonte à 499 avant J.-C. : Histiée, tyran de Milet, était retenu prisonnier à Suse par le roi des perses et il désirait transmettre un message à Milet. Il eut alors l'idée de raser un esclave, écrire son message sur le crâne de celui-ci et attendre que ses cheveux repoussent. Il l'envoya alors à Milet sachant qu'il serait fouillé mais que son message passerait inaperçu. Les historiens sont assez sceptiques quant à cet exemple, les esclaves étant à l'époque rasés et portant un bonnet, un esclave chevelu se serait remarqué immédiatement. Toujours est-il que l'idée est là et qu'il s'agit de la première trace de stéganographie.

1.2 La tablette de cire

Toujours le même Hérodote relate l'histoire de Démarate, ancien roi de Spartes réfugié en Perse qui, apprenant un projet d'invasion de la Grèce, fit parvenir un message à Sparte en prenant une tablette dont il gratta la cire pour écrire à même le bois et qu'il recouvrit ensuite à nouveau de cire : la tablette était vierge en apparence et le récipiendaire n'avait plus qu'à la gratter pour voir le message. Bien sûr, il valait mieux transporter plusieurs tablettes en même temps, le transport d'une seule tablette, qui plus est vierge, aurait sans doute éveillé les soupçons...

1.3 Le jus de citron

Qui ne s'est pas amusé étant enfant à écrire un message avec du citron pour le faire apparaître en le passant devant une flamme ? Il s'agit du principe de l'encre sympathique : on utilise un produit qui réagit dans certaines conditions pour redevenir visible. Dans le cas du jus de citron, l'acide citrique qu'il contient brûle avant le papier et laisse donc une trace marron sur celui-ci. Bien sûr il est recommandé de ne pas trop chauffer le document pour pouvoir le lire...

1.4 Un message dans un message

Les exemples ne manquent pas dans ce domaine (recherchez par exemple la correspondance entre Alfred de Musset et Georges Sand). L'abbé Jean Trithème (1462 à 1516) imagina dans les

	R	G	B
Pixel 1	10101010	11100010	11111001
Pixel 2	11111111	00000001	00000001
Pixel 3	11111111	11011111	00000100

Pour cacher la lettre **A**, il faut retrouver son code ASCII (**65**) et le convertir en binaire : **01000001**. Maintenant il suffit de remplacer les bits de poids faible des composantes des pixels précédents par les bits codant pour le caractère **A**. Le tableau suivant montre cette modification.

	R	G	B
Pixel 1	10101010	11100011	11111000
Pixel 2	11111110	00000000	00000000
Pixel 3	11111110	11011111	00000100

La figure 1 montre ces trois pixels avant et après modification. Vous ne voyez pas de différence ? Vraiment ? Ne prenez pas rendez-vous avec un ophtalmologue, les différences sont si minimes que l'on ne peut les discerner (par exemple le bleu du pixel 1 passe de **(170, 226, 249)** à **(170, 227, 248)**).

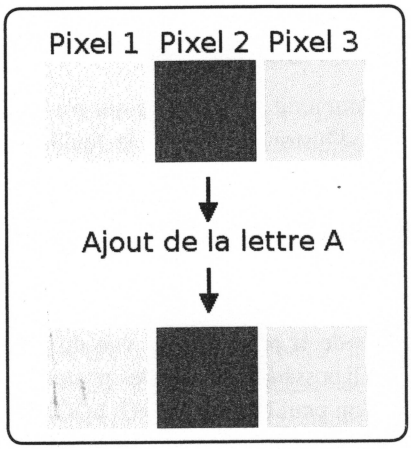

Fig. 1: Modification des bits de poids faible des composantes de trois pixels pour y dissimuler la lettre A.

On peut également stocker les lettres sur moins de pixels. C'est le **double LSB** ou le **triple LSB** :

- double LSB : un octet sur deux pixels. Les composantes du premier pixel contiennent chacune deux bits (les deux bits de poids faible) et le second pixel contient les deux derniers bits : un sur la composante rouge et un sur la composante bleue ;
- triple LSB : un octet sur un pixel. On stocke trois bits sur la composante rouge, deux bits sur la composante verte et trois bits sur la composante bleue.

Ces deux dernières adaptations du LSB nécessitent moins de pixels mais « détériorent » plus l'image.

Avec ce système, il faudra également transmettre le nombre de caractères du message (en le cachant lui aussi dans les bits de poids faible des premiers pixels de l'image). La taille minimale (en pixels) d'une image devant contenir un message de **n** lettres est donc liée à cette taille. Dans la suite j'ai choisi de me concentrer sur l'algorithme LSB et de considérer une taille maximale de message de 511 caractères (codé sur deux octets, donc six pixels). Il me faudra donc des images possédant au minimum **3n + 6** pixels.

Voici maintenant l'algorithme général permettant de dissimuler un message avec la technique du LSB :

```
t <- Taille du message
t_bin <- Conversion binaire de t
msg <- Message
img <- Image
code <- Copie de img
tete_ecriture <- (p0, 0) (pixel 0,
composante 0 pour rouge)

Pour chaque bit b de t_bin Faire
    Remplacer par b le bit de poids
faible de tete_ecriture
    tete_ecriture <- tete_ecriture
suivante (pixel ou composante)
Avancer tete_ecriture jusqu'au pixel
suivant

Pour chaque caractère car de msg Faire
    car_bin <- Conversion binaire de car
    Pour chaque bit b de car_bin Faire
        Remplacer par b le bit de poids
faible de tete_ecriture
        tete_ecriture <- tete_ecriture
suivante (pixel ou composante)
    Avancer tete_ecriture jusqu'au pixel
suivant

Écrire l'image code sur le disque
```

Pour une mise en pratique de cet algorithme nous allons utiliser Python et le module **PIL** (*Python Imaging Library*) qui va nous permettre de manipuler très simplement les pixels :

```
001: from PIL import Image
```

Pour commencer il faut bien entendu importer les éléments du module PIL dont nous aurons besoin. Ici il s'agit de **Image**.

```
004: def playHead(max, pixel=0,
component=0):
005:     current = (pixel, component)
006:     i = 0
007:     while i < max:
008:         yield current
009:         if current[1] == 2:
010:             current = (current[0]
+ 1, 0)
011:         else:
012:             current = (current[0],
current[1] + 1)
013:         i += 1
```

La fonction **playHead()** sert de tête de lecture. Elle retourne un tuple composé du numéro de pixel et de

Lettre	Sentence	Lettre	Sentence
A	dans les cieux	N	en paradis
B	à tout jamais	O	toujours
C	un monde sans fin	P	dans la divinité
D	en une infinité	Q	dans le déité
E	à perpétuité	R	dans la félicité
F	sempiternel	S	dans son règne
G	durable	T	dans son royaume
H	sans cesse	U, V, W	dans la béatitude
I, J	irrévocablement	X	dans la magnificence
K	éternellement	Y	au trône
L	dans la gloire	Z	en toute éternité
M	dans la lumière		

années 1500 un système permettant d'écrire des messages sous forme de textes religieux (abscons par définition) : il faisait correspondre aux lettres de l'alphabet des sentences et il les articulait ensuite à l'aide de mots « inutiles » (Jean Trithème, *« Polygraphia »*). Le tableau suivant montre un alphabet utilisé dans l'un de ses *Ave Maria* : voir tableau ci-dessus.

Ainsi *GNU/Linux* s'écrirait :

Ô durable en paradis

Et dans la béatitude !

Mais dans la gloire, irrévocablement,

Et en paradis, dans la béatitude,

Il trône dans la magnificence.

1.5 Et les signatures par points des imprimantes ?

Rappelez-vous de l'annonce de l'EFF indiquant que les constructeurs d'imprimantes faisaient en sorte que leur matériel marque les pages imprimées à l'aide de points minuscules **[1]**. L'information contenue par cette série de points est codée mais il s'agit là encore de stéganographie : le message est là, sous nos yeux, et nous ne le voyons pas !

1.6 Prison break

Pour finir un exemple qui n'a plus rien d'historique : dans le feuilleton américain **Prison break** créé par Paul Scheuring, le héros, Michael Scofield se fait faire un tatouage incorporant les plans de la prison dans laquelle son frère a été incarcéré (passages souterrains de la prison, carte vue du ciel, etc). Il possède par exemple un tatouage où l'on peut lire « bOLShOI bOOZE » mais qui lu à l'envers donne 32009 1045709 soit la position 32°00'9" - 104°57'09".

2 Un premier algorithme de stéganographie

Il ne faut pas confondre stéganographie et cryptographie : ici le message circule en clair devant les yeux de tous sans qu'on le remarque (bien qu'il soit possible, en plus, de le crypter). Le tatouage (*watermarking* en anglais) est également différent : on ajoute une information, visible ou non, et il est ensuite essentiel qu'elle résiste le plus possible aux modifications du support. Ce procédé est par exemple utilisé pour ajouter un copyright indélébile sur les fichiers comme les pdf.

Le tatouage est issu de la stéganographie mais vous vous doutez bien qu'il ne s'agit pas des mêmes algorithmes. Pour ne pas tout mélanger, dans le cadre de cet article nous allons nous restreindre à quelques méthodes de stéganographie « pure ».

2.1 Least Significant Bit ou LSB

On se place ici dans le cadre d'une image à modifier pour y intégrer un message. La technique du LSB simple permet de cacher un octet en utilisant trois pixels codés en RGB (rouge, vert, bleu). Pour chaque composante d'un pixel, on utilise un code compris entre `0` et `255`. En binaire cela donne un nombre entre `00000000` et `11111111`. Sur un pixel, si l'on modifie le bit de poids faible, on peut donc stocker une information sur trois bits (un bit sur le rouge, un bit sur le vert et un bit sur le bleu). Donc pour coder un octet, représentant une lettre, il nous faut bien trois pixels (la composante bleu du dernier pixel ne sera pas utilisée).

Supposons que les trois pixels de notre image soient les suivants :

la composante que l'on traite (les paramètres **pixel** et **component** permettent d'initialiser les valeurs de départ). Le code utilisé pour les composantes est **0** pour rouge, **1** pour vert et **2** pour bleu. Le fait d'utiliser l'instruction **yield** en ligne 8 permet de créer un générateur : nous créons une variable et en appelant la fonction **next()** nous obtenons les nouvelles valeurs (le paramètre **max** limite la génération de ces valeurs). Voici un exemple d'utilisation décorrélé de notre programme :

```
>>> g = playHead(12)
>>> print(next(g))
(0, 0)
>>> print(next(g))
(0, 1)
>>> print(next(g))
(0, 2)
>>> print(next(g))
(1, 0)
...
```

Revenons à notre code :

```
016: def linear2matrix(n, cols):
017:     return (n % cols, n // cols)
```

Les images sont lues sous forme de tableau (ou matrice) de pixels. Pour faciliter notre travail, il est plus simple de considérer les pixels de manière linéaire. L'objectif de la fonction **linear2matrix()** est donc de fournir les numéros de colonne et de ligne d'un pixel dont la position linéaire **n** est passée en paramètre. Pour effectuer ce calcul il faut connaître le nombre de colonnes de la matrice et celui-ci est donc passé également en paramètre en ligne 16 (paramètre **cols**).

```
020: def writeBitInImage(bit, img, head, cols):
021:     (n_pixel, component) = next(head)
022:     (col, row) = linear2matrix(n_pixel, cols)
023:     pixel = img.getpixel((col, row))
024:     new_pixel = (
025:         int(bin(pixel[0])[:7] + bit, 2) if component == 0
else pixel[0],
026:         int(bin(pixel[1])[:7] + bit, 2) if component == 1
else pixel[1],
027:         int(bin(pixel[2])[:7] + bit, 2) if component == 2
else pixel[2]
028:     )
029:     img.putpixel((col, row), new_pixel)
030:     return component
```

Les informations sont écrites sur le bit de poids faible de la composante issue du pixel de la tête de lecture. La fonction **writeBitInImage()** effectue cette tâche en prenant en paramètre le **bit** à écrire, l'**image**, la tête de lecture **head** et la largeur de l'image correspondant au nombre de colonnes **cols** de la matrice de pixels. En ligne 21 on récupère donc le numéro de pixel courant et la composante sur laquelle on doit écrire. Cela passe par un appel à **next()** sur la variable **head** qui est un générateur engendré par un appel à **playHead()** comme nous le verrons plus loin. En ligne 22 la fonction **linear2matrix()** nous fournit la position du pixel dans la matrice image, ce qui nous permet de lire le pixel à modifier en ligne 23. Dans les lignes 24 à 28 nous modifions le bit de poids faible de la composante stockée dans **component** (pour les autres composantes nous conservons la valeur du pixel notée **pixel[i]**). En ligne 29 nous pouvons remplacer le pixel dans l'image et en ligne 30 nous renvoyons la valeur de la composante actuelle (ce qui permet de se placer ensuite sur le pixel suivant en sautant les composantes inutiles).

Note

Sur des images gérant la transparence, il est possible d'utiliser le canal alpha de chaque pixel pour y stocker un bit de plus (du coup un caractère est stocké sur deux pixels). Si vous voulez utiliser ce canal il faudra simplement ajouter la gestion d'une quatrième composante, **getpixel()** renvoyant un tuple de quatre éléments.

```
033: def readBitInImage(img, head, cols):
034:     (n_pixel, component) = next(head)
035:     (col, row) = linear2matrix(n_pixel, cols)
036:     pixel = img.getpixel((col, row))
037:     return (bin(pixel[component])[-1], component)
```

readBitInImage() permet de lire le bit de poids faible de la composante du pixel indiqué par la tête de lecture. La valeur de retour est ce bit (utilisation de l'indice **-1** renvoyant le dernier élément de la chaîne générée par la conversion en binaire **bin()** de **pixel[component]**) et la valeur de la composante actuelle.

```
040: def lsb(msg, img, filename='out.png', verb=True):
041:     size = len(msg)
042:     image = Image.open(img)
043:     (width, height) = image.size
044:     nb_pixels = width * height
045:     head = playHead(nb_pixels)
046:     encoded = image.copy()
047:
048:     if nb_pixels < (3 * size + 6):
049:         print("Image trop petite pour y intégrer le
message")
```

```
050:         exit(1)
051:
052:     # Insertion de la taille du message
053:     b_size = '{:016d}'.format(int(bin(size)[2:]))
054:     if verb:
055:         print("Écriture de la taille : {} soit {}
b".format(size, b_size))
056:     for bit in b_size:
057:         last_component = writeBitInImage(bit, encoded, head,
width)
058:
059:     # On avance la tête jusqu'au prochain pixel
060:     while last_component != 2:
061:         (pixel, last_component) = next(head)
062:
063:     # Insertion du message
064:     for c in msg:
065:         car = '{:08d}'.format(int(bin(ord(c))[2:]))
066:         if verb:
067:             print("Écriture du caractère : {} soit {}
b".format(c, car))
068:         for bit in car:
069:             last_component = writeBitInImage(bit, encoded,
head, width)
070:         # On avance la tête jusqu'au prochain pixel
071:         while last_component != 2:
072:             (pixel, last_component) = next(head)
073:
074:     encoded.save(filename)
```

La fonction **lsb()** est chargée d'incorporer le message **msg** dans l'image **img** et de sauvegarder le résultat dans une image **filename** (par défaut **out.png**). Dans les lignes 41 à 46 nous récupérons la taille du message, nous lisons l'image et récupérons ses dimensions, nous calculons le nombre de pixels de l'image, nous créons la tête de lecture en l'initialisant sur la composante rouge du premier pixel puis nous copions l'image dans **encoded**. S'il n'y a pas assez de place pour insérer le message nous le signalons et nous sortons du programme. La suite suit l'algorithme en utilisant les différentes fonction que nous avons précédemment créées. En ligne 74 nous sauvegardons la nouvelle image contenant le message.

```
076: def detect_lsb(img, verb=True):
077:     image = Image.open(img)
078:     (width, height) = image.size
079:     nb_pixels = width * height
080:     head = playHead(nb_pixels)
081:     size = ''
082:     msg = ''
083:
084:     # On récupère la taille du message
085:     for i in range(16):
086:         bit, last_component = readBitInImage(image, head,
width)
087:         size += bit
088:     final_size = int(size, 2)
089:     if verb:
090:         print("Taille : {}b soit {} caractères".format(size,
final_size))
091:
092:     # On avance la tête jusqu'au prochain pixel
093:     while last_component != 2:
094:         (pixel, last_component) = next(head)
095:
096:     # On récupère le message
097:     for i in range(final_size):
098:         car = ''
099:         for j in range(8):
100:             bit, last_component = readBitInImage(image,
head, width)
101:             car += bit
102:         final_car = chr(int(car, 2))
103:         msg += final_car
104:         if verb:
105:             print("Caractère: {}b soit {}".format(car,
final_car))
106:         # On avance la tête jusqu'au prochain pixel
107:         while last_component != 2:
108:             (pixel, last_component) = next(head)
109:
110:     return msg
```

Suivant le même mécanisme que la fonction **lsb()**, la fonction **detect_lsb()** lit un message dissimulé dans l'image **img** passée en paramètre. Il faut bien sûr commencer par récupérer la taille du message (lignes 84 à 94) puis nous effectuons une boucle sur le nombre de caractères trouvés (lignes 96 à 108).

```
113: if __name__ == '__main__':
114:     lsb("GNU/Linux Magazine", "lena.png")
115:     msg = detect_lsb("out.png")
116:     print(msg)
```

En appliquant ce programme sur la traditionnelle photo de Lena (voir encadré), nous pourrons constater que le message est bien enfoui et indétectable à l'œil nu. Toutefois, avec un message un peu conséquent, une analyse colorimétrique peut montrer une anomalie comme le montre la figure 2.

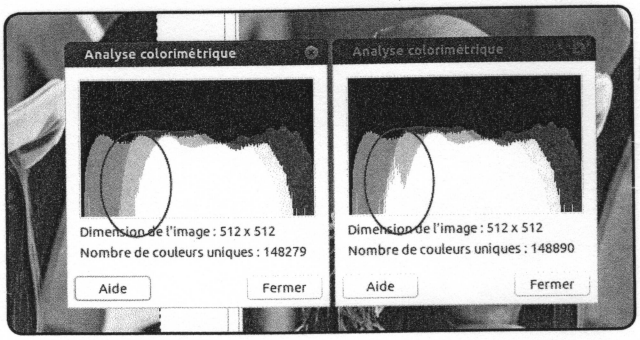

Fig. 2: Analyse colorimétrique de la photo de Lena originale (à gauche) et de celle contenant un message caché (à droite).

Évidemment, toutes les informations se trouvent en début de fichier... L'algorithme F5 propose une amélioration qui va disperser les informations sur tout le fichier.

Note

À PROPOS DE LENA

Les algorithmes de traitement d'image sont presque toujours appliqués sur la même photo de Lena (voir figure 3). Il s'agit en fait d'un morceau d'une photo de Lena Sjööblom **[2]**, playmate du numéro de novembre 1972 du magazine Play-boy. La photo complète **[3]** est un peu plus dénudée...

Mais pourquoi cette photo ? David Munson, professeur en informatique à l'Université du Michigan, l'explique de la manière suivante **[4]** :

« Tout d'abord, cette image contient un mélange intéressant de détails, de régions uniformes, et de textures, ce qui permet de bien tester les différents algorithmes de traitement d'image. C'est une bonne image de test ! Ensuite, « Lenna » [nom donné dans l'article original de Play-boy] est l'image d'une femme attirante. Ce n'est pas une surprise que la communauté de la recherche dans le traitement d'image (principalement masculine) gravite autour d'une image qu'elle trouve attirante. »

Fig. 3: Photo de Lena

Note

APPLICATION DU LSB DANS LA « VRAIE VIE »

Le 3 février 2012, l'ANSSI (Agence Nationale de la sécurité des systèmes d'information) à publier un nouveau logo en y cachant des données... Le reste de l'histoire se trouve dans l'article de Pierre Bienaimé accessible gratuitement sur notre site **[5]**.

2.2 Stéganographie facile avec le logiciel Steghide

Steghide est un programme disponible dans les dépôts de la plupart des distributions Linux qui, même s'il n'est pas tout récent, vous permettra d'effectuer des tests de stéganographie sans avoir à vous lancer dans le développement de vos propres outils. Sous Debian et dérivés il suffira de passer par **aptitude** pour l'installer :

```
# aptitude install steghide
```

Il faut noter que Steghide ne supporte que les formats de fichier jpeg, bmp, wav et au. Prenons donc une image de Lena en jpeg, créons un petit fichier **msg.txt** contenant une phrase et lançons Steghide :

```
$ steghide embed -e none -cf lena.jpg -ef msg.txt
Entrez la passphrase:
Entrez à nouveau la passphrase:
camouflage des données de "msg.txt" dans "lena.jpg". Terminé.
```

Nous pouvons constater que Steghide protège par défaut l'accès à vos données. Pour avoir des informations sur l'image contenant maintenant le message enfoui, il faut faire :

```
$ steghide --info lena.jpg
"lena.jpg":
  format: jpeg
  capacité: 5,2 KB
Essayer d'obtenir des informations à propos des données
incorporées ? (o/n) o
Entrez la passphrase:
  fichier à inclure "msg.txt":
    taille: 20,0 Byte
    cryptage: non
    compression: oui
```

Et pour retrouver le message :

```
$ steghide extract -sf lena.jpg
Entrez la passphrase:
```

```
le fichier "msg.txt" existe déjà. l'écraser ? (o/n) o
Écriture des données extraites dans "msg.txt".
$ more msg.txt
GNU/Linux Magazine
```

Les différentes options sont disponibles sur la page de man et il est même possible de choisir l'algorithme utilisé pour dissimuler les informations. À ce propos, nous pouvons d'ailleurs maintenant aborder un deuxième algorithme, l'algorithme F5.

3 Un algorithme plus complexe : F5

Comme souligné précédemment, le problème de l'algorithme LSB est d'enfouir les données de manière continue au début de l'image. WestFeld **[6]** a proposé un algorithme, baptisé F5 (en référence à ses précédents algorithmes F3 et F4), qui permet de répartir l'information dans une image au format jpeg. Cette méthode intervenant au niveau de la compression en jpeg, elle est beaucoup plus longue à implémenter et nous nous bornerons à un aperçu de ses aspects théoriques déjà fort complexes car faisant intervenir de nombreuses notions. Pour commencer, il faut savoir comment compresser une image au format jpeg.

3.1 La compression jpeg

L'algorithme de compression jpeg, dont le nom provient du *Joint Photographic Expert Group* qui créa la norme jpeg **[7]**, est composé de six étapes, données dans la suite.

3.1.1 Transformation de couleurs, passage du mode RGB au mode YCbCr

Nous avons vu que chaque pixel d'une image était codé suivant trois composantes rouge, vert ou bleu soit RGB. Dans cette étape cette information est transformée en luminance (Y), chrominance bleue (Cb) et Chrominance rouge (Cr). Sans trop rentrer dans les détails il s'agit simplement de changer d'espace colorimétrique. Les formules permettant de passer de RGB à YCbCr sont très simples **[8]** :

$$Y = 0{,}299\ R + 0{,}587\ G + 0{,}114\ B$$
$$Cb = -0{,}1687\ R - 0{,}3313\ G + 0{,}5\ B + 128$$
$$Cr = 0{,}5\ R - 0{,}4187\ G - 0{,}0813\ B + 128$$

3.1.2 Sous-échantillonnage

Pour compresser l'image, celle-ci va être dégradée en effectuant une moyenne de la chrominance (en général par blocs de 8 pixels). L'œil humain n'est pas capable de discerner cette dégradation sur des photos.

Le sous-échantillonnage peut être plus ou moins puissant et donc réduire plus ou moins le nombre de pixels.

3.1.3 Découpage en blocs de pixels

Lors de cette étape l'image est découpée en blocs de 64 pixels (blocs de 8 pixels de côté). Attention, en fonction du sous-échantillonnage, un bloc de 8 x 8 pixels peut correspondre à un bloc plus gros (par exemple 16 x 16 pixels) dans l'image d'origine.

3.1.4 Application de la DCT (Transformée en Cosinus Discrète)

La DCT permet de passer du domaine spatial au domaine fréquentiel, c'est-à-dire que l'image sera décomposée en fréquences, en sommes de cosinus comme son nom l'indique, ce qui permettra de la compresser. En effet, les fréquences mesurent les changements d'intensité des pixels et comme les valeurs des pixels d'une image sont généralement « linéairement » proches, seuls les changements rapides d'intensité du pixel dénotent des hautes fréquences et il y en a, en général, peu dans une image. L'œil humain étant plus sensible aux basses fréquences, on peut représenter toute l'image en utilisant très peu de coefficients.

La DCT est une transformée de Fourier qui s'applique à une matrice carrée et donne comme résultat une matrice de même dimension où les basses fréquences sont placées en haut à gauche alors que les hautes fréquences sont placées en bas à droite. Le traitement se fait en général par blocs de 64 pixels (8 pixels de côté) d'après le découpage en blocs précédent. Comme nous n'implémenterons pas l'algorithme F5, je ne donnerai pas l'équation de cette transformation que vous pourrez trouver facilement **[9]**.

Lors de la décompression de l'image, c'est la transformation inverse, nommée IDCT, qui sera appliquée.

3.1.5 Matrice de quantification

Lors de cette étape on va utiliser un facteur de qualité (généralement compris entre 0 et 100) qui va permettre de « simplifier » le signal en altérant plus ou moins (en fonction du facteur de qualité) les coefficients correspondants aux hautes fréquences (qui seront transformés en 0). Il s'agit en fait de prendre une matrice de quantification et de diviser terme à terme les valeurs de la matrice issue de la DCT par les valeurs de celle-ci).

Le choix des matrices de quantification est normalement laissé à l'appréciation du développeur. Toutefois, ce sont souvent les mêmes matrices (une pour la luminance et une pour la chrominance) qui sont utilisées car déjà testées et fournissant de bons résultats.

3.1.6 Application du codage RLE et Huffman

On applique enfin les algorithmes de compression RLE et Huffman pour réduire encore la taille de l'image **[9]**.

3.2 Matrix embedding

F5 utilise également un algorithme de stéganographie connu sous le nom de *matrix embedding* et introduit par Crandall en 1998 **[10]**. Cette méthode détourne l'utilisation des codes détecteurs et correcteurs d'erreur dans le but d'insérer un message dans une image mais en minimisant les modifications des coefficients de l'image.

De manière très simplifiée, dans la théorie des codes correcteurs, l'émetteur envoie un message **msg** en le transformant grâce à une matrice génératrice **G**. Le récepteur reçoit alors ce « mot de code » qui a pu être parasité pendant le transit. Grâce à une matrice de contrôle de parité il peut alors retrouver **msg**. C'est cette technique qui a été détournée.

3.3 L'algorithme F5 proprement dit

Soit **img** l'image utilisée pour y enfouir le message **msg** et un mot de passe **pwd**, l'algorithme F5 tel que décrit dans **[6]** est le suivant :

```
Démarrer la compression jpeg de img et arrêter juste après la quantification
Initialiser un générateur de nombres aléatoires avec pwd
Déterminer le paramètre k à partir de la capacité de l'image et la longueur du message secret n = 2k - 1
Enfouir le message secret msg à l'aide de la technique du matrix embedding
Terminer la compression jpeg
```

En elle-même la description de l'algorithme est très courte... mais s'il fallait en donner une version complète intégrant toutes les étapes de la compression jpeg et le *matrix embedding*, cela serait bien plus laborieux et par la suite long à implémenter... Voilà en tout cas les bases qui vous permettront de créer votre propre algorithme de stéganographie.

Conclusion

Nous avons pu voir deux exemples d'algorithmes de stéganographie mais il en existe bien d'autres ! De même, nous avons cherché seulement à dissimulé du texte dans des images mais il est bien sûr possible d'inclure des fichiers binaires dans d'autres types de fichiers tels que des fichiers son ou vidéo. Et même, pour être plus original, pourquoi pas dans des partitions de musique ou dans des en-têtes http ? Il ne vous reste plus qu'à créer votre propre algorithme... mais n'oubliez pas que plus l'information à enfouir est importante, plus il y a de risques pour qu'elle soit décelée ! Imaginez notre esclave Grec avec un message écrit sur tout le corps. Même avec des cheveux très longs pour le recouvrir il aurait paru tout de suite suspect... ■

Références

[1] Electronic Frontier Foundation, *« Is Your Printer Spying On You? »*: https://w2.eff.org/Privacy/printers/index.php

[2] Image originale de Lenna au format tiff : https://www.cs.cmu.edu/~chuck/lennapg/lena_std.tif

[3] Photo complète de Lena Sjööblom : http://www.lenna.org/full/len_full.html

[4] D. Munson, *« A note on Lena »*, IEEE transactions on image processing, vol. 5, n. 1, janvier 1996 : https://www.cs.cmu.edu/~chuck/lennapg/editor.html

[5] P. Bienaimé, *« Le challenge du logo ANSSI »*, MISC n°73, mai 2014 : http://connect.ed-diamond.com/MISC/MISC-073/Le-challenge-du-logo-ANSSI

[6] A. Westfeld, *« F5 - A Steganographic Algorithm. Higher Capacity Despite Better Steganalysis »*, Proceedings of the 4th International Workshop on Information Hiding, 2001, p. 289 à 302 : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.3651&rep=rep1&type=pdf

[7] Spécifications du format jpeg par l'ITU (*International Telecommunication Union*) : http://www.w3.org/Graphics/JPEG/jfif3.pdf

[8] Équation de la DCT : http://fr.wikipedia.org/wiki/JPEG#Transform.C3.A9e_DCT

[9] T. Colombo, *« La compression dans tous ses états »*, GNU/Linux Magazine n°181, avril 2015, p. 26 à 35.

[10] R. Crandall, *« Some notes on steganography »*, 1998 : http://dde.binghamton.edu/download/Crandall_matrix.pdf