DEPARTEMENT INFORMATIQUE
92 Bd Niels Bohr,
69100 Villeurbanne

TP2 - Stéganographie

R5.06 - SENSIBILISATION A LA PROGRAMMATION MULTIMEDIA

La stéganographie c'est l'art de cacher des images dans d'autres images.

I. MATERIEL TECHNIQUE

Vous devrez utiliser *python* et *PIL*. Vous pouvez utiliser un éditeur de texte pour éditer votre code, que vous sauvegarderez au format *.py*. Attention à l'indentation, en python c'est 4 espaces. Pour lancer votre code, ouvrez un terminal et taper *python chemin_vers_votre_code.py*.

II. FICHIERS UTILES

Fichier sources à compléter :

- encode.py, decode.py, stegano.py

Images utiles :

- mask.jpg, message.jpg, mask_lowres.jpg, message_lowres.jpg

Exercices à décoder :

- message_x.png, message_x_cool.png

III. ANATOMIE D'UNE IMAGE

Une image est un tableau de pixel. Chaque pixel est composé de trois canaux contenant le rouge, le vert et le bleu, chacun stocké dans un octet, soit 256 valeurs par canaux. Ainsi une image 1024x768 fait en fait 1024x768x3 octets.

IV. UN PEU DE BINAIRE

Un octet est composé de 8 bits. Chaque valeur entre 0 et 256 est ainsi représentée par une variation de ces 8 bits.

Adrien Peytavie – adrien.peytavie@univ-lyon1.fr

Note : La fonction **bin** affiche la visualisation binaire d'un nombre.

```
>>> bin(0)
'0b0'

>>> bin(10)
'0b1010'

>>> bin(200)
'0b11001000'

>>> bin(255)
'0b11111111'
```

Le code joint, dans le fichier *stegano.py* vous fournit les fonctions *high* et *low* permettant d'extraire les bits de poids fort et ceux de poids faible.

L'opération de *shift* décale les bits d'un côté ou de l'autre :

```
>>> bin(0b11 << 2)
'0b1100'

>>> bin(0b11 << 3)
'0b11000'
```

V. STEGANOGRAPHIE

On cache généralement les bits de poids fort d'une image dans les bits de poids faible d'une autre. Par exemple, pour une valeur 230, on aimerait cacher la valeur 257 dedans.

```
>>> bin(230)
'0b11100110'

>>> bin(210)
'0b11010010'
```

En prenant les quatre bits de poids fort de 230 (i.e. 1110) et les quatre bits de poids fort de 210 (i.e. 1101), on peut les coller ensemble pour créer un nouveau nombre.

```
>>> 0x11101101
237

>>> 0b11010000
208
```

On perd ainsi la partie faible du premier nombre (230) qui est un peu modifié (ici, on obtient 237). Le nombre caché quant à lui perd aussi sa partie faible et est aussi un peu modifié (208). Il y a perte d'information. Cette perte va se caractériser par des images légèrement dégradées.

VI. PARTIE 1 - DECODAGE SIMPLE (15 MINUTES)

Le vilain docteur X a transmis une image *message_x.png* à ses sbires. Vos experts vous annoncent aussi qu'ils pensent que les bits de poids fort de l'image cachée sont dissimulés dans les 4 bits de poids faible de l'image récupérée.

En vous servant de la base de code dans *decode.py*, écrivez un décodeur capable de révéler la vérité.

Dans ce code, *image* est un tableau contenant toutes les valeurs R, G et B d'une image. Vous devez parcourir ces valeurs, extraire les 4 bits de poids faible et les stocker dans le tableau *result*.

Note : Attention, si votre image apparaît noire, vous avez sans doute oublié de transformer les bits de poids faible en bit de poids fort.

Note : Vous pouvez procéder par étape. En premier lieu, copiez tous les pixels de l'image d'entrée dans l'image de sortie. Puis ne récupérez que les 4 bits de poids faible, le résultat devrait apparaître, mais très sombre. Puis ensuite transformez les bits de poids faible en poids fort.

VII. PARTIE 2 - ENCODEUR SIMPLE (15 MINUTES)

Pour pouvoir piéger docteur X, vous voulez lui envoyer des faux messages par le même moyen.

En utilisant la base de code dans **encode.py**, écrivez un encodeur simple capable de cacher les 4 bits de poids fort d'une image dans les quatre bits de poids faible d'une autre.

Note : Utilisez de petites images au début, c'est lent... Si vous n'avez pas d'image qui conviennent et de même taille, utilisez message.jpg et mask.jpg fournis.

Note : Servez-vous de votre décodeur pour vérifier votre travail.

VIII. PARTIE 3 - ENCODEUR DECODEUR VARIABLE (1 MINUTE)

Votre encodeur / décodeur fonctionne seulement sur 4 bits. Faites un encodeur / décodeur variable acceptant en argument le nombre de bit à utiliser.

Par exemple, votre encodeur peut accepter un argument n disant le nombre de bits à utiliser :

python encodeur.py 2

Dans ce cas-là, 2 bits de poids fort du message seront stockés dans les 2 bits de poids faible de l'image résultat, les 6 autres bits étant les bits de poids forts de l'image servant à cacher.

Discutez ensemble de vos résultats.

Note : En python, **sys.argv** vous permet d'accéder aux arguments de votre programme. Dans l'exemple précédant, *sys.argv* vaut ['encodeur.py', '2'], donc sys.argv[1] vaut '2'. Notez que c'est une chaîne de caractères à convertir en entier avec la fonction **int**.

Adrien Peytavie – adrien.peytavie@univ-lyon1.fr

Vous aurez besoin du module *sys* disponible grâce à *import sys* à mettre en début de programme.

IX. PARTIE 4 - ENCODEUR / DECODEUR PARAMETRABLE (1 MINUTE)

Utilisez *sys.argv* pour pouvoir passer le nom des fichiers en argument. Par exemple :

```
python encode.py 3 mask.jpg message.jpg res_enc.png

python decode.py 3 res_enc.png res_dec.png
```

va encoder le message contenu dans *message.png* grâce au masque

mask.png en utilisant 3 bis et va stocker le résultat dans

res_enc.png. Puis le décodeur va utiliser *res_enc.png* et stocker

le décodage dans *res_dec.png* en utilisant toujours 3 bits.

Note : Essayez ensuite de stocker le résultat de l'encodage dans un fichier .jpg au lieu de .png. Discutez.

IX.1 PARTIE 5 - ENCODEUR A MELANGE (5 MINUTES)

Un des problèmes de cette méthode est que l'image cachée est plutôt bien visible si on augmente le nombre de bits alloués au message.

Nous proposons de mélanger les pixels aléatoirement lors de l'encodage. Le module *random* contient une méthode *shuffle* permettant de mélanger aléatoirement les éléments d'une liste.

```
>>> l = [1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>>                                                                    l
[7, 6, 5, 4, 2, 1, 3]
```

Avant d'encoder votre image, mélangez les pixels du message. Observez que l'on n'est plus capable de voir l'image cachée, même en allant jusqu'à 7 bits.

Note : On pourra ajouter un paramètre en ligne de commande à l'encodeur pour dire si oui ou non il faut mélanger les pixels.

X. PARTIE 6 - DECODEUR A MELANGE

Bon, c'est bien tout cela, mais quand on décode cela ressemble à de la neige. Comment retrouver le bon ordre des pixels ?

On va utiliser une propriété intéressante des nombres aléatoires, ceux-ci sont prévisibles et on peut retrouver la même séquence si le générateur de nombre aléatoire est initialisé de façon identique.

```
>>> import random

>>> # Melange d'une liste
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>> l
[6, 7, 4, 2, 5, 0, 1, 3]

>>> # Nouveau mélange
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>> l
[5, 3, 7, 1, 4, 0, 6, 2]

>>> # Résultat différents !!
>>> # Initialisation du générateur
>>> random.seed(25)

>>> # Mélange d'une liste
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>> l
[5, 4, 3, 2, 1, 0, 7, 6]

>>> # Initialisation identique
>>> random.seed(25)

>>> # Nouveau mélange
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>> l
[5, 4, 3, 2, 1, 0, 7, 6]

>>> # Résultat similaires
```

Ainsi, en vous mettant d'accord avec le porteur du message sur une *seed* d'initialisation du générateur aléatoire, vous pouvez savoir comment celui-ci a mélangé son image.

Première étape, modifiez votre encodeur pour pouvoir changer la *seed* avant le mélange.

Deuxième étape, avant de décoder ou après avoir décodé, il faut mélanger de nouveau le résultat dans le décodeur.

Cependant il faut mélanger le résultat dans l'autre sens.

Si on observe le mélange suivant :

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(l)

>>> l
[5, 4, 3, 2, 1, 0, 7, 6]
```

On observe que l'élément qui était avant en position [0] de l'ancienne liste est maintenant dans la nouvelle liste en position 5.

Ainsi l'algorithme est le suivant :

```
image_demelangée[i] = image_melangée[l.index(i)]
```

Avec *l* une liste de correspondances comme dans l'exemple précédant et *index* la fonction python permettant de chercher un élément dans une liste.

Vous devez donc modifier votre décodeur pour qu'il prenne une *seed* et effectue le *démêlage*. Il est à votre charge de créer la liste de correspondances *l* (mais je suis trop gentil, le code est déjà donné).

Cependant cette solution est très lente et je vous encourage à la tester avec les images basse résolution *message_lowres.jpg* et *mask_lowres.jpg*.

Pourquoi cette solution est-elle lente ? Quelle est sa complexité ?

XI. PARTIE 7 - DECODEUR A MELANGE EFFICACE

Proposez une solution plus rapide pour décoder l'image *message_x_2.png*, qui utilise 6 bits pour stocker son message et utilise la seed 20.