Stéganographie : Bitmap et GIF

Foud Hind et Patti Philippe

Projet de stéganographie sur fichier image Bitmap et GIF
réalisé pour le cours de SYSG5

Table des matières

Introduction

L'objectif du projet est d'implémenter un programme illustrant le concept de sécurité suivant : La stéganographie.
Abordée du point de vue du cours de système, elle nous permet d'approfondir nos connaissances sur l'utilisation de la mémoire pour stocker des informations.
Il est évident que pour mener à bien un projet de cette envergure, d'autres compétences nous ont été nécessaires telles que l'apprentissage du langage C à un niveau plus basique notamment pour réaliser des opérations de manipulation de bits. Nous avons également eu besoin d'apprendre en détails les structures de fichier bitmap et GIF pour décider de la meilleure technique à utiliser afin d'y cacher des messages.
Dans notre implémentation, pour des raisons de clareté, nous avons choisi de suivre une structure de dossier 'bmp' et 'gif' regroupant le code qui permettra de cacher un message à l'intérieur de ces deux formats de fichiers respectifs. Dans ce rapport, nous passerons en revue l'organisation du projet, le choix pour chaque format, la technique utilisée et l'évolution du projet.
Nous fournirons aussi un guide pour utiliser les exécutables.

Organisation du projet

La structure de notre projet s'organise essentiellement en quatre répertoires dont voici le détail :

- **dist :** contient les exécutables du programme dans un dossier bmp et gif respectifs.
- **rapport :** contient le rapport en format tex et pdf, les fichiers qui serviront à construire celui-ci ainsi que le makefile qui le lancera. On pourra également y trouver le document pdf de la présentation.
- **rsc :** contient les ressources qui servent d'input au programme [bitmap et gif] ainsi que les outputs qui seront produits dans un dossier bmp et gif respectifs.
- **src :** contient les sources du programme dans un dossier bmp et gif respectifs

On peut également trouver à la racine, un makefile général qui automatise le lancement du programme ainsi qu'un fichier README.

Stéganographie ?

La stéganographie est l'art de la dissimulation : son objet est de faire passer inaperçu des données dans d'autres données. Elle se distingue de la cryptographie, « art du secret », qui cherche à rendre un message inintelligible à autre que qui-de-droit.
Les fichiers peuvent être de type divers : fichiers image, audio, html, vidéos etc.
Parmi les scénarios de tests envisagés, nous avons tenté de cacher :

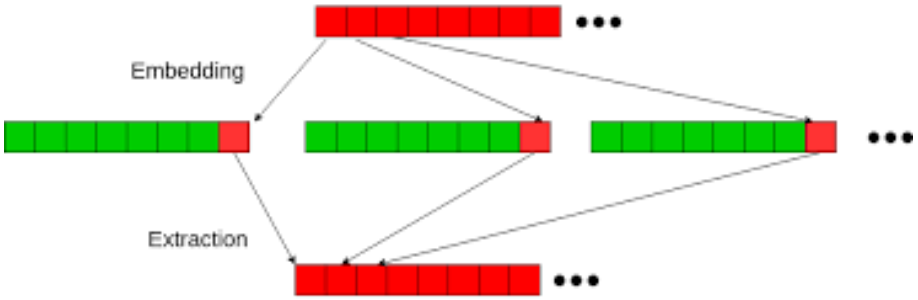
- ◇ un **texte** (.txt, .pdf) dans une **image** (.bmp/.gif)
- ◇ une **image** (.bmp, .gif) dans une **image** (.bmp/.gif)
- ◇ une **vidéo** (.mp4) dans une **image** (.bmp/.gif)

Il est évident que pour que cela soit possible, des conditions sont à poser. Il faudra bien sûr que le fichier à cacher soit plus petit que le fichier objet. Il faudra aussi peauffiner les conditions de faisabilité qui sont fortement dépendantes du type de fichier et veiller à respecter la règle d'or : le dissimuler à tout prix. Cela signifie qu'il faudra limiter au maximum l'impact sur le fichier dans lequel on aura caché des données sinon ce serait contraire au principe de stéganographie.

Technique utilisée

Pour coder/décoder un message, nous employons la technique du Least Significant Bit (LSB). Comme vous le voyez sur le schéma ci-dessous [Schéma général], cette technique consiste à se focaliser sur le bit le moins important d'un byte appelé le bit de poids faible. Pour coder le message, il suffit de lire un byte à cacher et de l'encoder dans le bit de poids faible d'un byte choisi. Pour décoder le message, il suffit de lire le bit de poids faible de ces bytes 'élus' et de reformer un message compréhensible.
C'est cette logique que nous nous sommes employés à suivre dans le cas de traitement sur ces deux formats de fichiers image. Ainsi, la technique est la même mais certains choix se sont imposés de part les différences entre les deux structures de ces fichiers image. Ces choix portent notamment sur 'l'élection' des bytes dans lesquels nous allons cacher l'information. En effet, pour que le message ait le plus petit impact possible sur le fichier image, il faut employer des bytes qui peuvent perdre un bit d'information sans que cela ne se remarque. Par exemple, pour une couleur codée en rgb, le changement du dernier bit a un impact très faible, quasi invisible à l'oeil nu. Il est donc adéquat pour y dissimuler des données.

Schéma général



Encodage : extrait de code source concernant la manipulation des LSB

```
1  void hide_bit(FILE *src_img, FILE *dest, const int secret_bit)
2  {
3
4      char src_img_buffer = fgetc(src_img);
5
6      int img_bit = src_img_buffer & 1; // donne val du lsb
7
8      if (img_bit != secret_bit)
9      {
10         if (secret_bit == 0)
11             src_img_buffer = src_img_buffer & ~1; // met le premier bit a 0
12         else
13             src_img_buffer = src_img_buffer | 1; // met le dernier a 1
14     }
15     fputc(src_img_buffer, dest);
16 }
```

../src/utils/bmp.c

Décodage : extrait de code source concernant la manipulation des LSB

```
1  #if LSB
2  int decode_bit(FILE *src_img)
3  {
4      return fgetc(src_img) & 1;
5  }
6  #else
7  int decode_bit(FILE *src_img)
8  {
9      return (fgetc(src_img) >> 7) & 1;
10 }
11 #endif
```

../src/utils/bmp.c

Guide pratique

```
1  #NOM : Makefile
2  #CLASSE : SYSG5
3  #OBJET : Steganographie avec Bitmap et GIF
4  #AUTEUR : Foud Hind et Patti Philippe
5  #HOWTO : make ; make run_gif; make build_gif; make clean
6
7  # make              : execute les demos
8  # make run          : execute les demos
9  # make build        : compile les demos
10 # make clean        : supprime les fichiers generes
11
12 # make run_bmp      : execute uniquement la demo pour le format bmp
13 # make build_bmp    : compile uniquement la demo pour le format bmp
14 # make clean_bmp    : supprime uniquement les fichiers generes lies au format bmp
15
16 # make run_gif      : execute uniquement la demo pour le format GIF
17 # make build_gif    : compile uniquement la demo pour le format GIF
18 # make clean_gif    : supprime uniquement les fichiers generes lies au format GIF
```

../makefile

Bitmap

Pourquoi le bitmap ?

Le format BITMAP aussi appelé DIB (Device Independent Bitmap) a été conçu par Windows corporation pour pouvoir échanger des images entre devices sans avoir à se soucier de la logique de ceux-ci. Ces images ont des extensions .bmp ou encore .dib. Techniquement, une image bitmap est un format de fichier non compressé, cela signifie entre autre que chaque pixel possède sa représentation sous forme d'un bit ou d'une série de bits. On peut donc opposer sa structure à celle d'une image PNG, JPEG ou encore GIF qui utilise la compression pour regrouper des pixels similaires afin de réduire la taille globale du fichier. On les appelle donc bitmaps car ils ne sont pas compressés (possible de le faire cependant). Ils n'ont donc aucune perte de données et sont donc généralement plus lourds.
La structure du format BITMAP est connue et disponible en ligne. En voici un schéma :

Bitmap Data Example using a 2x2 Pixel, 24-Bit Bitmap

Offset	Size	Hex Value	Value	Description
0	2	`42 4D`	"BM"	Magic Number (unsigned integer 66, 77)
2	4	`46 00 00 00`	70 Bytes	Size of Bitmap
6	2	`00 00`	Unused	Application Specific
8	2	`00 00`	Unused	Application Specific
10	4	`36 00 00 00`	54 bytes	The offset where the bitmap data (pixels) can be found.
14	4	`28 00 00 00`	40 bytes	The number of bytes in the header (from this point).
18	4	`02 00 00 00`	2 pixels	The width of the bitmap in pixels
22	4	`02 00 00 00`	2 pixels	The height of the bitmap in pixels
26	2	`01 00`	1 plane	Number of color planes being used.
28	2	`18 00`	24 bits	The number of bits/pixel.
30	4	`00 00 00 00`	0	BI_RGB, No compression used
34	4	`10 00 00 00`	16 bytes	The size of the raw BMP data (after this header)
38	4	`13 0B 00 00`	2,835 pixels/meter	The horizontal resolution of the image
42	4	`13 0B 00 00`	2,835 pixels/meter	The vertical resolution of the image
46	4	`00 00 00 00`	0 colors	Number of colors in the pallet
50	4	`00 00 00 00`	0 important colors	Means all colors are important
Start of Bitmap Data				
54	3	`00 00 FF`	16,711,680	Red, Pixel (0,1)
57	3	`FF FF FF`	16,777,215	White, Pixel (1,1)
60	2	`00 00`	0	Padding for 8 bytes/row (Could be a value other than zero)
62	3	`FF 00 00`	255	Blue, Pixel (0,0)
65	3	`00 FF 00`	65,280	Green, Pixel (1,0)
68	2	`00 00`	0	Padding for 8 bytes/row (Could be a value other than zero)

Application du LSB

Nous avons choisi de cacher des informations à partir de la fin du header c'est-à-dire après le cinquante-quatrième byte du fichier. Cela nous permet d'amoindrir les altérations de l'image en sortie car les modifications sont invisbles à l'oeil humain. De plus, comme dit plus haut, ce fichier n'utilisant généralement jamais d'algorithme de compression, il ne nous sera pas nécessaire de nous soucier de perte de données qui impacteraient le décodage d'une image dans laquelle on aurait caché des données.

(a) Bitmap source

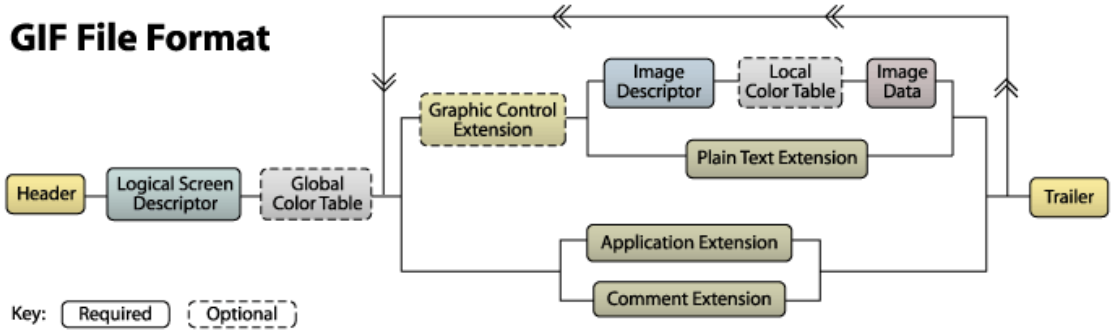
(b) Bitmap dest

FIGURE 1 – Encodage d'un fichier texte dans un bitmap

GIF

Pourquoi le GIF ?

Le format GIF permet de stocker plusieurs images dans un fichier. Ceci permet de créer des diaporamas, voire des animations si les images sont affichées à un rythme suffisamment soutenu. Chaque image d'une animation peut avoir sa propre palette. Le GIF est un format très utilisé, particulièrement sur les réseaux sociaux. Ce projet peut donc intéressé pas mal de gens.
La structure du format GIF est connue et disponible en ligne. En voici un schéma :



Application du LSB

Nous avons choisi de cacher des informations dans les couleurs qui se trouvent dans les Local Color Table. Cette section facultative peut revenir devant chaque bloc image data. Si cette section n'existe pas, nous la rajoutons.

Il est possible de calculer la taille maximale du message cachée à l'avance : en comptant le nombre de bloc image data et en le multipliant par le nombre de byte dans la Global Color Table.

Avancement

Lecture

Le programme peut déja lire un gif en entier, section par section. Les tailles des sections sont indiquées à des endroits différents par sections. La lecture se fait donc différemment en fonction de la section.

De plus, on peut déja savoir le nombre maximal de Local Color Table pour le fichier modifié.

Lecture

Le programme réécrit le gif dans un nouveau fichier, en insérant des Local Color Table, identique à la Global Color Table. Ceci permettra d'y insérer un message. L'insertion du message devrait être évidente, vue qu'elle sera presque identique à la stéganographie dans un fichier bitmap.

Il y a toutefois encore un bug dans la réécriture du fichier gif, probablement à l'écriture des nouvelles Local Color Table.