# 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 approfondi du langage C, 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 ce rapport, nous passerons en revue l'organisation du projet, le choix pour chaque format et la technique utilisée. Nous fournirons aussi un guide pratique 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.

# 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
```

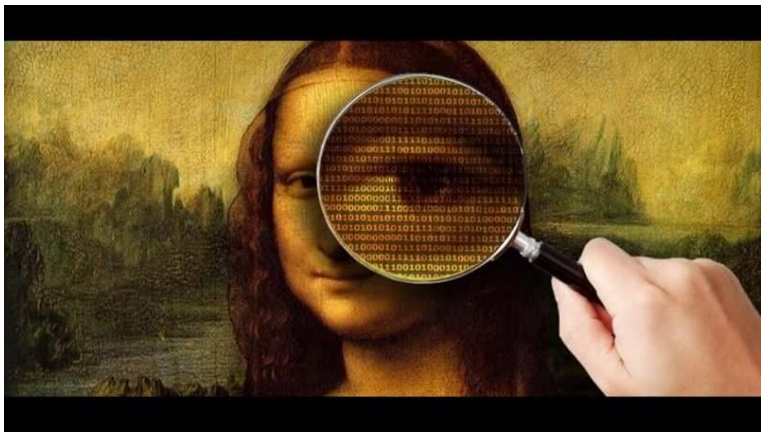
../makefile

# Stéganographie : Définition

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 :

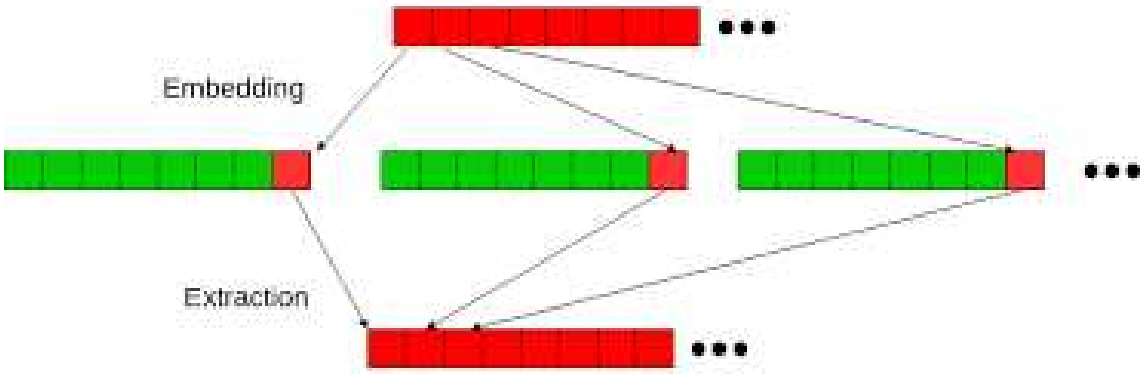
- ◇ un **texte** (.txt, .pdf) dans une **image** (.bmp/.gif)
- ◇ une **image** (.bmp, .gif) dans une **image** (.bmp/.gif)
- ◇ une **vidéo** (.mp4) dans une **image** (.bmp/.gif)

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.



# Least Significant Bit (L.S.B.)

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.

## 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 dernier bit a 0
12         else
13             src_img_buffer = src_img_buffer | 1; // met le dernier bit 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  void decode_secret_bmp(FILE *src_img, FILE *dest, const unsigned length)
2  {
3      char dest_buffer;
4      for (unsigned i = 0; i < length; i++)
5      {
6          dest_buffer = 0;
7          for (int j = 0; j < 8; j++)
8          {
9              dest_buffer <<= 1;
10             int bit = decode_bit(src_img);
11             if (bit == 0)
12                 dest_buffer = dest_buffer & ~1;
13             else
14                 dest_buffer = dest_buffer | 1;
15         }
16         fputc(dest_buffer, dest);
17     }
18 }
```

../src/utils/bmp.c

```
1  int decode_bit(FILE *src_img)
2  {
3      return fgetc(src_img) & 1;
4  }
```

../src/utils/bmp.c

# Bitmap

## Aperçu du 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.
Une image bitmap est un format de fichier généralement non compressé. Cela signifie que chaque pixel possède sa représentation sous forme d'une série de bits. Les bitmaps n'ont aucune perte de données et sont donc généralement plus lourds. On peut opposer leur structure à celle d'une image PNG, JPEG ou encore GIF, qui utilisent la compression pour regrouper des pixels similaires.
Les headers des bitmaps peuvent employer plusieurs versions. Ces versions donnent plus ou moins de métadonnées sur le bitmap. Le BITMAPINFOHEADER est une des versions les plus simples et rétrocompatible. Actuellement, il existe des versions plus complètes, tels que le BITMAPV4HEADER ou BITMAPV5HEADER.
La structure du format BITMAP est connue et disponible en ligne. En voici un schéma, utilisant le BITMAPINFOHEADER :

**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) |

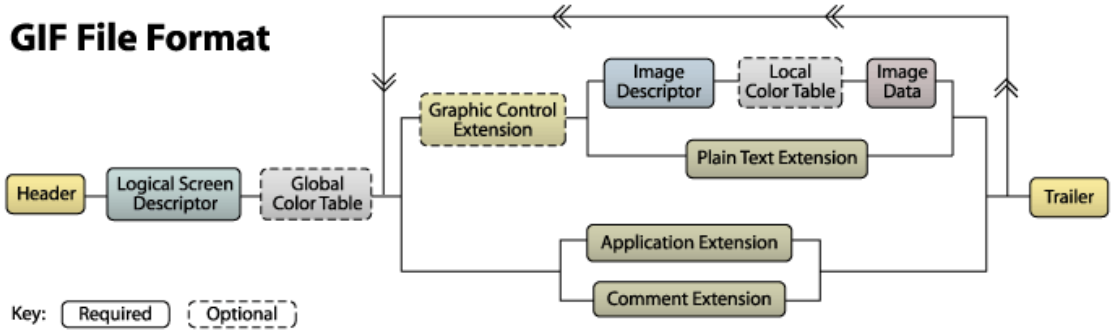
## Application du LSB

Nous cachons les données dans les bytes décrivant les couleurs de l'image source, juste après le header. Les modifications étant faites sur des couleurs, l'image est peu altérée. Les modifications sont invisibles à l'oeil humain. De plus, ce fichier n'utilisant généralement pas d'algorithme de compression, on ne doit pas se soucier de perte de données, ce qui aurait probablement impacter le décodage d'une image dans laquelle on aurait caché des données.

# GIF

## Aperçu du 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 bytes des Local Color Table. Les Color Table décrivent des couleurs, chaque couleur étant codée sur 3 bytes. Ces informations ne sont pas compressées contrairement au bloc image data.

Cette section, Local Color Table, est facultative et peut revenir devant chaque bloc image data. Si elle n'existe pas, nous la rajoutons en copiant alors la global color table. Au final, on aura autant de local color table qu'il y a de blocs image data.

Il est possible de calculer la taille maximale du message cachée à l'avance. Pour cela, on compte le nombre de bytes que l'on peut cacher par Color Table : nombre de bytes dans une color Table / 8. Puis, on multiplie cela par le nombre potentiel de Local Color Table dans le fichier, ce qui revient à compter le nombre de bloc image data.

Nous avons décidé de cacher la taille du message dans la première Local Color Table. Le gif source doit donc contenir au moins 2 LCT pour pouvoir cacher un message.

La taille du message est codée sur un unsigned, donc 4 bytes cachés sur 32 bytes d'une Local Color Table. Par soucis de simplification, les color tables doivent contenir au moins 32 bytes d'informations. Ceci peut se vérifier facilement : le gif doit employer au moins 16 couleurs différentes. Il n'y a que quelques rares exceptions qui ne correspondront pas à ce critère.

# Présentation

## Préambule

Dans le cadre de l'évaluation du projet, une défense de celui-ci a eu lieu sur base d'un document pdf fourni avec le projet, suivi d'une démonstration. Ainsi, pour rendre cette dernière plus consistante, il nous a été proposé d'implémenter une autre technique relativement comparable à celle du LSB, appelée MSB, acronyme de Most Significant Bit.

## Most Significant Bit (M.S.B.)

Pour coder/décoder un message, nous employons ici la technique du Most Significant Bit (MSB). Cette technique consiste à se focaliser sur le bit le plus important d'un byte appelé le bit de poids fort. Pour coder le message, il suffit de lire un byte à cacher et de l'encoder dans le bit de poids fort d'un byte choisi. Pour décoder le message, il suffit de lire le bit de poids formats de ces bytes 'élus' et de reformer un message compréhensible.

## Encodage : extrait de code source concernant la manipulation des MSB

```
1  void hide_bit(FILE *src_img, FILE *dest, const int secret_bit)
2  {
3      char src_img_buffer = fgetc(src_img);
4
5      int img_bit = (src_img_buffer & 0x80) >> 7; // donne val du msb
6
7      if (img_bit != secret_bit)
8      {
9          if (secret_bit == 0)
10             src_img_buffer = src_img_buffer & 0x7F; // met le premier bit a 0
11         else
12             src_img_buffer = src_img_buffer | 0x80; // met le premier bit a 1
13     }
14     fputc(src_img_buffer, dest);
15 }
```

../src/utils/bmp.c

## Décodage : extrait de code source concernant la manipulation des MSB

```
1  void decode_secret_bmp(FILE *src_img, FILE *dest, const unsigned length)
2  {
3      char dest_buffer;
4      for (unsigned i = 0; i < length; i++)
5      {
6          dest_buffer = 0;
7          for (int j = 0; j < 8; j++)
8          {
9              dest_buffer <<= 1;
10             int bit = decode_bit(src_img);
11             if (bit == 0)
12                 dest_buffer = dest_buffer & ~1;
13             else
14                 dest_buffer = dest_buffer | 1;
15         }
16         fputc(dest_buffer, dest);
17     }
18 }
```

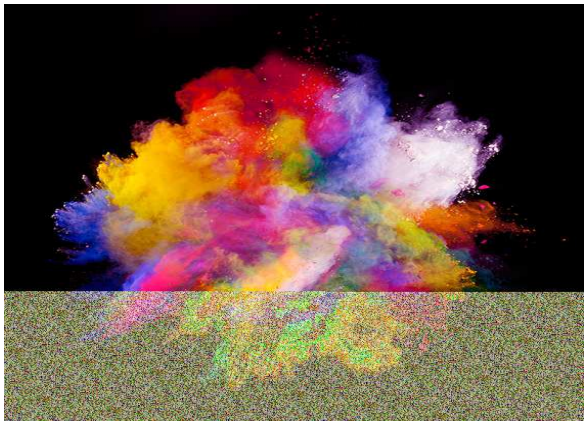
../src/utils/bmp.c

```
1  int decode_bit(FILE *src_img)
2  {
3      return (fgetc(src_img) >> 7) & 1;
4  }
```

../src/utils/bmp.c

## Résultats obtenus

(a) Bitmap source



(b) Bitmap dest

FIGURE 1 – Encodage d'une vidéo .mp4 dans un bitmap

FIGURE 2 – Altération visible du fichier d'origine

# Conclusion

Ce projet nous a permis de pratiquer de nombreuses compétences importantes pour les futurs informaticiens que nous sommes telles que le travail de recherche et la collaboration.

Nous avons dû appronfondir nos connaissances en programmation C pour répondre à des besoins de compréhension des formats de fichier mais aussi pour prendre les meilleures décisions concernant l'implémentation du code. Dans un premier temps, nous nous sommes efforcés d'afficher les fichiers afin d'en maitriser les différentes sections et contraintes qui y sont liées.

Nous avons ensuite commencé à coder selon une technique reconnue pour être simple et efficace qui se base sur le LSB évoquée plus haut. Il s'agit donc de travailler sur les bits de poids faible ce qui crée une différence imperceptible à l'oeil humain. D'abord dans le bitmap, pour des questions de simplicité, puis dans le gif.

Nous avons réutiliser le code le plus possible. Ce qui implique que le traitement est le même, la seule différence liée au format est l'endroit où l'on applique ce traitement. Nous travaillons après le header dans le cas des bitmaps alors que pour les gifs, nous traitons les local color table. Cela se justifie par le fait que celles-ci ne souffrent pas de compression et garantissent, en principe, un décodage sans encombres.

Pour terminer, nous avons également envisagé d'utiliser une autre technique appelée le MSB. Comme son nom l'indique, elle implique de travailler sur les bits de poids fort, ce qui trahit la manipulation du fichier. Ceci ne fait partie de notre implémentation que pour démontrer ce qu'il se passe lorqu'on ignore la bonne pratique.

# Références

https ://docs.microsoft.com/
https ://www.fileformat.info/
http ://www.matthewflickinger.com/lab/whatsinagif/