Rapport SYSG5 : Steganographie

Noé DELCROIX - G55990

November 2022

Contents

1 Introduction

Dans ce rapport, je vais vous présenter le résultat de mon travail de recherche au sujet de la stéganographie et plus précisément les techniques less significant bits (LSB) et zero-width characters.
La stéganographie est un ensemble de techniques permettant de cacher des données au sein d'un autre fichier. La stéganographie n'a rien de nouveau, peut s'appliquer sur plusieurs supports que ce soit des documents physiques (papier, pierre, photos, etc.) ou informatisé (images numériques, fichiers textes, etc.).
dans ce rapport, je vais présenter deux techniques. L'une permettant de stocker des données dans les pixels que compose une image PNG (ou tout autre format d'image n'utilisant pas de compression et représentant ces pixels en true color/RVB), l'autre permet de stocker n'importe quelle donnée dans un fichier texte ou tout autre format ne vérifiant pas l'intégrité de ses données.

2 LSB

La technique de stéganographie LSB, pour least significant bit, est une méthode pour cacher des données que ce soit du texte, une image ou tout autre fichier dans une image en utilisant le bit de poids faible des octets représentant chaque pixel.
Cette technique ne comporte pas beaucoup de limitations ci ce n'est d'avoir une donnée à cacher d'une taille maximale en bits de 3/8 du nombre de pixels composant l'image au format PNG-8 (j'explique à la section x.x quelles en sont les raisons). [2]

2.1 Format PNG

Pour illustrer cette technique de stéganographie, je vais utiliser le format d'image PNG avec une taille de canal de couleur sur 8 bits.

2.1.1 Qu'est-ce que le format PNG ?

Le format PNG a été créé dans le but de compenser les limitations du format GIF (format propriétaire) tout en le rendant beaucoup plus simple à implémenter. Il représente ses pixels sous forme matriciel avec 3 canaux de couleurs RVB.

2.1.2 Chunks

Le format PNG est composé d'un magic number et de multiples chunks. Un chunk est une structure permettant de décrire les différentes données d'un fichier PNG. Chaque chunk est composé de 4 parties :

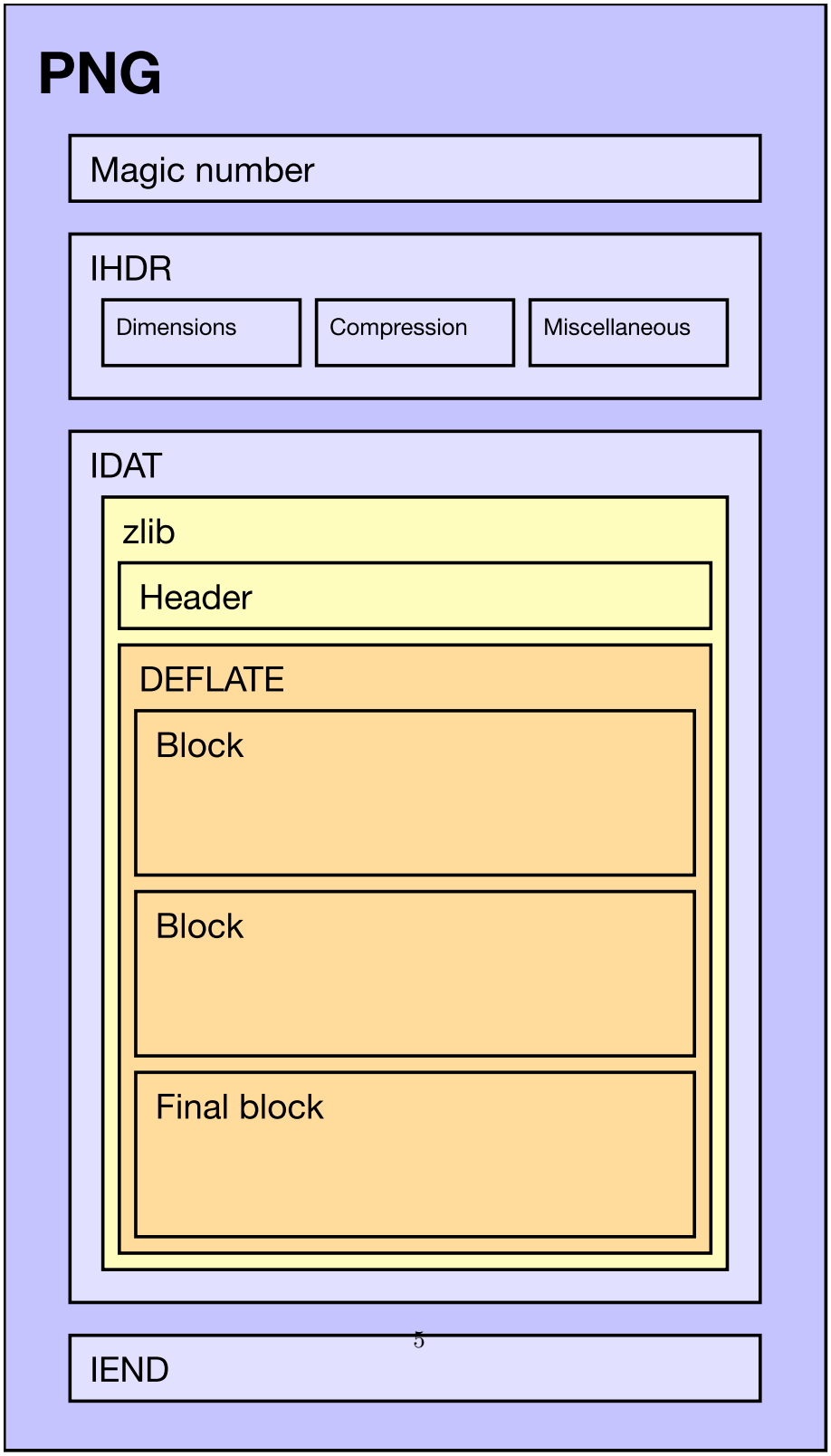
2.1.2.1 Longueur Sur 4 octets et peut être 0, spécifie la taille que prend la section données du chunk.

2.1.2.2 Type Sur 4 octets, spécifie le type du chunk (exemple : IDAT, IEND, etc.).

2.1.2.3 Données De la taille spécifiée dans la section longueur, contient les données de ce chunk.

2.1.2.4 CRC Checksum de l'intégrité de la section données.

2.2 Structure du format PNG



2.2.1 PNG file signature/magic number

0	89 50 4e 47 0d 0a 1a 0a	• Special: File signature • Length: 8 bytes	• "□PNG␍␊␚␊"

Suite de 8 octets permettant de reconnaitre le format de fichier PNG. Cette suite est celle-ci : 137 80 78 71 13 10 26 10

2.2.2 IHDR image header chunk

8	00 00 00 0d 49 48 44 52 00 00 01 2c 00 00 00 f0 08 02 00 00 00 35 94 ce c2	• Data length: 13 bytes • Type: IHDR • Name: Image header • Critical (0) • Public (0) • Reserved (0) • Unsafe to copy (0) • CRC-32: 3594CEC2	• Width: 300 pixels • Height: 240 pixels • Bit depth: 8 bits per channel • Color type: RGB (2) • Compression method: DEFLATE (0) • Filter method: Adaptive (0) • Interlace method: None (0)

Chunk indispensable et doit être en premier qui contient les méta données du fichier PNG tels que la largeur, la longueur, la taille d'un canal de couleur (8 bits en PNG-8), méthode de compression, etc.

2.2.3 IDAT chunk (Image data)

111	00 00 4f 85 49 44 41 54 78 da ed bd 59 94 5c c7 79 26 f8 fd 7f c4 bd b9 d5 0e 14 76 80 04 09 52 5c c0 45 a4 48 4a 94 65 89 6e 5b 94 45 db 92 77 f7 39 ea f6 d6 9e ee e9 33 0f d3 f3 d0 d3 0f 33 f3 3c 67 c6 ef ad ... 23 27 61 8e 1c 3b 8c 9c 84 39 72 ec 30 72 12 e6 c8 b1 c3 f8 ff 01 1b 21 8a 38 ee 4d 21 9d	• Data length: 20 357 bytes • Type: IDAT • Name: Image data • Critical (0) • Public (0) • Reserved (0) • Unsafe to copy (0) • CRC-32: EE4D219D	

Chunk contenant les pixels de l'image. Il peut y avoir plusieurs chunk IDAT et peut être compressé si un algorithme de compression est mentionné dans le chunk IHDR. C'est principalement ce chunk que nous allons utiliser.

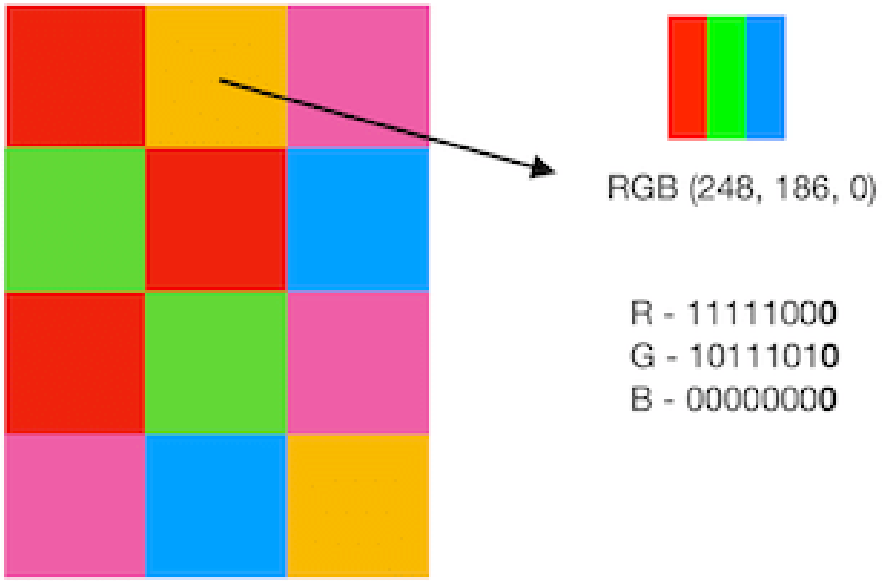
2.2.4 IEND chunk

20 578	00 00 00 00 49 45 4e 44 ae 42 60 82	• Data length: 0 bytes • Type: IEND • Name: Image trailer • Critical (0) • Public (0) • Reserved (0) • Unsafe to copy (0) • CRC-32: AE426082	

Chunk marquant la fin du fichier PNG. Il doit être placé à la fin et contient une section data vide.
D'autres chunks peuvent exister pour contenir d'autres informations facultatives

tels que la transparence, les méta données (date de création, modification, etc.), la pallette de couleurs, etc.

2.3 Structure du segment IDAT



Ici, je vais considérer que nous utillisons une image PNG sans compression où les pixels sont représentés par le type true color. Deux autres type existent (grayscale et indexed color) mais je ne m'attarderai pas dessus.
Dans la section données du segment IDAT, les pixels sont allignés l'un à la suite de l'autre. Chacun des pixels est composé de 3 canaux (rouge, vert et bleu) pour représenter la couleur de ce pixel. Chaque ligne de pixels est d'une longeur définie par le segment IHDR.
Un pixel d'une image PNG-8 a donc une taille de 3 octets et permet d'avoir 256^3 couleurs différentes.

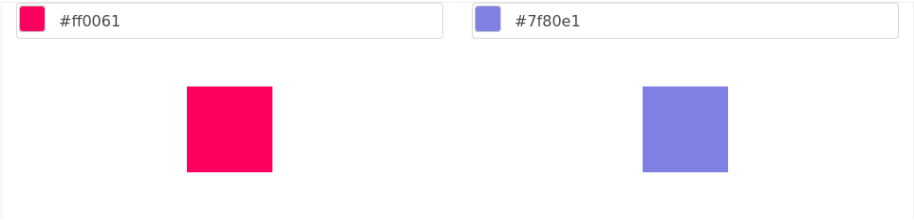
2.4 Compression

Les fichiers PNG utilisant très souvent une compression pour les octets du segment IDAT, j'utiliserai la bibliothèque libpng pour l'implémentation de mon script afin de ne pas devoir gérer cette partie compression/decompression avant lecture/écriture.
Le format PNG supporte par défaut l'algorithme de compression deflate/inflate mais est capable d'en utiliser d'autres tel que GZIP. [3] [5]

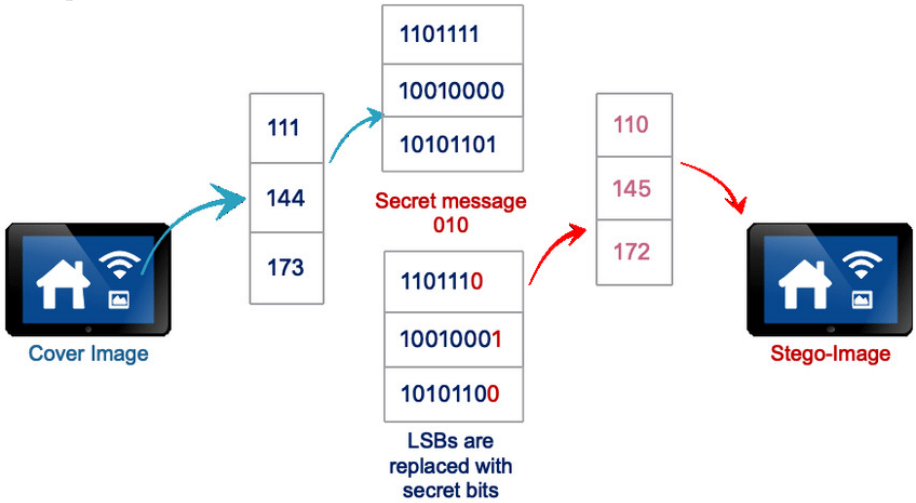
2.5 LSB sur PNG

L'idée de la stéganographie least significant bit est d'utiliser le bit de poids faible des canaux d'un pixel pour insérer des données.
Voici un exemple pour mieux comprendre :

Ici, nous avons 2 couleurs. Une couleur de référence (à gauche) et une couleur où j'ai inversé le bit de poids faible des 3 canaux de couleur.
La différence est (quasi) invisible à l'oeil nu et le fait qu'un pixel soit de taille très petite rend l'impossibilité de détecter la différence de couleur, en tant qu'humain.
Maintenant si l'on inverse les bits de poids fort, la couleur change totalement car ce bit à la possibilité d'ajouter ou de retirer 128 valeurs de couleur.



L'idée est donc de stocker des données binaires dans ces bits de poids faible.
Exemple :



3 LSB en pratique

Pour la partie pratique, on va donc d'abord charger les bits du texte à cacher dans un buffer et ensuite parcourir chacun des octets du segment IDAT du fichier PNG, reinitialiser le dernier bit de l'octet et le changer par le bit du buffer du fichier txt.
Pour cela, nous utiliserons les opération binaires et les masques binaires pour sélectionner et modifier les bits de l'image plus facilement. nous utiliserons ici principalement les opérateurs AND, OR et NOT.
J'ai dû me baser sur le script de BarbDev pour comprendre l'ordre et l'enchainement que devait avoir les fonctions de libpng [4] faute de documentation suffisamment complète à propos de la librairie libpng.

3.1 Encodage

Commençons par charger les bits du texte à cacher dans un buffer et créons le descripteur de fichier pour l'image :

```
FILE *fp = fopen(imgPath, "wb");
    img.txtBuff=(unsigned char *)malloc(sizeof(unsigned char)*strlen(text)*8);
    if(!fp){
        printf("Unable to open image. Check if the file exists and that it was correctly specified in the command.\n"
        return 1;
    }
    img.png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    img.info_ptr = png_create_info_struct(img.png_ptr);

unsigned size=img.height*img.width*3;
    if (png_get_color_type(img.png_ptr, img.info_ptr) != PNG_COLOR_TYPE_RGB){
        printf("The color type is not supported.\n");
        return 1;
    }

    if(strlen(text)*8>size){
        printf("Text is too long for this file.\n");
        return 1;
    }
    fillTxtBufferLsb(text);
```

Ensuite, chargeons l'IDAT du fichier PNG, contenant les octets représentant les bits au format RVB dans un buffer.

```
png_init_io(img.png_ptr, fp);

    png_set_IHDR(img.png_ptr, img.info_ptr, img.width, img.height, img.bit_depth, img.color_type, PNG_INTERLA
    png_write_info(img.png_ptr, img.info_ptr);
```

Modifions les derniers bits de chaque octets du buffer d'image par les bits du buffer du texte, réécrivons le buffer de l'image dans le fichier et fermons le descripteur de fichier.

```
writeData(text);
png_write_image(img.png_ptr, img.row_pointers);
png_write_end(img.png_ptr, NULL);
img.row_pointers = (png_bytep*) malloc(sizeof(png_bytep) * img.height);
    for (img.y=0; img.y < img.height; img.y++)
        img.row_pointers[img.y] = (png_byte*) malloc(png_get_rowbytes(img.png_ptr, img.info_ptr));

    png_read_image(img.png_ptr, img.row_pointers);
    fclose(fp);

for(int i=0; i<img.height && counter<strlen(text)*8; i++){
        for(int j=0; j<img.width && counter<strlen(text)*8; j++){
            img.row_pointers[i][j]=(img.row_pointers[i][j]&0xFE)+img.txtBuff[counter];
            counter++;
        }
    }
```

3.2 Utilisation du script pour l'encodage

Executer la section build du Makefile pour la compilation et l'édition des liens :

```
make build
> gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
```

Exécuter l'exécutable généré en lui passant le type de stéganographie (lsb), l'action (encode), le chemin du fichier qui va stocker la donnée et le texte qui va être caché.

```
./steganography lsb encode /home/noedelcroix/Documents/SYSG5/code/supports/example.png Hello world
> Finished encoding
```

3.3 Décodage

Pour le décodage, il y a moins d'étape que pour l'encodage. On peut imprimer directement les charactères dans le stdout.
Commençons par créer le descripteur de fichier pour l'image :

```
FILE *fp = fopen(imgPath, "rb");
    if(!fp){
        printf("Unable to open image. Check if the file exists and that it was correctly specified in the command.\n"
        return 1;
    }
    img.png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    img.info_ptr = png_create_info_struct(img.png_ptr);
```

On va alors boucler sur chacun des caractères de la partie IDAT de l'image PNG, lire le dernier bit de chaque octet, les grouper par octets et retourner l'octet sous forme de charactères au stdout.

```
unsigned char currentChar=0;
    unsigned counter=0;
    for(int i=0; i<img.height; i++){
        for(int j=0; j<img.width; j++){
            if(counter%8==0 && counter!=0){
                //printf("%d %d\n", i, j);
                if(currentChar<127)
                printf("%c", currentChar);
                currentChar=0;
            }
            currentChar=(currentChar<<1) | (img.row_pointers[i][j]&0x01);
            counter++;
        }
    }
```

3.4 Utilisation du script pour le décodage

Executer la section build du Makefile pour la compilation et l'édition des liens :

```
make build
> gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
```

Exécuter l'éxécutable généré en lui passant le type de stéganographie (lsb), l'action(decode) et le chemin du fichier qui contient la donnée cachée.

```
./steganography lsb decode /home/noedelcroix/Documents/SYSG5/code/supports/example.png Hello world
> Finished decoding
```

3.5 Résultat

Comme il est assez complexe de retrouver le segment IDAT dans un fichier PNG en C ou Bash sans librairies, nous allons simplement comparer la taille de l'image avant et après, pour montrer que l'ajout de données n'allourdit pas le fichier cachant la donnée.
Commençons par compiler et lire la taille du fichier avant encodage :

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ stat --printf="example.png size :
%s bytes\n" supports/example.png
example.png size : 303611 bytes
```

Commande encodage :

```
make build
./steganography lsb encode supports/example.png Hello World
```

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ make build
gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ ./steganography lsb encode supports/example.png "Hello World"
Encoding finished
noedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

On check la taille de l'image après encodage :

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ stat --printf="example.png size :
%s bytes\n" supports/example.png
example.png size : 302872 bytes
```

La donnée est bien enregistrée malgré que la taille de l'image n'est pas changée :

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ ./steganography lsb decode supports/example.png | head -c 11
Hello Worldnoedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

4 Cacher du texte dans un fichier texte

Je vais maintenant parler d'une autre technique de stéganographie. Cette technique de stéganographie permet de cacher des données au sein d'un fichier texte, au moyen de caractères dits zero-width. Ce sont des charactères faisant partie des wide characters qui ne font pas partie d'unicode mais de UTF-8.
Cette technique n'a pas de limitations au niveau de la taille à insérer mais augmente la taille du fichier initial (16 fois la taille de la donnée à insérer). [1]

4.1 Caractères zero-width

Les caractères zero-width sont des caractères unicodes qui ne prennent pas de place dans le texte. Ils sont donc invisibles au lecteur du fichier. Ils ne font pas partie d'unicode car ils sont codés sur 2 octets.

4.2 Fonctionnement

4.2.1 Encodage

Cette technique convertit la valeur binaire de la donnée à insérer en caractères zero-width. On utilise deux caractères zero-width au choix (pour la démo et l'exemple j'utiliserai les caractères zero-width space et zero-width non joiner). Un des deux caractères zero width va représenter la valeur 1 binaire et l'autre caractère va représenter la valeur 0 binaire.

Unicode Number	UTF-16 decimal code	Description
U+180E	6158	mongolian vowel separator
U+200B	8203	zero width space
U+200C	8204	zero width non-joiner
U+200D	8205	zero width joiner
U+200E	8206	left-to-right mark
U+200F	8207	right-to-left mark
U+FEFF	65279	zero width no-break

Voici un exemple avec les caractères zero width space (ZWSP) et zero width non-joiner (ZWNJ) :
donnée : 'h'
valeur binaire : 01101000
Si l'on considère le caractère ZWSP comme valant 0 et ZWNJ comme valant 1 :

0	ZWSP
1	ZWNJ
1	ZWNJ
0	ZWSP
1	ZWNJ
0	ZWSP
0	ZWSP
0	ZWSP

On ajoute donc les caractères ZWSP ZWNJ ZWNJ ZWSP ZWNJ ZWSP ZWSP ZWSP à la fin du fichier, ceux-ci représentant les bits de la données insérée et n'étant pas vu par l'humain.

4.2.2 Décodage

Pour le décodage, il suffit de lire les zero width characters présents dan sle fichier et les convertir un par un en bits que l'on regroupe en octets pour pouvoir les convertirs en unicode.

5 Cacher du texte dans un fichier texte en pratique

Pour la pratique, il faut retrouver les caractères zero-width pour pouvoir les convertir. Pour cela, il y a 2 manières de faire : Soit parcourir le fichier à la recherche du premier caractère zero width (ce que l'on va faire pour notre implémentation), soit stocker la taille de la donnée cachée en binaire sur les 8 derniers caractères zero-width du fichier.

5.1 Encodage

Les caractères zero-width n'étant pas supportés par le C, il faut inclure une librairie pour faire les conversions UTF-8 unicode. Pour cela, j'utilise la librairie wchar.h.
Il nous faut ensuite choisir 2 caractères zero-width pour représenter le 1 et 0 binaire :

```
wchar_t zwnj=0x200C; //zero-width non joiner represents 1
wchar_t zws=0x200B; // zero-width space represents 0
```

On crée notre file descriptor vers le fichier qui va stocker notre message caché :

```
textinfos.txtPtr=fopen(textinfos.path, "rw+");
    if(textinfos.txtPtr==NULL){
        printf("Unable to open file. Check if the file exists"
               " and that it was correctly specified in the command.");
        return 1;
    }
```

Ensuite, on va convertir le texte à encoder en binaire et stocker chacun des bits dans un buffer de unsigned char (plus petite valeur adressable avec des valeurs allant de 0 à 255).

```
int fillTxtBuffer(const char * txt){
    textinfos.txtBuff = (unsigned char *) malloc(sizeof(unsigned char) * strlen(txt)*8);
    for(unsigned i=0; i<strlen(txt); i++){
        char currentChar=txt[i];
        for(int j=7; j>=0; j--){
            textinfos.txtBuff[(i*8)+j]=currentChar%2;
            currentChar/=2;
        }
    }

    textinfos.size=strlen(txt)*8;

    return 0;
}
```

On place le curseur de notre pointeur de fichier à la fin du fichier.

```
fseek(textinfos.txtPtr, −1, SEEK_END);
```

On crée un fonction qui va convertir nos bits de notre buffer texte en caractères zero-width et les ajouter à notre fichier.

```
void addToFile(char bin){
    if(bin==1){
        fputwc(zwnj, textinfos.txtPtr);
    }else{
        fputwc(zws, textinfos.txtPtr);
    }
}
```

Et enfin, on va boucler sur chacun des bits du buffer texte et appeler la fonction qui va convertir les bit du buffer et écrire ces caractères zero-width.

```
for(unsigned i=0; i<textinfos.size; i++){
        addToFile(textinfos.txtBuff[i]);
    }
```

5.2 Utilisation du script pour l'encodage

Executer la section build du Makefile pour la compilation et l'édition des liens :

```
make build
> gcc −g −Wall −o steganography main.o zwnj−zws.o lsb.o −lm −lpng −lz
```

Exécuter l'éxécutable généré en lui passant le type de stéganographie (zero-width), l'action(encode), le chemin du fichier qui va stocker la donnée et le texte qui va être caché.

```
./steganography zero−width encode /home/noedelcroix/Documents/SYSG5/code/supports/test.txt Hello world
> Finished encoding
```

5.3 Décodage

Pour commencer, on crée de nouveau nos variables contenant nos zero-width characters.

```
wchar_t zwnj=0x200C; //zero−width non joiner represents 1
wchar_t zws=0x200B; // zero−width space represents 0
```

On crée un file descriptor en lecture vers le fichier stockant les données cachées.

```
textinfos.txtPtr=fopen(textinfos.path, "r");
```

On parcoure le fichier texte jusqu'à trouver le premier caractère zero-width.

```
wchar_t val;
    val=fgetwc(textinfos.txtPtr);
    while(val!=zwnj && val!=zws && !feof(textinfos.txtPtr)){
        val=fgetwc(textinfos.txtPtr);
    }
```

Et enfin, on lit les caractères zero-width du fichier, on les convertit en binaire (avec la fonction), on crée un char en groupant ces bits par 8 et on les retourne dans stdout.

```
unsigned counter=7;
    char currentChar=pow(2, counter)*convertZw(val);
    while(!feof(textinfos.txtPtr) && (val==zwnj || val==zws)){
        counter--;
        val=fgetwc(textinfos.txtPtr);
        currentChar+=pow(2, counter)*convertZw(val);
        if(counter==0){
            printf("%c", currentChar);
            counter=8;
            currentChar=0;
        }
    }
```

Fonction convertZw :

```
unsigned convertZw(const wchar_t val){
    return val==zwnj;
}
```

5.4 Utilisation du script pour le décodage

Executer la section build du Makefile pour la compilation et l'édition des liens :

```
make build
> gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
```

Exécuter l'éxécutable généré en lui passant le type de stéganographie (zero-width), l'action(decode) et le chemin du fichier qui contient la donnée cachée.

```
./steganography zero-width decode /home/noedelcroix/Documents/SYSG5/code/supports/test.txt Hello world
> Finished decoding
```

5.5 Résultat

Comparons le contenu du fichier avant et après.
Avant :

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ make build
gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ echo Hello > test.txt
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ cat test.txt
Hello
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ hexdump test.txt
0000000 6548 6c6c 0a6f
0000006
noedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

Commande encodage :

./steganography zero—width encode $(pwd)/test.txt ”Ceci est un message secret.”

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ ./steganography zero-width encode $(pwd)/test.txt "Ceci est un message secret."
Encoding finished
noedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

Après:

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ make build
gcc -g -Wall -o steganography main.o zwnj-zws.o lsb.o -lm -lpng -lz
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ cat test.txt
Hellonoedelcroix@ankond-pc:~/Documents/SYSG5/code$ hexdump test.txt
0000000 6548 6c6c e26f 8b80 80e2 e28c 8b80 80e2
0000010 e28b 8b80 80e2 e28b 8c80 80e2 e28c 8b80
0000020 80e2 e28c 8c80 80e2 e28b 8b80 80e2 e28c
0000030 8b80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
0000040 e28b 8b80 80e2 e28b 8c80 80e2 e28c 8b80
0000050 80e2 e28c 8c80 80e2 e28b 8c80 80e2 e28b
0000060 8b80 80e2 e28c 8b80 80e2 e28b 8c80 80e2
0000070 e28b 8b80 80e2 e28b 8b80 80e2 e28b 8b80
0000080 80e2 e28c 8c80 80e2 e28b 8b80 80e2 e28c
0000090 8b80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
00000a0 e28c 8b80 80e2 e28b 8c80 80e2 e28c 8b80
00000b0 80e2 e28c 8c80 80e2 e28c 8b80 80e2 e28c
00000c0 8b80 80e2 e28b 8b80 80e2 e28b 8c80 80e2
00000d0 e28b 8b80 80e2 e28b 8b80 80e2 e28b 8b80
00000e0 80e2 e28c 8c80 80e2 e28c 8b80 80e2 e28c
00000f0 8b80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
0000100 e28b 8c80 80e2 e28c 8c80 80e2 e28b 8b80
0000110 80e2 e28b 8c80 80e2 e28b 8b80 80e2 e28b
0000120 8b80 80e2 e28b 8b80 80e2 e28c 8c80 80e2
0000130 e28b 8c80 80e2 e28c 8b80 80e2 e28c 8b80
0000140 80e2 e28c 8c80 80e2 e28b 8b80 80e2 e28c
0000150 8b80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
0000160 e28c 8b80 80e2 e28b 8c80 80e2 e28c 8b80
0000170 80e2 e28c 8c80 80e2 e28c 8b80 80e2 e28b
0000180 8c80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
0000190 e28b 8b80 80e2 e28b 8b80 80e2 e28c 8b80
00001a0 80e2 e28c 8c80 80e2 e28b 8b80 80e2 e28c
00001b0 8c80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
00001c0 e28b 8b80 80e2 e28c 8b80 80e2 e28c 8b80
00001d0 80e2 e28b 8c80 80e2 e28b 8b80 80e2 e28b
00001e0 8b80 80e2 e28b 8b80 80e2 e28c 8c80 80e2
00001f0 e28c 8b80 80e2 e28b 8c80 80e2 e28c 8b80
0000200 80e2 e28c 8c80 80e2 e28b 8b80 80e2 e28c
0000210 8b80 80e2 e28c 8b80 80e2 e28c 8c80 80e2
0000220 e28b 8b80 80e2 e28b 8c80 80e2 e28c 8b80
0000230 80e2 e28c 8c80 80e2 e28c 8b80 80e2 e28b
0000240 8c80 80e2 e28b 8b80 80e2 e28c 8c80 80e2
0000250 e28b 8b80 80e2 e28c 8b80 80e2 e28c 8b80
0000260 80e2 e28c 8c80 80e2 e28c 8b80 80e2 e28c
0000270 8b80 80e2 e28b 8b80 80e2 e28b 8c80 80e2
0000280 e28b 8c80 80e2 e28c 8c80 80e2 008b
000028d
noedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

Commande décodage :

./steganography zero−width decode $(pwd)/test.txt

```
noedelcroix@ankond-pc:~/Documents/SYSG5/code$ ./steganography zero-width decode $(pwd)/test.txt
Ceci est un message secret.
Decoding finished
noedelcroix@ankond-pc:~/Documents/SYSG5/code$
```

On remarque qu'après encodage du message caché, les données sont invisibles au niveau de l'interprétation des caractères unicodes. "Hello" apparait bien dans les 2 cas (avant et après encodage), ni plus, ni moins.
Mais au niveau du contenu binaire, on voit que l'on a beacuoup plus d'octets. On a en fait 8 fois la taille du message encodé qui se sont rajoutés à la fin du fichier texte (ici 27*8 octets pour le message "Ceci est un message secret.").
les caractères UTF-8 étant codés sur minimum 2 octets (pouvant aller jusque 6 octets), le code UTF-8 du zero-width no joiner est donc 8b80 80e2 e28c et du zero-width space est 8b80 80e2 008b

6 Utilisation des codes

6.1 Démo

```
make
```

6.2 Utilisation manuelle

```
./steganography <zero-width|lsb> <decode|encode> <chemin_fichier> <
    texte_optionnel>
```

- Algorithme de stéganographie : zero-width ou lsb
- Action : encode ou decode
- Chemin vers le fichier qui stocke ou va stocker la donnée à cacher
- Texte qui va être caché (uniquement pour encode)

7 Conclusion

En conclusion, nous avons vu deux techniques de stéganographies parmies une innombrable liste de techniques.
La technique LSB permet donc de cacher des données dans une image au format PNG et est sans doute compatible avec n'importe quel format d'image codant ses pixels en RVB. Cette technique limite la taille maximale des données à cacher au nombre de pixels de l'image fois 3/8 (3 canaux de couleurs et 8 bits par caractère unicode) en octets.
La technique zero-width est un peu plus exotique et permet de cacher des données dans un fichier texte brut. Cette technique de stéganographie ne limite pas la taille des données à insérer mais celle-ci augmente considèrablement la taille du fichier, là où la technique LSB n'allourdit pas le fichier.

8 Références

References

[1] BarbDev. Libpng commands for read and write png files. https://github.com/Polytech-Projects/SteganoPngLsb/blob/master/main.c (visited: 2022-12-05).

[2] LibPNG community. Png rationale. http://www.libpng.org/pub/png/spec/1.2/PNG-Rationale.html (visited: 2022-12-05).

[3] LibPNG community. Png structure. http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html (visited: 2022-12-05).

[4] Nada Abdul Aziz Mustafa. Text hiding in text using invisible character. https://www.researchgate.net/publication/343366769$_{T}ext_{h}iding_{i}n_{t}ext_{u}sing_{i}nvisible_{c}haracter(visited : 2022 - 12 - 05)$.

[5] Nayuki. Png file chunk inspector. https://www.nayuki.io/page/png-file-chunk-inspector (visited: 2022-12-05).