

# TMA - Tatouage d'images

Alexandra Chaton - Alexis Rollin - Antoine Saget

Janvier 2021

## 1 Introduction

Ce TP a pour but d'étudier la sténographie, autrement dit l'opération qui permet de cacher un message dans une image. La sténographie étudiée sera la sténographie LSB. Cette sténographie a pour caractéristique d'écrire le message à cacher dans les bits de poids faibles du contenu hôte. Notre étude portera à la fois sur le décodage et le codage d'un tel message.

Une présentation vidéo est disponible [ici \(https://youtu.be/K6XfWXU9Pik\)](https://youtu.be/K6XfWXU9Pik).

## 2 Décodage

Décodage : Restitution d'informations codées sous leur forme originale.

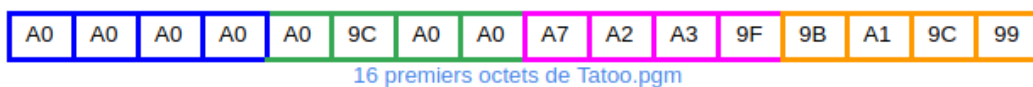
### Question 1

Toute image au format PGM possède un entête ASCII suivi des octets de l'image. Cette entête va servir à décrire le format PGM et donner les dimensions de l'image. La présence de cet entête signifie que les détails du message à cacher ne seront présents qu'après celui-ci.

Les 15 premiers octets de Tatoo.pgm, l'image où le message est caché, ne nous intéressent donc pas. C'est d'ailleurs pour cela que la fonction `load_pixmap()` renvoie les octets à partir du 16e.

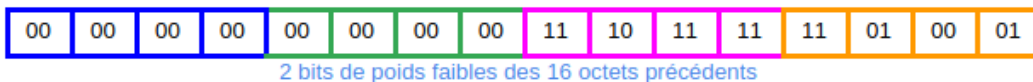
#### Taille du fichier caché

Pour retrouver la taille du fichier caché il suffit de lire les 16 premiers octets renvoyés par `load_pixmap()` :

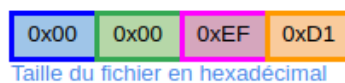


Ensuite pour chaque octet on va récupérer les deux bits de poids faibles.

*Exemple* : A0 = 10100000 donc nous nous intéressons aux bits 00.



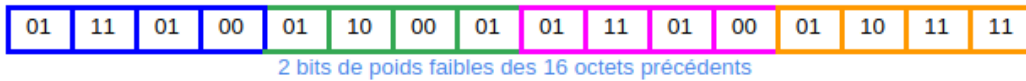
Ce qui va donner les valeurs suivantes en hexadécimal :



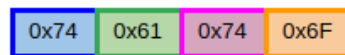
Ce qui donne comme taille de fichier caché : **61 393** octets. La totalité de ces octets est donc cachée dans l'image Tatoo.pgm mais nous ne savons pas encore exactement à partir de quel octet du fichier Tatoo.pgm ni quel est le nom du fichier caché.

## Nom du fichier caché

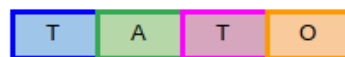
Le nom du fichier caché se trouve à la suite des octets spécifiant la taille du fichier. On va suivre les mêmes étapes que pour la taille du fichier, ce qui nous donne :



Ce qui va donner les valeurs suivantes en hexadécimal :



Ces valeurs hexadécimales correspondent aux 4 lettres suivantes :



Si on continue ces différentes étapes pour les octets qui suivent on obtient que le nom du fichier caché est : **tatoumina1.jpg**. Le nom est composé de 14 caractères il est donc codé sur 14 octets eux-mêmes répartis sur  $(14 \times 4) = 56$  octets dans Tatoo.pgm.

## Question 2

D'après la question précédente, et après analyse du fichier, nous avons pu déterminer que dans Tatoo.pgm nous avons :

- **16 octets** : qui nous servent à récupérer 4 octets déterminant la taille du fichier caché.
- **128 octets** : qui vont servir pour déterminer le nom du fichier (ici écrit sur 56 octets). Le reste sert de bourrage (ici 72 octets).

Ce qui nous fait donc un total de 144 octets. Si on ajoute le fait que l'entête du fichier caché est défini sur 15 octets, on peut déterminer que les pixels de l'image tatoumina1.jpg commencent au 159e octet.

## Question 3 :

**Objectif** : Ecrire un programme (en C ou en Java) permettant d'extraire toutes les informations cachées dans un image et de les sauver dans un nouveau fichier de sortie.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "pixmap_io.h"
6 #include "shared.h"
7
8 /**
9  * Read a full byte starting at i of pixels.
10  * This will read at pixels[i], pixels[i+1], pixels[i+2], and pixels[i+3]
11  */
12 char read_byte(unsigned char *pixels, int i) {
13     char n;
14     for (int j = 0; j < 4; j++) {
15         n = n<<2 | (pixels[i+j] & 0x03);
16     }
17     return n;
```

```

18 }
19
20 /**
21  * Read a full int starting at i of pixels.
22  * This will read from pixels[i] to pixels[i+15]
23  */
24 int read_int(unsigned char *pixels, int i) {
25     int n = 0;
26     for (int j = 0; j < 16; j++) {
27         n = n<<2 | (pixels[i+j] & 0x03);
28     }
29     return n;
30 }
31
32 void read_filename(unsigned char *pixels, char* filename) {
33     for(int i = 0; i < FILENAME_HEADER_SIZE; i++) {
34         filename[i] = read_byte(pixels, SIZE_HEADER_SIZE*4 + i*4);
35     }
36 }
37
38 void read_file(unsigned char *pixels, char* bytes, int n) {
39     for(int i = 0; i < n; i++) {
40         bytes[i] = read_byte(pixels, TOT_HEADER_SIZE*4 + i*4);
41     }
42 }
43
44 int main(int argc, char const *argv[])
45 {
46     if (argc != 2) {
47         fprintf(stderr, "Error: Incorrect number of arguments.\n");
48         printf("Usage: %s <filename>\n", argv[0]);
49         return -1;
50     }
51
52     char* filename = argv[1];
53     printf("Filename: %s\n", filename);
54
55     unsigned char *pixels;
56     int width, height;
57
58     if((pixels = load_pixmap(filename, &width, &height)) != NULL) {
59         printf("Image size: %d %d\n", width, height);
60         unsigned long available = width*height/4 - TOT_HEADER_SIZE;
61         printf("Available size: %ld\n", available);
62
63         // Reading the size
64         unsigned int n = read_int(pixels, 0);
65         printf("Message file size: %u\n", n);
66
67         // If the read message size is greater than the available size, there
68         // is a problem
69         if (n > available) {
70             fprintf(stderr, "Error: the message size is greater than the
71             available space in the file.\nEither %s is not hiding anything or
72             corrupted.\n", argv[1]);
73             return -1;
74         }
75
76         // Reading the filename
77         char filename[FILENAME_HEADER_SIZE];
78         read_filename(pixels, filename);
79         printf("Message file name: %s\n", filename);
80
81         // Reading the file
82         char bytes[n];
83         read_file(pixels, bytes, n);
84
85         // Writing the file
86         char* prefix = "tatou_";
87         char out_filename[FILENAME_HEADER_SIZE+strlen(prefix)];
88         sprintf(out_filename, "%s%s", prefix, filename);
89
90         FILE* fout = fopen(out_filename, "wb");
91         fwrite(bytes, 1, n, fout);

```

```

89     fclose(fout);
90
91     printf("Message successfully written into %s\n", out_filename);
92 } else {
93     fprintf(stderr, "Error when reading %s pixels.\n", filename);
94     return -1;
95 }
96
97 return 0;
98 }

```

## 3 Encodage

### Question 4

**Objectif :** Ecrire et tester un programme (en C ou en Java) permettant de cacher un fichier dans une image.

```

99 #include <stdio.h>
100 #include <stdlib.h>
101 #include <string.h>
102 #include <math.h>
103
104 #include "pixmap_io.h"
105 #include "shared.h"
106
107 /**
108  * Write a full byte starting at i of pixels.
109  * This will write at pixels[i], pixels[i+1], pixels[i+2], and pixels[i+3]
110  */
111 void write_byte(unsigned char *pixels, int i, char byte) {
112     for(int j = 3; j >= 0; j--) {
113         pixels[i + j] = (pixels[i + j] & 0xFC) | (byte & 0x03);
114         byte >>= 2;
115     }
116 }
117
118 /**
119  * Write a full int starting at i of pixels.
120  * This will write from pixels[i] to pixels[i+15]
121  */
122 void write_int(unsigned char *pixels, int i, int val) {
123     for(int j = 15; j >= 0; j--) {
124         pixels[i + j] = (pixels[i + j] & 0xFC) | (val & 0x03);
125         val >>= 2;
126     }
127 }
128
129 /**
130  * Write the file size at the beginning of the file
131  */
132 void write_size(unsigned char *pixels, unsigned int size) {
133     write_int(pixels, 0, size);
134 }
135
136 /**
137  * Write the filename after the file size header
138  */
139 int write_filename(unsigned char *pixels, char* filename) {
140     if(strlen(filename) >= FILENAME_HEADER_SIZE) {
141         fprintf(stderr, "Error: filename too long!\n");
142         return -1;
143     }
144
145     // Start after the 4 bytes needed for the file size
146     int start = SIZE_HEADER_SIZE*4;
147
148     int i = 0;
149     char val = filename[i];
150     // Write the filename until then end of the string
151     while (val != '\0') {
152         write_byte(pixels, start + i*4, val);

```

```

153     val = filename[++i];
154 }
155
156 // Adding a '\0'
157 write_byte(pixels, start + i*4, '\0');
158 return 0;
159 }
160
161 /**
162  * Write the file data after the file size+name header
163  */
164 void write_data(unsigned char *pixels, FILE* f, unsigned int size) {
165     // Start after the header
166     int start = TOT_HEADER_SIZE*4;
167
168     // Read the file bytes one by one
169     char val;
170     fread(&val, 1, 1, f);
171
172     // Write each byte after the header
173     for(int i = 0; i < size; i++) {
174         write_byte(pixels, start + i*4, val);
175         fread(&val, 1, 1, f);
176     }
177 }
178
179 int main(int argc, char const *argv[])
180 {
181     if (argc != 3) {
182         fprintf(stderr, "Error: incorrect number of arguments.\n");
183         printf("Usage: %s <filename> <msg filename>\n", argv[0]);
184         return -1;
185     }
186
187     char* filename = argv[1];
188     printf("File name:\t%s\n", filename);
189     char* msg_filename = argv[2];
190     printf("Tatou file name: %s\n", msg_filename);
191
192     unsigned char *pixels;
193     int width, height;
194
195     if((pixels = load_pixmap(filename, &width, &height)) != NULL) {
196         printf("Image size:\t%d %d\n", width, height);
197         unsigned long available = width*height / 4 - TOT_HEADER_SIZE;
198         printf("Available size:\t%d\n", available);
199
200         FILE* msg = fopen(msg_filename, "rb");
201         if (msg == NULL) {
202             fprintf(stderr, "Error when opening message file : %s\n",
203 msg_filename);
204             return -1;
205         }
206
207         // Go to the end of the file to get the file size.
208         fseek(msg, 0, SEEK_END);
209         long msg_size_l = ftell(msg);
210         rewind(msg);
211
212         // Make sure the file size can be encoded with 4 bytes
213         // The max size encoded with 4 bytes is 2^33-1 bytes ≈ 8.5 Gb
214         // And would need an image of at least sqrt(2^33-1)*4 = 370728*370728
215         // pixels to be hidden in
216         if (msg_size_l >= (pow(2, SIZE_HEADER_SIZE*8 + 1) - 1)) {
217             fprintf(stderr, "Error: the message file size is too big and
218 cannot be encoded with 4 bytes.\n");
219             return -1;
220         }
221         unsigned int msg_size = msg_size_l;
222         printf("Message size:\t%d\n", msg_size);
223
224         // If the msg size is greater than the available space in the file ,
225         stop
226         if (msg_size_l > available) {

```

```

223     fprintf(stderr, "Error: the message file is too big: %ld > %ld\n",
msg_size.l, available);
224     return -1;
225 }
226
227 // Hide header within pixels
228 write_size(pixels, msg_size);
229 printf("Message size successfully encoded.\n");
230
231 if(write_filename(pixels, msg_filename) != 0) {
232     return -1;
233 } else {
234     printf("Message file name successfully encoded.\n");
235 }
236
237 // Hide data within pixels
238 write_data(pixels, msg, msg_size);
239 printf("Message data successfully encoded.\n");
240
241 // Write the pixels to a new image
242 char* prefix = "coded.";
243 char out_filename[strlen(filename)+strlen(prefix)];
244 sprintf(out_filename, "%s%s", prefix, filename);
245
246 store_pixmap(out_filename, pixels, width, height);
247 fclose(msg);
248
249 printf("Message successfully hidden in %s\n", out_filename);
250 } else {
251     fprintf(stderr, "Error when reading %s pixels.\n", filename);
252     return -1;
253 }
254
255 return 0;
256 }

```

## 4 Usage

Comme vous avez pu le voir précédemment, notre implémentation du codeur et du décodeur a été faite en C. Il faut donc commencer par compiler ces programmes à l'aide du Makefile fourni :

**make**

Pour cacher un fichier dans une image au format PGM, on utilise le programme **codeur** :

```
./codeur [path_image] [path_fichier_à_cacher]
```

Le résultat de l'exécution est l'image tatouée **coded\_[nom\_image\_originale]**.

Pour extraire un fichier caché dans une image au format PGM, on utilise le programme **decodeur** :

```
./decodeur [path_image]
```

Le résultat de l'exécution est l'image extraite **tatou\_[nom\_fichier\_caché]**.

## 5 Résultats

La sténographie LSB sur 2 bits de poids faible à l'avantage d'être indétectable à l'oeil nu :



(a) Image originale



(b) Fichier à dissimuler



(c) Image tatouée

Après décodage, il n'y a aucune perte. Le message à dissimuler original et le message extrait sont strictement identiques :



(a) Image cachée originale



(b) Image extraite

Un moyen simple et infallible pour vérifier que le message original à cacher et le message extrait sont les mêmes est d'observer leurs différences. On peut utiliser la commande `diff` pour cela : elle ne renvoie rien si les fichiers sont strictement identiques.