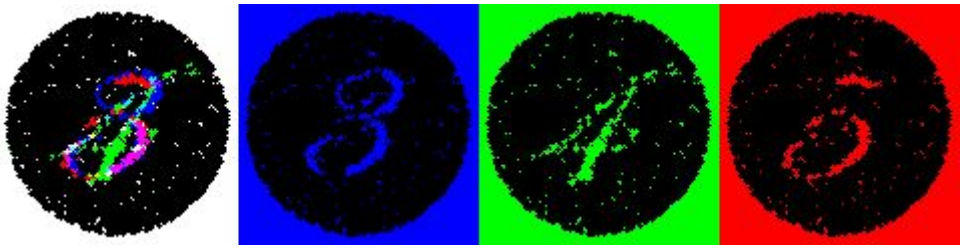
Architecture des Ordinateurs

Rapport Projet en Assembleur MIPS



Stéganographie : cacher du texte dans une image

LAFORÊT Nicolas MENY Alexandre

Université de Strasbourg

1. Introduction - L'art de la dissimulation

> *Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity.*

Le but de ce projet était de réaliser un programme, en langage assembleur MIPS, permettant de cacher du texte dans des images. Pour cela, nous devions utiliser le fait que l'oeil ne repère pas de minuscules changements de teinte dans un pixel. Nous devions donc modifier certaines teintes de pixel en ajoutant des informations permettant de coder les caractères du message secret.

Images BMP

Le format **.bmp** est un fichier bitmap, c'est-à-dire un fichier d'image graphique stockant les pixels sous forme de tableau de pixels et gérant les couleurs soit en couleur vraie soit grâce à une palette indexée.

La structure d'un fichier bitmap est la suivante :

- En-tête du fichier **14 octets**
- En-tête du bitmap / de l'image **40 octets**
- Palette de couleurs (optionnelle, si pas couleurs réelles)
- Le corps de l'image

Lors de ce projet, nous allons nous concentrer uniquement sur les images en couleurs réelles (pas de palette), codée sur 24 bits (8 bits pour le **rouge**, 8 bits pour le **vert** et 8 bits pour le **bleu**).

Encodage du texte

Le principe général pour réaliser cette opération est de lire pixel par pixel, et de générer une nouvelle image contenant les pixels modifiés. Chaque caractère du message secret (représenté par 8 bits) sera codé dans 8 octets successifs de l'image (1 bit du caractère codé dans chacun des 8 octets).

Dans les fichiers .bmp, le premier pixel est situé sur la ligne de pixels la plus basse de l'image et la colonne de pixels la plus à gauche de l'image. Cette information n'est pas nécessaire pour la réalisation de ce projet, mais nous permet que l'exemple qui suit soit plus réaliste.

Exemple : Prenons le caractère t, son code ASCII est 116, qui s'écrit en binaire 01110100. On va donc cacher 0 dans un premier octet, puis 1, puis 1, puis 1, puis 0, …

Considérons une image de 3x4 pixels qui contiendrait les pixels suivants :

145	222	210	14	57	15	78	53	11
12	0	112	121	56	89	84	12	56
200	1	201	170	65	32	122	56	232
255	12	59	78	49	154	122	11	180

Ainsi, une fois le caractère t codé dans les 8 premiers octets, on obtient :

145	222	210	14	57	15	78	53	11
12	0	112	121	56	89	84	12	56
200	1	201	170	65	32	122	56	232
254	13	59	79	48	155	122	10	180

Le caractère suivant sera codé sur les 8 prochains octets, et ainsi de suite. On commencera à coder le texte dès le premier octet de l'image.

Découpage en sous tâches

Lors de ce projet, nous avons décidé de découper le problème en différentes tâches afin de le rendre plus simple à appréhender et de pouvoir travailler simultanément sur des tâches différentes.

Le découpage s'est fait comme suit :

- Découpage d'un caractère en sa valeur ASCII binaire
- Découpage d'une chaîne de caractère en sa valeur ASCII binaire
- Ouverture / Fermeture et lecture d'un fichier
- Récupération des couleurs des pixels de l'image
- Écriture dans un fichier
- Comment indiquer que le message est fini
- Décoder le message

Une fois toutes ces tâches réalisées, il a suffit de les assembler dans un algorithme pour finir le programme d'encodage.

Quant au programme de décodage, il a simplement fallu modifier légèrement le programme pour pouvoir décoder le message caché dans une image.

2. Choix d'implémentation

Syscall 35

Pour découper un caractère sous la forme de sa valeur ASCII en binaire, nous avons premièrement remarqué dans la documentation MIPS que le syscall 35 permettait d'afficher un entier (int) sous sa forme binaire. Nous avons donc essayé de voir si cela pouvait marcher directement avec un caractère.

```
.data
char:           .asciiz "t"

.text
main:
        la      $t0, char
        lb      $a0, ($t0)
        li      $v0, 35
        syscall
Exit:
        li      $v0, 10
        syscall
```

Le programme retourne :

00000000000000000000000001110100

Ce qui est bien la valeur ASCII de la lettre "t" (avec des 0 rajoutés à gauche pour afficher 32 chiffres). Ainsi, nous avons pu remarquer que les caractères étaient directement codé en ASCII.

Découpage d'une chaîne de caractères

Suite à cette remarque, nous avons pu réaliser un petit algorithme permettant d'afficher **bit par bit** la valeur ASCII d'un caractère, puis d'une **chaîne de caractère** sans utiliser le syscall 35.

Le programme suivant est commenté dans le fichier *steganographie_encode.asm*

Ce programme affiche :

```
Entrez un message : At      # L'utilisateur entre la chaîne "At" au clavier
01000001                    # Valeur de "A" en ASCII binaire
01110100                    # Valeur de "t" en ASCII binaire
```

```
.data
chariot:                .asciiz "\n"
ask_input:              .asciiz "Entrez un message : "
msg:                    .space 128

.text
main:
        li      $v0, 4
        la      $a0, ask_input
        syscall

        li      $v0, 8
        la      $a0, msg
        li      $a1, 128
        syscall

loop_char_init:
        li      $t0, '\n'
        li      $t1, 128
        li      $t2, 0

loop_char:
        beq     $t2, $t1, loop_char_end
        lb      $t3, msg($t2)
        addi    $t2, $t2, 1
        beq     $t3, $t0, loop_char_end

        loop_bits_init:
                li      $t4, 0
                li      $t8, 7

        loop_bits:
                blt     $t8, $t4, loop_bits_end
                move    $t5, $t3
                srlv    $t5, $t5, $t8
                sll     $t5, $t5, 31
                srl     $t5, $t5, 31
                subi    $t8, $t8, 1

                move    $a0, $t5
                li      $v0, 1
                syscall

                bge     $t8, $t4, loop_bits

        loop_bits_end:
                li      $v0, 4
                la      $a0, chariot
                syscall
                bne     $t3, $t0, loop_char
loop_bits_end:

Exit:
        li      $v0, 10
        syscall
```

Ouverture / Fermeture et lecture d'un fichier

Cette partie a été assez rapide, il a suffit de bien lire la documentation MIPS. Le syscall 13 permet d'**ouvrir un fichier**, le syscall 14 permet de **lire un fichier** et le syscall 16 permet de **fermer un fichier**.

Il ne faut pas oublier de récupérer le ***descripteur de fichier*** après l'ouverture du fichier afin de pouvoir le fermer plus tard avec le syscall 16.

Attention, pour l'ouverture d'un fichier, il faut que le noms du fichier soit dans une chaîne de caractère se finissant par un caractère **null**. Pour cela nous avons rajouté une fonction permettant de changer le '\n' (qui apparaît lorsqu'on a fini de taper le nom du fichier) par le caractère **null**.

Récupération des couleurs des pixels de l'image

Dans la boucle de traitement des pixels, il suffit de se déplacer octet par octet, car dans la partie de l'image les fichier .bmp, les couleurs de chaque pixel sont disposées les unes après les autres en commençant par le **bleu**, puis le **vert** et enfin le **rouge**.

En incrémentant la position dans le tableau de pixels d'un octet, on se déplace de couleur en couleur.

Écriture dans un fichier

Pour l'écriture dans un fichier, c'est comme l'ouverture, la fermeture et la lecture, il suffit de bien lire la documentation, il faut utiliser le syscall 15.

Il ne faut pas pour autant oublier qu'avant d'écrire dans un fichier, il faut l'ouvrir avant, en récupérant le ***descripteur de fichier*** car il est nécessaire lors de l'écriture. De plus, il ne faut pas non plus oublier de fermer le fichier.

La fin du message ?

Avec l'algorithme à ce niveau, nous nous sommes demandés comment nous allions indiquer que le message était fini. Pour cela, nous avons optés pour l'écriture de **huit 0 consécutifs** à la fin du message, toujours dans les bits de poids faibles des couleurs des pixels.

Décoder le message

Enfin, la dernière partie de ce projet fut la réalisation du programme de décodage. Globalement la structure était la même. La seule différence étant qu'on ne change pas l'image, on utilise simplement la lecture et la récupération des bits de poids faibles des couleurs des pixels jusqu'à tomber sur huit 0 consécutifs.

Pour cela, nous avons récupéré les bits de poids faibles par paquet de 8, en effectuant un décalage à gauche commençant à 7 bits pour le premier bit récupéré, puis 6 pour le deuxième, jusqu'à un décalage à gauche de 0 bit pour le 8e. Tout en les additionnant dans une autre variable afin d'avoir la valeur ASCII des caractères.

Exemple (pseudo code): Pour le caractère "t"

$a0 = 0

1e bit récupéré : $a0 += 0 << 7
2e bit récupéré : $a0 += 1 << 6
3e bit récupéré : $a0 += 1 << 5
4e bit récupéré : $a0 += 1 << 4
5e bit récupéré : $a0 += 0 << 3
6e bit récupéré : $a0 += 1 << 2
7e bit récupéré : $a0 += 0 << 1
8e bit récupéré : $a0 += 0 << 0

À la fin de la boucle $a0 vaut 116 qui est bien la valeur ASCII de "t"

Ensuite il suffit d'utiliser le syscall 11, qui permet d'afficher un caractère depuis sa valeur ASCII.

3. Difficultés rencontrées

Lors de ce projet, la principale difficulté a été le langage de programmation, si on ne commente pas notre code il est très difficile de le relire, ce qui est problématique lorsqu'on travaille en groupe. Ainsi, nous avons fait attention de commenter chacune de nos instructions afin de ne pas perdre de temps à essayer de retrouver ce que fait telle ou telle partie de code.

4. Conclusion

Pour conclure, de projet nous a permis de remarquer l'importance de la documentation et des commentaires dans le code, car lorsqu'on travaille en groupe, on ne pense pas forcément de la même manière. Nous avons aussi pu remarquer l'importance du découpage en sous tâches, lors d'un problème imposant. En effet peu importe la taille du problème, si on le découpe en plusieurs petits problèmes, le projet devient tout de suite plus simple.