Projet Nigma : Deuxième soutenance

CrypTeam : LAPÔTRE Guillaume (lapotr_g)
GANIVET Justin (ganive_j)
LADEVIE Stéphane (ladevi_s)
GISLAIS Sébastien (gislai_s)

8 avril 2009

Table des matières

1 Introduction

Nous voilà arrivé à la deuxième soutenance, à mis chemin de la conclusion du Projet Nigma, le logiciel qui révolutionnera (au moins) l'univers mystérieux de l'espionnage, de la duperie et du mensonge. Rappelons nous en quoi consiste exactement le Projet Nigma de la Crypteam : il s'agit d'un logiciel associant des techniques de cryptages modernes (tel que le DES et le RSA) à l'un des plus vieux principe du monde en matière de transfert protégé d'informations : la stéganographie. Ainsi l'utilisateur du programme pourra crypter diverses données et les cacher dans une innocente image. Il pourra ensuite faire transiter cette image via Internet, ainsi il sera aisé au destinataire muni du même logiciel de récupérer les précieuses données (plan d'invasion ou recette de cuisine). L'indélicat indiscret qui tenterait s'approprier ces fameuses données n'aurait d'autres moyens que de savoir quelle image les dissimule, de posséder le logiciel ainsi que les clefs de cryptage, bref c'est pas gagné. L'intérêt du Projet Nigma repose sur l'expansion des réseaux sociaux sur la toile tel que Facebook, Myspace ou encore les « blogs » qui sont de véritables plateformes d'échanges d'images et de photographies en tout genre, des échanges qui ne sont pas surveillés et en apparence tout à fait innocents. La force du logiciel repose en majeur partie sur cette « faille » offrant un moyen de communication insoupçonnable, immédiat et mondial.

MISE EN GARDE : Nous vous rappelons que le logiciel n'est initialement pas prévu à des fins malhonnêtes et crapuleuses (tel que l'espionnage ou le terrorisme...). Dans le cas contraire la Crypteam ne saurait être tenu pour responsable de l'utilisation déviante. L'utilisateur est invité à assumer pleinement ses actes devant la loi sans essayer d'y impliquer les concepteurs du logiciel. Merci de votre compréhension.

À présent la Crypteam est fière de vous présenter le rapport de la seconde soutenance du Projet Nigma, dans lequel vous aurez l'immense plaisir de découvrir les nouvelles fonctionnalités du logiciel !

2 Cryptographie

Cette fois-ci encore, nous avons respecté ce que nous avions prévu de présenter pour cette seconde soutenance. En effet nous avons implémenté l'algorithme DES (Data Encryption Standard). Contrairement au RSA présenté en première soutenance, le DES est un algorithme de chiffrement symétrique. Cela signifie que pour chiffrer ou déchiffrer un fichier on utilise la même clef. Ce type de chiffrement a ses avantages et inconvénients par rapport au chiffrement asymétrique. Son principal avantage est sa rapidité : il est 1000 fois plus rapide que le RSA ! Son principal inconvénient est le fait de ne posséder qu'une seule clef. Ainsi le problème principal est l'échange de cette clef entre les différents protagonistes qui veulent échanger des données chiffrées. En effet pour le RSA il suffisait de donner sa clef publique aux personnes désirant chiffrer des messages, ensuite ils renvoient les messages chiffrés et on peut les déchiffrer à l'aide de notre clef privé jamais échangé avec personne. Si l'on envoie notre clef DES au n'importe qui, il y a un risque qu'un pirate puisse intercepté la clef et donc s'en servir ensuite pour déchiffrer les messages confidentiels. Cependant il y a une astuce permettant de contourner le problème : On chiffre la clef DES avec un chiffrement RSA !

Un petit exemple : Alice veut envoyer un message confidentiel à Bob. Bob crée donc un jeu de clef RSA puis envoie la clef publique à Alice. Alice crée sa clef DES, puis chiffre son message à envoyer avec cette clef. Finalement, elle envoie à Bob sa clef DES chiffrée avec la clef publique RSA ainsi que le message chiffré avec la clefs DES. Bob reçoit donc les deux fichiers, avec sa clef privé RSA il déchiffre le fichier de clef envoyé par Alice, puis avec la clef DES qu'il vient de déchiffrer il peut déchiffrer le message d'Alice !

Le DES a été achevé en 1977, c'est donc un algorithme de chiffrement ancien. Sa sécurité n'est plus optimale, en effet un état peut casser une clef DES en quelque minutes maintenant. Cependant nous pensons qu'il est intéressant de se pencher sur cet algorithme qui fût un des premiers algorithmes de chiffrement symétrique défini rigoureusement.

Du fait de son ancienneté les spécifications du DES étaient prévus pour une réalisation matérielle. C'est à dire pour que ce soit des puces qui réalisent le chiffrement. Ainsi certaines partie du protocole sont faciles à réaliser sur une puce mais sont difficile à réaliser en Ocaml. Nous développerons dans nos parties respectives.

2.1 Partie de Guillaume Lapôtre

Pour implémenter le DES en Ocaml je me suis tout d'abord documenté sur cet algorithme de chiffrement complexe bien qu'utilisant que des opérations basiques (en effet le DES est optimisé pour une réalisation matérielle...). Je me suis donc basé sur un livre qui m'a été bien utile pour la l'implémentation de RSA qui se nomme *Cryptographie appliquée* par Bruce Schneier. Ce livre est très pratique car il explique très bien comment fonctionne un bon nombre d'algorithmes de chiffrement et il y a aussi l'histoire de chaque algorithme : Comment il a été créé, par qui, suite à quels besoins, etc.

J'ai donc tout d'abord commencé par la création d'un clef DES. Sa création fût bien plus simple que la création d'une paire de clefs RSA. En effet, la génération des clefs RSA nécessitent de posséder au préalable de grands nombres premiers qui furent quelque peu difficile à obtenir. Une clef DES est un nombre de 64 bits. Puis le DES ne se sert que de 56 bits des 64 bits présents initialement. Les 8 bits inutilisés sont des bits de parités. Ils servent à vérifier lorsque l'on envoie la clef que la clef reçue n'est pas corrompue. C'est à dire qu'aucun bit n'a changé au court du transfert. Ainsi, une clef DES à la propriété d'avoir un nombre pair de bits à 1 pour chaque octet. Ce sont les 8 bits de parité qui permettent d'assurer cette propriété.

On remarque déjà que l'on va manipuler beaucoup de données au bit à bit. Je me suis donc poser la question : Comment manipuler des nombres codés sur 32 à 64 bits. J'ai donc tout d'abord créé des fonctions permettant les manipulations de base sur les bits d'un nombre : accéder au i^{e} bit d'un nombre et mettre sa valeur à 0 ou 1. Puis, pour faciliter la manipulation des bits de mes nombres et la vérification de mes fonctions j'ai codé une fonction qui nous a servi tout au long de la réalisation du DES, cette dernière prend un nombre en paramètre et renvoi un tableau contenant dans chacune de ses cases les bits du nombre. Ainsi, dans la case 0 il y a le bit numéro 0 et ainsi de suite. J'ai bien sur codé sa fonction réciproque qui prend en paramètre un tableau contenant tout les bits d'un nombres et qui renvoie le nombre. Avec ces deux fonctions on à pu réaliser tout les manipulations que l'on voulait sur nos blocs.

Je me suis ensuite occupé des différentes fonctions de manipulation de la clef. Tout d'abord il y a une permutation de clef qui réarrange les bits dans un ordre prédéfini et qui ignore les bits de parité. On se retrouve donc avec une clef de 56 bits. Ensuite à chaque ronde du DES[1], on sépare la clef en deux sous-clefs de 28 bits chacune et on effectue une ou deux rotations gauche suivant la ronde que l'on est en train d'effectuer. Ensuite on réassemble les

[1] le protocole exacte d'une ronde du DES sera expliquée par Sébastien

deux sous clefs puis on effectue une permutation compressive qui compresse la clef qui était de 56 bits en une clef de 48 bits. L'ordre dans lequel les bits sont réarrangés est spécifié dans le DES. Ainsi j'ai pu générer les 16 sous-clefs requises pour le chiffrement d'un bloc.

Je me suis ensuite occupé d'une toute autre partie dans l'algorithme DES qui est la gestion des S-box ou tables de substitution. Ces tables sont le point-clef dans la sécurité du DES. En effet elles sont les seuls éléments non linéaires dans cet algorithme et elles confèrent à l'algorithme son niveau de sécurité. La chose qui pourrait paraître étonnante est que les 8 tables de substitutions du DES sont totalement publiques maintenant. Ces tables de substitution se représentent initialement sous la forme d'un tableau à deux entrées. La méthode permettant d'accéder à la case du tableau que l'on recherche est un peu particulière[2]. Nous avons donc initialement un bloc de 48 bits que nos tables de substitutions vont transformer en un bloc de 32 bits de façon absolument pas linéaire.

Nos tables ont 4 rangs et 16 colonnes. Découpons notre bloc de 48 bits en 8 sous-blocs de 6 bits. Prenons notre premier sous-bloc et nommons les bits qui le compose : b1 b2 b3 b4 b5 b6. Le nombre composé par les deux bits b1 b6 va permettre la sélection du rang puis les 4 autres bits b2 b3 b4 b5 vont permettre de sélectionner la colonne, le tout sur la table de substitution n°1. Ainsi nous avons donc sélectionner une case de notre tableau qui contient un nombre sur 4 bits. En faisant de même pour les 8 sous-blocs on a donc utilisé les 8 tables de substitutions et récupéré les 32 bits escomptés.

On peut remarquer que la sélection d'une case dans une table n'est pas forcément une chose aisée due à la façon dont on utilise les bits de nos sous-blocs. En effet en réalisation matérielle cette sélection ne pose pas de problème par contre en réalisation logicielle celle-ci n'est pas optimisée. J'ai donc réarrangé les s-box pour obtenir 8 tableaux à une dimension en remettant bien dans l'ordre les bits b1 b2 b3 b4 b5 b6. Ainsi lorsque j'utilise une table je n'ai besoin que du bon sous-bloc de 6 bits pour récupérer sa représentation décimale et avoir le numéro de la case du tableau que je veux ! Par ailleurs, cela me permet aussi de limiter les risques de mauvaise utilisation des tables de substitutions.

Voici un exemple de l'utilisation d'une table de substitution avec le nombre 27 dont la représentation en binaire est %011011

[2]Encore une fois, Sébastien expliquera dans sa partie exactement quand est-ce que l'on s'en sert.

S_5		4 bits au centre de l'entrée															
		0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1100**	**1011**	**1100**	**1101**	**1110**	**1111**
Bits externes	**00**	0010	1100	0100	0001	0111	1100	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1100	0011	1001	1000	0110
	10	0100	0010	0001	1011	1100	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1100	0100	0101	0011

Voici une S-box où l'on voit bien le nombre "b1 b6" pour la sélection de la ligne et le nombre "b2 b3 b4 b5" pour la sélection de la colonne

On voit donc sur l'image ci-dessus quelle cellule est sélectionnée. La table de substitution nous renvoie donc %1001 soit en décimal le nombre 9.

Puis j'ai travaillé sur la création de la fonction principale. Cette dernière gère les fonctions de chiffrage de données, de déchiffrage ainsi que la fonction de création de la clef. Cette fonction gère aussi tout la partie accès et lecture/écriture dans les différents fichiers utiles. Ainsi, lorsque l'on appelle le programme avec le paramètre `--clef`, le programme crée un fichier nommé clefDES et écrit une clef dedans avec les bits de parité à jour. Lorsque l'on appelle le programme avec le paramètre `--chiffre`, il faut lui donner en deuxième paramètre le fichier à chiffrer et en troisième paramètre le fichier contenant une clef DES. Le programme ouvre le fichier contenant la clef pour la récupérer puis crée les 16 sous-clef. Ensuite il ouvre le fichier à chiffrer et récupère les 8 premiers caractères. Comme un caractère est représenté sur un octet donc sur 8 bits, les 8 caractères lu donnent donc 64 bits. Ensuite on chiffre ces 64 bits avec les 16 sous-clefs et on écrit les 8 nouveaux caractères correspondant au bloc de 64 bits chiffré. Puis on fait de même pour chaque bloc de 8 caractères du fichier à chiffrer. Comme un fichier ne contient pas forcément un multiple de 8 caractères et, par contre, le DES à besoin de bloc de 64 bits, nous avons trouvé une astuce pour que le dernier bloc soit aussi chiffré : On complète le bloc avec des caractère espace pour avoir au final un bloc de 8 caractères. C'est pour cela que nos fichiers chiffrés contiennent tout le temps un multiple de 8 caractères.

2.2 Partie de Sébastien Gislais

Les *Federal Information Processing Standards* (FIPS) sont des standards publics développés et annoncés par le gouvernement des États-Unis pour l'usage des agences gouvernementales non militaires et entrepreneurs gouvernementaux (*government contractors*). Beaucoup de standards FIPS standards sont des versions modifiées des standards ANSI, IEEE, ISO, etc.

Quelques standards FIPS ont été originellement développés par le gouvernement des États-Unis. Par exemple, les standards pour encoder des données (ex : code pays), mais plus significativement, des algorithmes de chiffrement tel que *Data Encryption Standard* (DES) (FIPS 46) et *Advanced Encryption Standard* (AES) (FIPS 197).

Le premier standard DES est publié par FIPS le 15 janvier 1977 sous le nom FIPS PUB 46. La dernière version avant l'obsolescence date du 25 octobre 1999 FIPS PUB 46-3.

Nous avons utilisé la publication 46-3 du FIPS comme support pour notre implémentation du DES. Cette documentation officielle, en Anglais, nous a permis de confirmer l'exactitude des explications que nous avions trouvées dans l'ouvrage *Cryptographie appliquée* (ISBN 2-7117-8676-5) du cryptologue Bruce Schneier. Ce standard de cryptographie est donc parfaitement documenté, clair et précis, comme devrait l'être tout standard. Nous pouvons nous pencher maintenant sur sa réalisation logicielle. En premier avertissement, nous découvrons que le DES est conçu avant tout pour une réalisation matérielle. Loin de nous imaginer à première vue la difficulté d'une implémentation logicielle, nous commençons l'étude de l'algorithme.

Le DES est un système de chiffrement par blocs ; il chiffre les données par blocs de 64 bits. La longueur de la clef est de 56 bits. Généralement, la clef est exprimée comme un nombre de 64 bits avec un bit sur huit utilisé comme bit de contrôle de parité. Ces bits de parité ne rentrent pas en compte dans l'utilisation de la clef lors du chiffrement ou du déchiffrement.

Le DES a 16 rondes, c'est-à-dire qu'il applique 16 fois la même combinaison de techniques au bloc de texte en clair (voir figure 1 page 9). Les techniques de confusion et de diffusion de l'algorithme sont la répétition d'une substitution suivie d'une permutation. Comme ces opérations sont uniquement arithmétiques et logiques, il est très facile de réaliser l'algorithme matériellement avec des puces spécialisées qui pourront l'exécuter très rapidement.

Le DES manipule le texte en clair par blocs de 64 bits. Après une permutation initiale, le bloc est coupé en une partie droite et une partie gauche, chacune d'une longueur égale de 32 bits. Après cela, il y a 16 rondes d'opérations identiques, appelées « fonction f », lors desquelles les données sont combinées

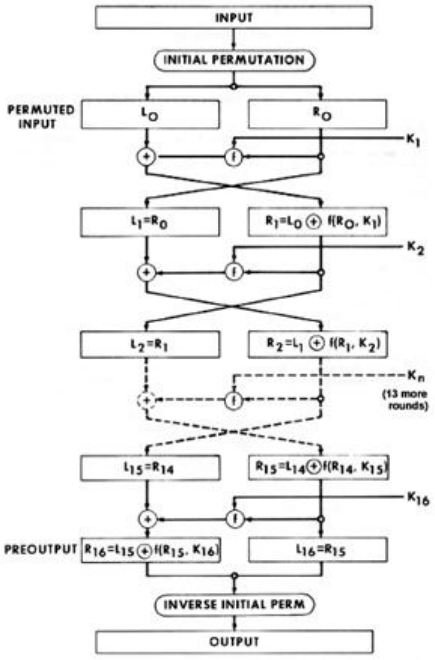


Fig. 1 – DES

à la clef. Après la 16e ronde, les parties droite et gauche sont réassemblées et une permutation finale (l'inverse de la permutation initiale) termine l'algorithme.

À chaque ronde (voir figure 2 page 10), les bits de la clef sont décalés et 48 bits sont alors sélectionnés parmi les 56 bits de la clef (permutation compressive, voir figure 3 page 11). La partie droite des données est étendue à 48 bits par une permutation expansive (appelée E) puis combinée avec 48 bits de la clef décalée et permutée par un *ou exclusif*, et ensuite remplacée par 32 nouveaux bits par un algorithme de substitution (avec les tables de substitution 1 à 8) et permutée une fois de plus.

La « fonction f » est constituée de ces quatre opérations. La sortie de la « fonction f » est alors combinée avec la moitié gauche par un *ou exclusif*. Le résultat de ces opérations devient la nouvelle moitié droite ; l'ancienne moitié de droite devient la nouvelle moitié gauche. Si B_i est le résultat de la i^{e} itération, L_i et R_i respectivement des moitiés gauche et droite de B_i, K_i est la clef de 48 bits pour la i^{e} ronde, et f est la fonction qui fait toutes les substitutions, permutations et *ou exclusif* avec la clef comme dit plus haut, alors une ronde est décrite par :

$$L_i = R_{i-1}$$

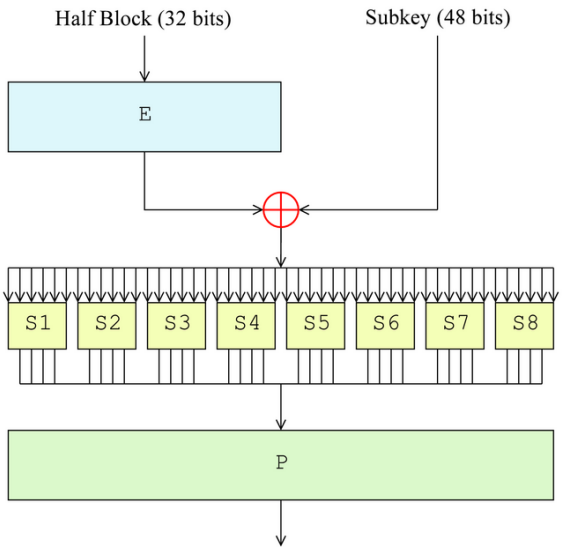


FIG. 2 – Une ronde du DES

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Ces opérations sont répétées 16 fois, donnant le DES à 16 rondes.

Comme on peut le voir, l'algorithme de chiffrement peut-être construit très facilement avec du matériel électronique simple (portes logiques, *ou exclusif*, substitution de bits) qui ne coûtent rien en temps d'exécution. En revanche, les substitutions de bits en logiciel sont très coûteux car on effectue un parcours itératif complet de nos blocs de bits plusieurs fois.

Caml est un langage de programmation de très haut niveau qui est très utile pour programmer en faisant abstraction du matériel ou de la représentation machine des données et permet de se focaliser sur l'algorithmique. Oui, Caml n'est absolument pas adapté pour une utilisation très bas niveau qu'est le DES avec des manipulations bit à bit à outrance.

Néanmoins, cela n'arrête pas la Crypteam qui réalise une implémentation du DES exceptionnelle : 5 secondes pour chiffrer une image de quelques dizaines de kilo-octets. Nous obtenons le même temps pour le déchiffrement de ce même algorithme, alors que pour le RSA le temps de déchiffrement était 100 fois supérieur au chiffrement !

Je me suis occupé de réaliser les opérations successives du DES, c'est-à-dire les permutations, l'utilisation des S-Boxes préalablement optimisées pour l'itératif par Guillaume, les expansions et enfin combiner le tout pour obtenir les 16 rondes demandées par l'algorithme.

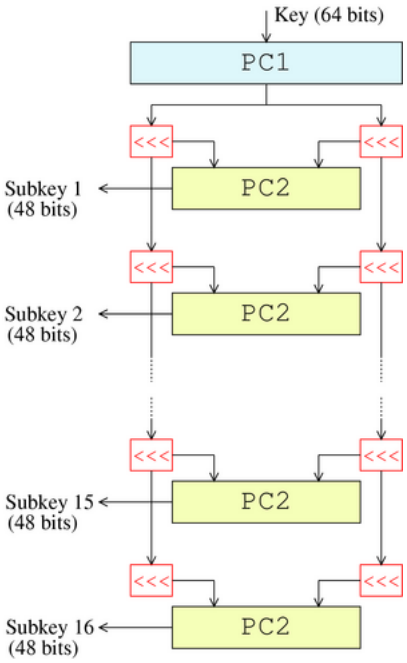


FIG. 3 – Création des clefs compressées

Un suivit scrupuleux de la documentation officielle et du livre de Bruce Schneier m'a permis de réaliser un code clair, compréhensible et efficace en Caml. On pourra facilement l'étudier et le reprendre par la suite si l'humeur nous en dit.

3 Stéganographie

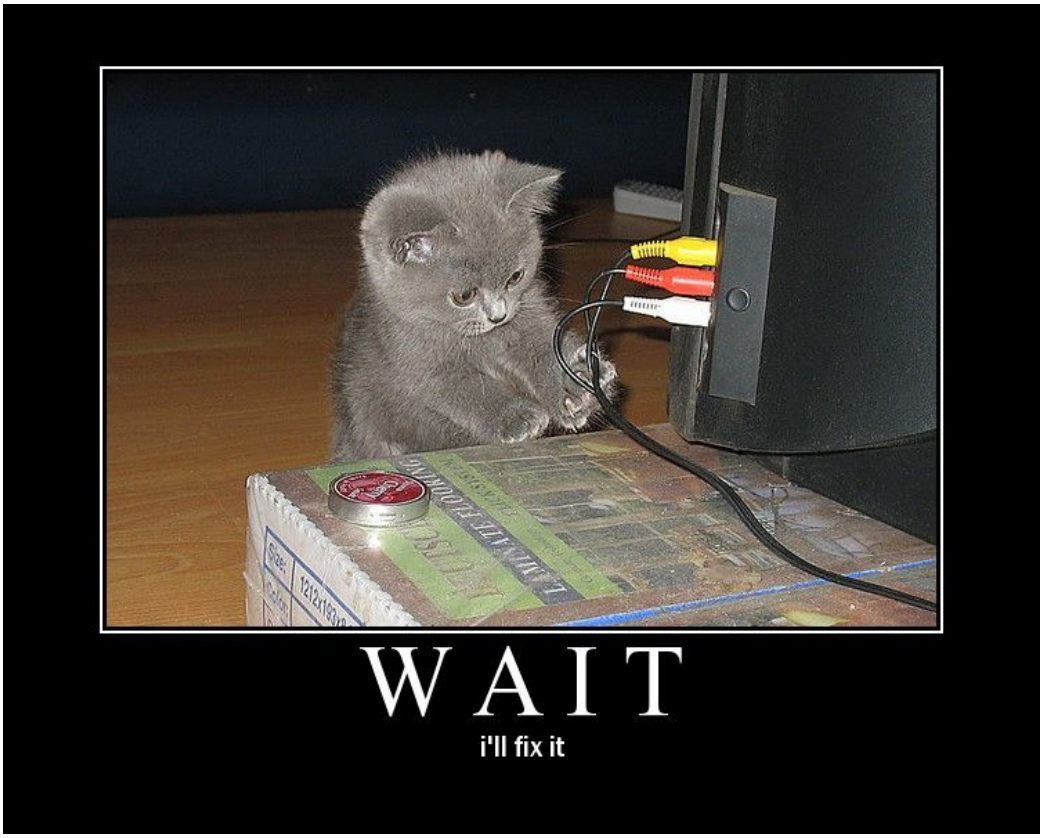
Rappelons l'objectif principal de l'équipe stéganographie pour cette seconde soutenance : trouver un nouveau système de stéganographie pour cacher plus de données que la technique présentée en 1re soutenance (baptisée sobrement : StefOne-MS01) sans perdre pour autant le « camouflage » des dites données. Après de longues heures d'investigation au cœur de nos cerveaux malades nous avons réussi à trouver une technique qui, bien que moins discrète, peut contenir 3 fois plus de données que le petit frère StefOne-MS01 : Just-Inferno-MS02 ! Par soucis de clarté nous les appellerons MS01 (Méthode de Stéganographie 01) et MS02 (Méthode de Stéganographie 02) dans la suite du rapport de soutenance.

3.1 Partie de Justin Ganivet

Pour cette soutenance, je me suis principalement occupé de l'interface graphique du Projet Nigma. Et la tâche n'était pas aisée puisque je n'ai aucune expérience dans le graphisme. En demandant à mes contacts, je me suis tourné vers Gtk+. Pourquoi Gtk ? Il n'avait pas l'air très compliqué et il est très optimisé pour créer des interfaces. J'ai donc lu plusieurs tutoriaux et entrepris de créer ma première fenêtre Gtk. Deux, trois fonctions, les flags de compilation qui vont bien et j'avais une fenêtre Gtk d'ouverte. C'était donc beaucoup plus facile que DirectX où il fallait passer quarante-deux heures pour ouvrir une fenêtre.

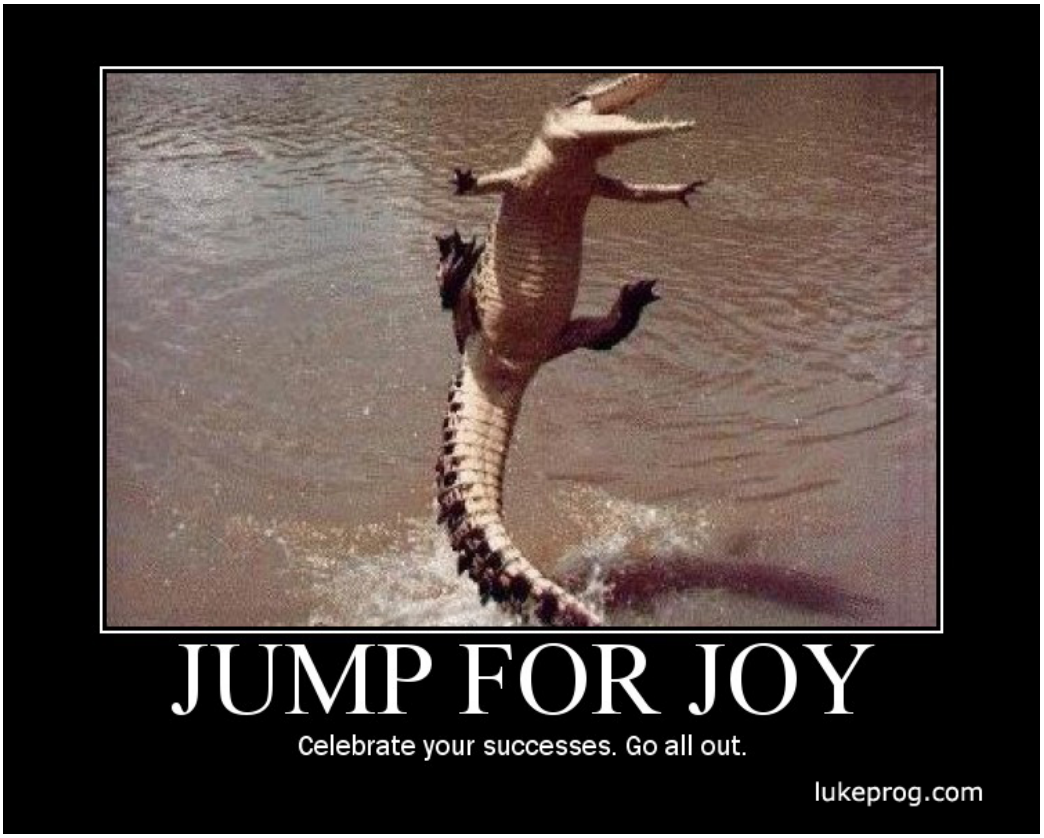
J'étais pourtant toujours très rétissant à l'idée de faire du graphique jusqu'au moment où on me parla de Glade. Glade est un programme générant une interface pouvant être utilisée par Gtk. De plus, Glade est un WYSIWYG, ce qui est très pratique pour générer l'interface que l'on souhaite. Alors que demander de plus ?

Avec l'aide de Stéphane, nous avons donc créé notre interface graphique. Elle devait être séparée en trois parties, une partie encodage, une autre décodage et enfin un aperçu de l'image. Une fois notre interface créée et enregistrée en .glade, il nous fallait l'intégrer au programme. J'avais découvert une fonction de Gtk qui permettait de charger une interface XML directement. Je l'ai donc incorporé au code de départ. Le problème était que je n'arrivais pas à trouver comment charger les *widgets* puis affecter des signaux de ces derniers.



Puis j'ai découvert que dans Glade, on pouvait attribuer les signaux directement dans le XML. Je me suis donc dit qu'il devait avoir un moyen de charger tout ces signaux facilement. J'ai découvert la solution au travers de la libglade. La libglade intègre permet d'ajouter très facilement une interface XML dans son projet mais aussi de charger facilement les *widgets* à partir du XML. Mieux encore ! On peut laisser la libglade attribuer les signaux aux *widgets* tout seul comme un grand. Le grand hic est que bizarrement la libglade s'est comportée comme un enfant capricieux avec moi. Elle m'a envoyé massivement des `seg fault`, des *warnings* et autre joyeuseries. J'ai donc enlevé cette fonction mystique et ai réfléchi à une autre possibilité.

J'ai donc chargé depuis le XML tout les *widgets* qui devait être connecté aux signaux en les mettant dans une structure regroupant tout les pointeur vers ces objets. Ensuite j'ai attribué ces derniers manuellement grâce aux fonctions de Gtk. Et là, magie ! Plus de `seg fault`, ma fenêtre se fermait correctement, le bonheur !

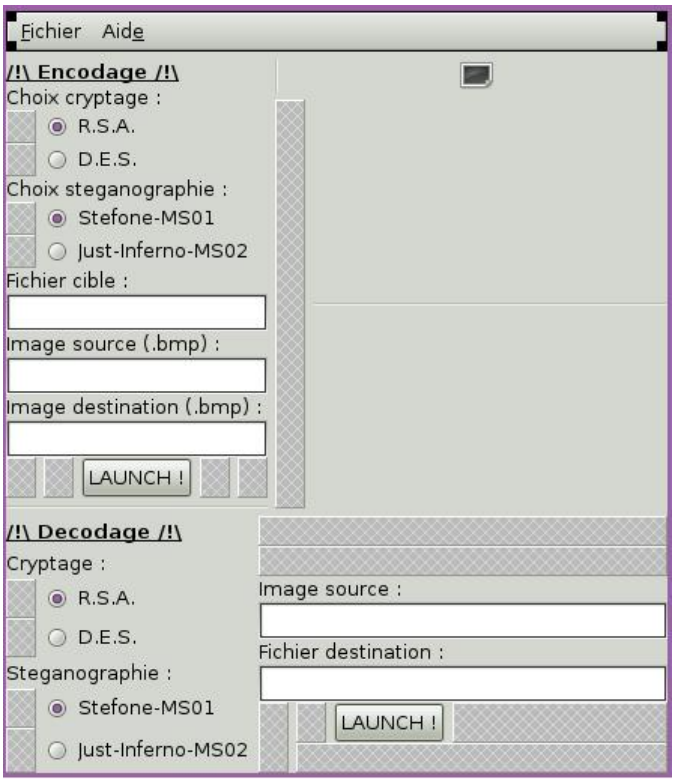


Il me restait donc à coder ces fameux signaux. Pour ce faire, j'ai crée une mégastructure qui regroupe l'état de l'interface. Ainsi, je stocke le nom du fichier source, le nom de l'image source, le nom de l'image de destination et de quelle manière on veut crypter et stéganographier. Cette structure a toute son utilité dans les signaux attribués aux boutons. Ainsi, dès que l'on clique sur l'un des deux boutons, la fonction va regarder l'état de la structure. La fonction va tester si on est en encodage ou en décodage. Puis on va *forker* le programme et exécuter soit le binaire de cryptologie ou celui de stéganographie. Le père attend que le binaire ait fini puis *fork* de nouveau et exécute le programme complémentaire.

Pour l'instant, l'exécution des binaires se fait sans les arguments ce qui lance l'aide. Ce n'est pas très intéressant pour le moment mais pour la troisième soutenance nous allons pouvoir lancer directement ces binaires depuis l'interface. Ensuite, pour indiquer les différents fichiers il faut taper manuellement le chemin des fichiers. Nous projetons pour la prochaine soutenance de lancer une boîte de dialogue pour pouvoir ouvrir plus facilement ces fichiers. Nous avons aussi un aperçu d'image qui affiche une image prédéterminée. Cette zone sera par la suite utilisée pour afficher ce que don-

nera l'image en fin d'encodage. On peut imaginer par la suite une barre de défilement, une aide directement dans l'interface et un menu en haut.

En conclusion, Gtk est un outil très puissant mais qui demande un peu de pratique avant de faire quelque chose de bien. Glade reste indispensable pour créer une interface digne de ce nom. La libglade quant à elle est difficile à gérer même si je pense qu'elle doit être très puissante et efficace.



3.2 Partie de Stéphane Ladevie

3.2.1 Encodage

Pour commencer nous allons rappeler le principe de la MS01 et nous verrons ensuite celui de la MS02 afin de mieux les comparer. Souvenez vous, la MS01 considérait un caractère comme une chaîne de 9 bits (les 8 bits du système ASCII plus 1 bit servant à conserver la structure pixel), cette chaîne est cache au sein de 3 pixels en changeant la parité des champs de couleurs RGB des pixels. Nous avions donc un ratio de un caractère pour 3 pixels ce qui correspond à 160 ko de données pour une image 800×600.

La MS02 quant à elle va cacher un caractère entier dans un unique champ de couleur, on va donc avoir 3 caractères cache au cœur d'un pixel, très

rentable mais fort peu discret puisque l'image sera modifie intégralement... Nous avons donc exécuté cette opération tout les 3 pixels afin de conserver un certain camouflage de l'opération sans pour autant perdre l'avantage de la MS02 : sa capacité de stockage supérieur. En effet en faisant ainsi on obtient un ratio de 1 caractère pour 1 pixel ce qui correspond pour une image 800×600 a une capacité de 480 ko de données. L'autre intérêt de cette technique et sa rapidité d'exécution ce qui peut s'avérer fort pratique dans les situations périlleuses que l'on trouve généralement dans *Black & Mortimer* mais aussi dans la vraie vie !

Problème

Nous avons rencontre un problème de discrétion sur les images de bonne qualité, en effet modifier un pixel tout les 3 pixels s'est révélé trop flagrant (on obtient une sorte de pelouse en bas de l'image, efficace sur une photo de paysage mais beaucoup moins sur une image de Clichy-sous-Bois...). Nous avons donc crée un système qui va repartir les pixels a modifier sur toute l'image en fonction de la taille de l'image et de la taille des données. Ainsi on obtient une image beaucoup plus homogène et donc beaucoup plus discrète, ceci reste valable tant que la capacité maximale de l'image n'est pas atteinte, dans le cas contraire on obtiendra l'écart initial de 3 pixels et l'utilisateur devra choisir l'image la plus appropriée au transfert de ses données.

La partie du Castor Bricoleur

Voyons un peu ce qui se cache dans les entrailles de la bête MS02... Fondamentalement le code ressemble beaucoup à celui de la MS01 puisque l'on parcourt le fichier de la même façon à l'aide de la même structure pixel (petit rappel) :

```
struct pixel
{
  int blue;
  int red;
  int green;
};
```

Les changements majeurs sont :

- le calcul de la variable i qui déterminera l'écart entre les pixels codés ainsi que son intégration à l'image,
- le nouveau système permettant le codage de 3 caractères en 1 pixel.

Avant de réaliser l'encodage proprement dit il nous faut calculer l'écart qui séparera les pixels modifiés, on va calculer cet écart "i" en fonction de la taille de l'image à l'aide de la fonction :

```
int get_i(char *s, int taille)
```

`get_i` prend en paramètres le nom de l'image que l'on désire utiliser et la taille de la chaîne de caractères que l'on désire cacher. Les informations nécessaire à la détermination de i (hauteur, largeur) sont récupérés dans le *header* du fichier BMP, ainsi on calcul le nombre total de pixel utilisables que l'on multiplie par 3 (nombre de caractères pouvant être caché dans un pixel) et on le divise par la taille de la chaîne de caractères. La fonction renvoie alors l'écart le plus approprié pour répartir uniformément les pixels modifiés dans l'image.

La procédure chargée de l'encodage lui-même est :

```
void code2(char *source, char *destination, int i, char *s)
```

Cette procédure prend en paramètres l'image source, l'image de destination, l'écart i et la chaîne de caractères à dissimuler.

Le fonctionnement est simple, on recopie le *header* dans le fichier de destination, on ajoute ensuite dans le premier pixel de l'image notre i de la manière suivante :

```
putc(i % 256, dst);
putc((i / 256) % 256, dst);
putc(i / (256 * 256), dst);
```

Un pixel entier est consacré au i car sa valeur peut être supérieur à 255 et un champ de couleur codé sur 8 bits ne suffit donc pas à le stocker. On parcours ensuite le fichier source en copiant les pixels inchangés ainsi que les pixels modifiés (contenant 3 caractères et situé tout les i pixels) dans le fichier de destination et ce jusqu'à ce qu'on arrive au bout de la chaîne de caractères passé en paramètre. Nous avons rencontré un petit problème sur la fin de la chaîne de caractères... En effet il n'est pas certain que le nombres de caractères de la chaîne soit un multiple de trois et donc que le dernier pixel modifié soit complet, il a donc fallu gérer les différents cas pour ne pas laisser un champ de pixel vide ce qui aurait eu pour conséquence de décaler tout les autres pixels et de calciner l'image d'une manière tout à fait atroce.

```
for (k = 0; k != 3; k++)
{
  if ((s[si]=='\0') && (k==0))
  {
    putc(p.blue, dst);
    putc(p.green, dst);
    putc(p.red, dst);
    break;
  }
  if ((s[si]=='\0') && (k==1))
  {
    putc(p.green, dst);
    putc(p.red, dst);
    break;
  }
  if ((s[si]=='\0') && (k==2))
  {
    putc(p.red, dst);
    break;
  }
  putc(s[si], dst);
  si++;
};
```

Voici la boucle qui va métamorphoser notre pixel en (petit) message secret ! On test à chaque fois si on arrive sur la fin de la chaîne, lorsque c'est le cas on va regarder le nombre de champs RGB laissés vide que l'on complétera par les valeurs d'origine. De cette manière notre pixel est complet et on peut continuer à parcourir le fichier sans risque de décalage de bits.

On pose ensuite la marque de fin de parcours i pixels après le dernier pixel modifié, il s'agit d'un pixel blanc comme pour la MS01.

On finit ensuite de parcourir le fichier jusqu'à la fin en le recopiant dans le fichier destination et on obtient ainsi notre image porteuse de précieuses informations !

3.2.2 Décodage

Et à présent... le décodage (une surprise me diriez vous) !

```
char *decode2(char* source)
```

Cette fonction prend pour unique paramètre le nom de l'image à décoder et renvoie la chaîne de caractères cachée dans l'image.

Le décodage de la MS02 est lui aussi plus rapide que celui de la MS01, en effet les caractères sont directement présent dans l'image, il n'y a donc pas besoin de faire de tests de parité sur les champs RGB et de reconstruire les caractères à l'aide d'une chaîne de bits.

Le principe de la fonction est le suivant : on saute le *header* de l'image, on arrive alors sur le premier pixel qui contient la valeur du i définissant la répartition des pixels dans l'image. On extrait cette valeur en réalisant l'opération inverse de celle exécutée lors de l'encodage :

```
i = getc(src);
i = i + getc(src) * 256;
i = i + getc(src) * (256 * 256);
```

On parcours ensuite le reste de l'image en extrayant correctement les caractères, et ce jusqu'à la rencontre du fameux pixel blanc. Les caractères extraits de l'image sont concaténés pour obtenir la chaîne d'origine qui sera renvoyée par la fonction decode2. Cette chaîne sera par la suite stockée dans un fichier pour offrir à l'utilisateur une manipulation libre de ses données.

3.3 Conclusion

Pour cette deuxième soutenance nous avons donc réalisé une nouvelle méthode de stéganographie et amélioré l'ancienne.

Lors de la première soutenance l'utilisateur devait rentrer à la main la chaîne de caractère qu'il souhaitait dissimuler dans l'image, maintenant il n'a plus qu'à entrer en paramètre le nom du fichier à encoder et le tour est joué (l'utilisateur intelligent évite les efforts inutiles). De même le décodage crée un fichier qui contiendra les données issues de l'image au lieu de simplement les afficher.

La nouvelle méthode de stéganographie possède une plus grosse capacité de données et une plus grande rapidité d'exécution que la première mais elle est aussi moins discrète, nous n'avons pas cherché à crée une méthode ultime mais juste offrir à l'utilisateur une autre alternative de stéganographie et ainsi une plus grande liberté d'utilisation. En fonction de ses besoins il pourra donc choisir l'une ou l'autre méthode.

Nous tenons à préciser que la Crypteam se décharge de toute responsabilité quant à une mauvaise utilisation du logiciel conduisant à une arrestation, une exécution sommaire ou encore un privage de dessert.

Pour la dernière soutenance nous allons tenter d'élaborer et d'implémenter à nouveau une technique de stéganographie (que l'on peut déjà appeler sans

trop s'avancer MS03) qui sera discrète comme la MS01 et qui possédera une capacité d'encodage équivalente à celle de la MS02 au détriment de la vitesse d'exécution (nul n'est parfait).

4 Conclusion

Cette préparation de soutenance fût donc encore une fois un expérience fort enrichissante. Nous avons rencontré plein d'obstacles mais le fait de les surpasser les uns après les autres nous a fait progresser !

Du coté de la cryptographie, nous avons donc appris le maniement d'un nouvel algorithme de chiffrement qui est le DES. Étant un algorithme de chiffrement symétrique cela nous à permis de nous diversifier par rapport au premier algorithme implémenté : le RSA qui est un algorithme de chiffrement asymétrique. De plus, on peut dire que ces algorithmes sont complémentaires. En effet, le DES a pour avantage sa rapidité tandis que le RSA a pour avantage sa sécurité. De plus comme il est expliqué dans la partie Cryptographie, un des principaux défauts du DES est contourné grâce au RSA : l'échange de l'unique clé de chiffrement et de déchiffrement DES est normalement complexe à effectuer de manière sûre mais si l'on chiffre la clé DES à l'aide du RSA nous n'avons plus ce problème !

Du coté de la stéganographie, on peut voir que nous avons aussi bien avancé, nous avons deux nouveaux algorithmes de stéganographie plus performants que le premier présenté en première soutenance. L'apparition d'un application graphique est aussi une bonne chose. Notre programme sera donc plus accessible et plus facile à utiliser par l'utilisateur *lambda*.

Pour la soutenance finale nous présenterons donc un dernier algorithme de chiffrement qui est l'AES. Comme le DES, c'est aussi un algorithme de chiffrement symétrique par contre étant plus récent que le DES, son niveau de sécurité est bien plus important. Nous aurons aussi bien sûr une application graphique finalisée et on ne verra plus que nous avons un programme de cryptographie et un programme de stéganographie car l'application graphique réunira tous les programmes nécessaires pour chiffrer un fichier à l'aide de l'algorithme choisi et le cacher dans l'image sélectionnée.