

# Cryptographie et jeu de cartes



M1 STIC  
Codage et cryptographie  
Rapport de projet

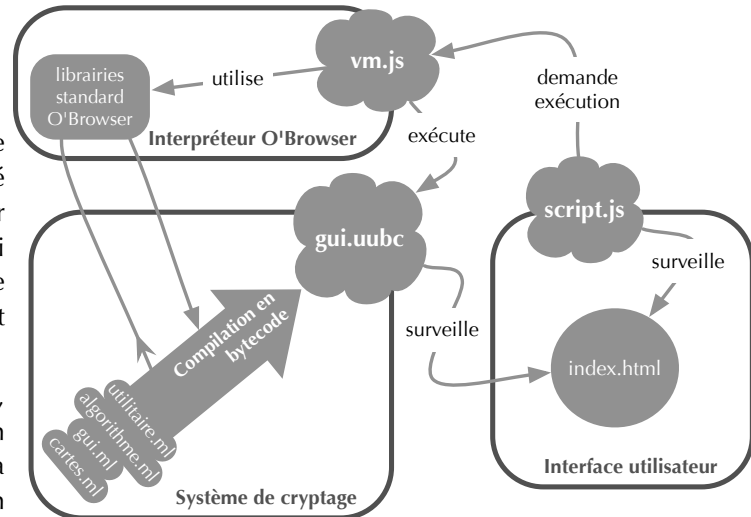
Janvier 2010  
Jonathan Aceituno  
Paul Albert

## Introduction

Pour implémenter la méthode de cryptage proposée par le sujet, nous avons choisi l'originalité. En effet, nous avons décidé d'utiliser le langage **Objective Caml** à travers un interpréteur Javascript de *bytecode*, nommé **O'Browser**, nous permettant ainsi de pouvoir faire une jolie interface à notre programme sous forme de page HTML, saupoudrée de graphismes magiques et d'animations époustouflantes.

Afin de bien comprendre l'architecture de notre réalisation, présentée ci-contre, il faut savoir que le programme est divisé en plusieurs unités : le code Objective Caml renferme toute la logique du système de cryptage utilisant une représentation virtuelle des cartes, tandis que le code Javascript sert à donner vie à la page HTML et permet de lier les requêtes de l'utilisateur à la logique du système. Il faut bien comprendre que les instructions OCaml ont été compilées en bytecode puis sont exécutées par l'interpréteur de bytecode OCaml qu'est O'Browser. Ce dernier est réalisé en Javascript et fonctionne sur **Firefox 3.5**, et plus précisément grâce à son moteur Javascript nommé **Spidermonkey**.

Utiliser un tel interpréteur écrit dans un langage de script pose cela dit un problème assez ennuyeux : le déroulement du programme peut devenir très lent, ce qui est assez gênant, surtout pour les opérations coûteuses telles que, par exemple, le cryptage d'une image.



## 1 Méthode de cryptage

La méthode de cryptage décrite dans le sujet est celle du **chiffre de Vernam**. Nous avons choisi de laisser les cartes dans un ordre assez conventionnel par défaut, triées par couleurs et par valeurs, mais l'utilisateur peut les réorganiser afin qu'il corresponde à celui de son acolyte. Le jeu de 54 cartes est mis dans un désordre connu. S'ensuit la phase de mélange des cartes selon B. Schneier. Le joker noir recule d'une position, puis le joker rouge, lui, recule de deux positions. On fait ensuite une double coupe par rapport aux jokers. Toutes les cartes au dessus du joker le plus haut sont interverties avec celles situées sous l'autre joker. Une autre coupe est alors effectuée : on prend les  $n$  cartes au dessus du paquet et on les intercale entre la dernière carte et le reste du paquet ( $n$  étant la valeur de cette dernière carte). On regarde alors la première carte, disons qu'elle a pour valeur  $m$ , et l'on va alors prendre la  $m+1^{\text{e}}$  carte. Si cette carte est un joker, on retourne à la première étape de l'élaboration d'une valeur de la clé, et si tel n'est pas le cas on ajoute une lettre à la clé dont l'indice est la valeur de cette carte modulo 63 pour bien tomber sur un indice de lettre du dictionnaire *base64*. On répète le procédé autant de fois que l'on désire de lettres dans notre clé. Ce processus ne laissant pas de hasard possible, l'interlocuteur qui connaîtra l'ordre initial des cartes parviendra à générer le même flux. Par rapport au sujet, nous avons agrandi l'alphabet de 26 caractères à 63, ceci nous permettant d'utiliser l'algorithme *base64* qui permet la transformation de n'importe quel flux de données en caractères compris dans une liste de 63 caractères prédéfinis, ces données à transformer pouvant être une image, par exemple. Ainsi notre système est capable de crypter n'importe quelle information ou fichier puisqu'un flux de données quelconque peut être converti en *base64* et l'opération inverse est aussi possible.

Une fois le procédé de génération d'un flux de caractères établi, il suffit de générer une clé ayant la même longueur que le message à coder puis additionner ou soustraire terme à terme les valeurs numériques correspondant aux caractères (tout en les restreignant à la limite de 63), pour coder ou décoder un message.



## 2 Pertinence de l'approche fonctionnelle

Les fonctions de calcul de la clé utilisant les fonctions de mélange des cartes, et ces dernières utilisant elles-mêmes des fonctions atomiques de déplacement et de traitement des cartes, qui, elles encore, utilisent des fonctions plus élémentaires de traitement des nombres, des caractères et des listes (pour prendre comme métaphore la structure de la matière), nous avons pu apprécier l'utilité d'un langage fonctionnel fortement typé comme **Objective Caml**. La concision du style fonctionnel permet d'obtenir un code très clair et sans ambiguïté, donc moins sujet à générer des erreurs. Nous trouvons ce point important, surtout lorsqu'il s'agit de développer des applications ayant trait à la sécurité. Si ce style est éloigné des problèmes de bas niveau comme la gestion de la mémoire, les programmes **Objective Caml** sont très optimisés et aussi rapides que des équivalents C++.

Malheureusement, nous ne pouvons pas faire valoir cet argument pour ce projet si spécial : l'implémentation d'**Objective Caml** utilisée ici n'est pas officielle, c'est **O'Browser**, un interpréteur de bytecode déjà optimisé, codé en Javascript. Ce détail enlève toute valeur aux optimisations du code : Javascript étant déjà lui-même un langage interprété, dont le but n'est pas la vitesse, les programmes résultants ne brillent pas par leur vitesse d'exécution. Cependant nous avons choisi de suivre cette voie car l'utilisation d'un navigateur web nous offrait la possibilité de créer une interface saisissante de manière très simple grâce à la librairie Scriptaculous, tandis que le pouvoir d'un langage fonctionnel était au service de notre algorithme. Le fait qu'il existe peu de librairies d'interfaces graphiques pour OCaml a affirmé notre choix.

## 3 Détail de la réalisation

### utilitaire.ml

Le module Utilitaire contient toutes les fonctions-outil utiles qui ne sont pas présentes dans la (pauvre) librairie standard d'OCaml. Parmi elles des fonctions de conversion, de génération et de modification de listes, et une implémentation du modulo qui fonctionne pour les nombres négatifs.

### algorithme.ml

Le module Algorithme implémente dans des fonctions toutes les étapes du procédé de génération de clé, ainsi que les fonctions de codage et de décodage d'un message (ou d'une image). L'écriture des étapes est très facile car il s'agit d'appeler des fonctions sur des tas de cartes : il en résulte une programmation très naturelle.

### index.html

La page XHTML est décrite dans ce fichier. Il s'agit d'une description uniquement structurelle de la page, car nous respectons le principe de séparation du contenu (index.html), de la présentation (style.css) et du comportement (script.js).

## Bilan

Malgré la lenteur de l'interpréteur O'Browser, projet de thèse de doctorat hélas peu mature, utiliser OCaml est bel et bien un atout pour implémenter un tel problème logique, il facilite grandement la programmation. L'utilisation conjointe de ce langage et des technologies XHTML, CSS et Javascript a permis de rendre vivante l'interface graphique tout en gardant un programme stable.

### cartes.ml

Le module Cartes contient les fonctions et les structures de données nécessaires au traitement des cartes. On définit donc un type carte donnant toutes les possibilités, et un tas de cartes est une simple liste. On trouvera des fonctions de conversion pour passer d'une carte à sa valeur au bridge, de génération d'un jeu trié, de déplacement ou de tirage de cartes et de coupe d'un paquet.

### gui.ml

Le module Gui fait le lien entre l'interface graphique et le système de cryptage en rajoutant dynamiquement du HTML dans le DOM de la page qui fait l'interface, et en associant des fonctions OCaml à des événements sur des objets DOM de la page, en prenant soin de récupérer les valeurs voulues au moment opportun à partir de la page.

### script.js

Ce script est chargé lorsque la page HTML est ouverte : il associe des événements sur la page à des actions Javascript de sorte qu'elle puisse répondre aux demandes de l'utilisateur et jouer le rôle d'interface. Par le biais d'un événement, le code OCaml est exécuté par vm.js (la machine virtuelle O'Browser) à la demande de script.js. Le cryptage/décryptage d'image utilise la fonction *base64* et des fonctions DOM propres à Firefox 3.5. Les effets graphiques sont à la charge de la librairie scriptaculous.js

### POUR TESTER LE PROJET

Lancer index.html avec Firefox 3.5 minimum.

### POUR COMPILER LE PROJET

Au cas où quelque chose ne va pas, il faut recompiler. Ayez installé **ocaml** et **sharutils**. Dans le répertoire du projet :

```
cd lib/caml; make clean; make  
cd ../..; make clean; make
```