Cryptographie symétrique

Projet avancé automne 2024

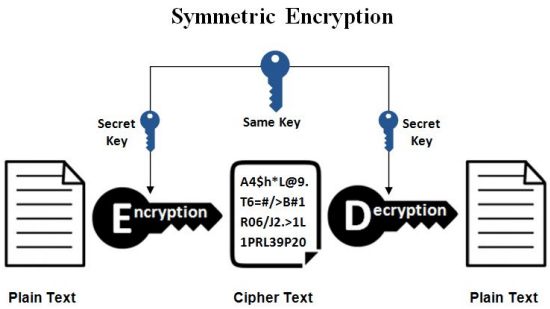


Table des matières

1 Introduction

La cryptographie consiste à remplacer un message lisible par tout le monde, le message en clair, par un message inintelligible (le chiffré) sauf pour ceux qui connaissent le "truc" qui permet de retrouver le message en clair.

Le "truc" consiste à utiliser une clef qui permet de transformer le message en clair en message codé (opération de chiffrage) et vice-verça (opération de déchiffrage). Si la clef de chiffrage et de déchiffrage est la même on parle de chiffrage symétrique [1]. La clef doit rester secrète et partagées seulement entre les personnes voulues.

Chercher à déchiffrer un message codé, donc chiffré, sans en avoir la clef est le décryptage. On parle de "casser le code", c'est à dire trouver la clef. On emploie aussi le terme anglais crack (to crack the code, cracker le code).

Le projet traite de cryptographie symétrique et est divisé en trois parties indépendantes mais complémentaires.

— Indépendantes veut dire qu'on peut programmer chaque partie sans avoir rien fait concernant les autres (ou presque).

— Complémentaires veut dire que les trois parties participent à la construction d'un système complet de génération de clef, chiffrage/déchiffrage et de décryptage (restreint).

Dans la suite on présente le problème et le travail à réaliser pour chacune des parties.

1. Il existe des chiffrages asymétriques.

2 Partie 1 : chiffrer/déchiffrer un message avec une clef symétrique

Cette partie s'occupe de chiffrer et déchiffrer des messages. On va utiliser deux méthodes différentes. Le message en clair (à chiffrer) sera lu dans un fichier texte, et le message chiffré (le chiffré) sera écrit dans un autre fichier texte.

2.1 Méthode 1 : Chiffrage caractère par caractère avec l'opérateur *xor*

La façon la plus simple (et la moins robuste) pour chiffrer un message est d'utiliser l'opérateur *xor* de ou exclusif(ˆen C). L'opérateur va faire le xor bit à bit de ses deux opérandes. Les opérandes ici sont des octets.

Remarque : il est préférable de raisonner en terme d'octets plutôt que de caractères. En effet ce procédé de chiffrage peut s'appliquer sur n'importe quel fichier de données (image, son, exécutable, etc) en traitant le fichier octet par octet.

Principe de la méthode : Soit *msg* un message (unsigned char *) [2] et *key* une chaîne de caractères de même type. Le principe est d'écrire la clef sous le message et de faire le *xor* des octets en colonnes. Si la clef est plus courte que le message on répète la clef autant de fois que nécessaire. Si la clef est plus longue que le message, on la tronque à la longueur du message. Ensuite on fait le xor sur les codes UTF8 des caractères (en C on peut faire directement $c1\hat{}c2$ avec $c1$ et $c2$ de type *unsigned char*).

Pour chiffrer on calcule donc $C = A \oplus B$. Le résultat C est le chiffré de A. Le déchiffrement s'effectue en combinant le chiffré C avec la clé B par la même opération : $C \oplus B$. Il se trouve qu'elle fait retrouver le clair A.

L'application de l'opération XOR étant simple en informatique, ces traitements peuvent s'effectuer à très grande vitesse.

Exemple : msg = "Les carottes sont cuites", key = "rutabaga"

Chiffrage :

L	e	s		c	a	r	o	t	t	e	s		s	o	n	t		c	u	i	t	e	s
r	u	t	a	b	a	g	a	r	u	t	a	b	a	g	a	r	u	t	a	b	a	g	a
3e	10	7	41	1	0	15	e	6	1	11	12	42	12	8	f	6	55	17	14	b	15	2	12

Le résultat du chiffrage n'est pas obligatoirement formé de caractères affichables, c'est pourquoi il est ici écrit en hexadécimal.

Le déchiffrage consiste simplement à faire la même opération en partant du message chiffré avec la même clef.

2.1.1 Travail à réaliser

1. Ecrire une fonction implémentant le chiffrage XOR sur les messages et des clefs. La chaînes de caractères à chiffrer et la clef seront passés en paramètre de la fonction. La même fonction permet le chiffrement et le déchiffrement.

2. En C les chaînes de caractères sont des tableaux dynamiques.

2. Codez la fonction *gen_key* de génération aléatoire d'une clef alphanumérique [3]. La longueur sera passée en paramètre.
3. Créer des programmes principaux de test et valider le code écrit.

2.2 Extension : Principe du chiffrage par masque jetable

Le chiffrement par la méthode du masque jetable consiste à combiner le message en clair avec une clé présentant les caractéristiques très particulières suivantes :

— La clé doit être une suite de caractères au moins aussi longue que le message à chiffrer.
— Les caractères composant la clé doivent être choisis de façon totalement aléatoire.
— Chaque clé, ou « masque », ne doit être utilisée qu'une seule fois (d'où le nom de masque jetable).

L'intérêt considérable de cette méthode de chiffrement est que si les trois règles ci-dessus sont respectées strictement, le système offre une sécurité théorique absolue, comme l'a prouvé Claude Shannon en 1942.

L'argument théorique est le suivant, dans son principe : si on ne connaît que le texte chiffré et que toutes les clés sont équiprobables, alors tous les textes clairs de cette longueur sont possibles et avec la même probabilité puisqu'il y a bijection, une fois le chiffré fixé, entre clés et textes clairs. Connaissant le texte chiffré, il n'y a aucun moyen de distinguer parmi ceux-ci le texte clair original qui correspond. Une analyse statistique est vaine. La connaissance d'une partie du texte clair et du chiffré correspondant donnent la partie de la clé utilisée, mais ne donnent aucune information supplémentaire : le reste de la clé est indépendant de la partie connue, la clé n'est pas réutilisée.

Ce type d'impossibilité, appelé sécurité sémantique, ne repose pas sur la difficulté du calcul, comme c'est le cas avec les autres systèmes de chiffrement en usage. Autrement dit, le système du masque jetable est inconditionnellement sûr.

Un chiffrement à la main par la méthode du masque jetable fut notamment utilisé par Che Guevara pour communiquer avec Fidel Castro. C'est aussi le principe du téléphone rouge entre les USA et l'URSS pendant la guerre froide (par telex).

2.2.1 Problème de l'utilisation unique de chaque clé

Le risque que fait courir la réutilisation d'une clé est facile à montrer.

Soit un message M1 masqué grâce à la clé K, nous obtenons le chiffré C1. Supposons qu'un autre message M2 soit chiffré avec le même masque K, fournissant le chiffré C2.

Nous avons les relations suivantes : $C1 = M1 \oplus K$ et $C2 = M2 \oplus K$

Supposons qu'un adversaire applique l'opération XOR aux deux chiffrés C1 et C2, et réutilisons les propriétés vues ci-dessus :

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = (M_1 \oplus M_2) \oplus (K \oplus K) = M_1 \oplus M_2$$

On obtient le XOR des deux messages en clair. C'est très dangereux, car tout effet de masque de la clé K a disparu.

3. Chaque caractère de la clef est généré aléatoirement en tant qu'un entier sur 8 bits.

Si par exemple un adversaire connaît les deux messages chiffrés et l'un des messages en clair, il peut trouver instantanément le deuxième message en clair par le calcul :

$C_1 \oplus C_2 \oplus M_1 = M_2$ (1)

En fait, même sans connaître l'un des clairs, des méthodes plus complexes permettent souvent de retrouver $M1$ et $M2$.

2.2.2 Travail à réaliser :

1. A partir du code précédent, implémenter le masque jetable avec une clef aléatoire. Elle pourra être sauvée dans un fichier.
2. Valider le code.

2.3 Méthode 2 : Chiffrage par blocs : Cypher Block Chaining (CBC)

La méthode précédente est trop facile à cracker car elle induit trop de régularités dans le chiffré. Nous allons utiliser une autre méthode basée elle aussi sur l'opérateur xor.

A lieu de chiffrer les caractères (ou octets) du message un par un, on va traiter plusieurs octets à la fois. Le groupe d'octets traité à chaque étape s'appelle un bloc. Le nombre d'octets constituant un bloc est fixe et choisi au départ. Chaque bloc donnera un bloc chiffré de même taille.

Le message est donc divisé en une suite $B_1, B_2, \ldots, B_n$ de blocs de même taille. Le dernier bloc B_n peut être éventuellement complété avec des caractères espace si besoin (padding). On va donc chiffrer les blocs dans l'ordre B_1, jusqu'à B_n pour produire une suite de blocs chiffrés $C_1, ..., C_n$.

La méthode CBC consiste, pour chiffrer le bloc B_i, à :

1. Faire un *xor* entre le bloc chiffré C_{i-1} et le bloc clair B_i pour produire le bloc intermédiaire CC_i,
2. Chiffrer CC_i avec la méthode de chiffrement choisie, $\mathcal{E}_K$ où K est la clef, pour obtenir C_i.

Pour commencer le processus, c'est à dire chiffrer le premier bloc pour lequel il n'y a pas de chiffré d'ordre précédent, on utilise un *vecteur d'initialisation* (VI) qui est un bloc arbitraire choisi au départ.

$C_1 = \mathcal{E}_K(B_1 \oplus VI)$
$C_i = \mathcal{E}_K(B_i \oplus C_{i-1})$
où $\oplus$ symbolise l'opérateur *xor*.

Le déchiffrement se fait en renversant le processus :

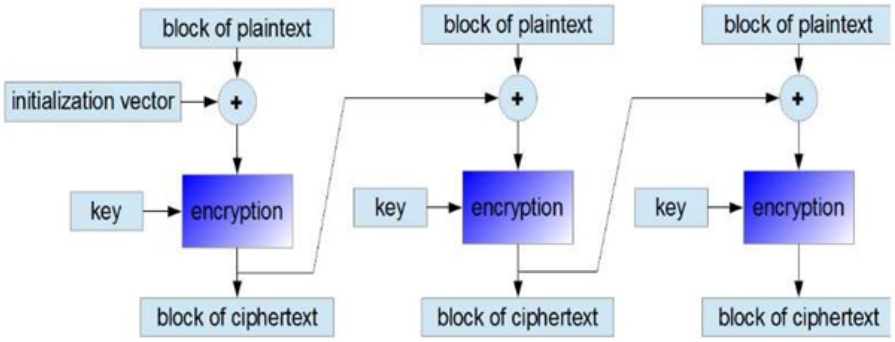
$B_1 = VI \oplus \mathcal{D}_K(C_1)$
$B_i = C_{i-1} \oplus \mathcal{D}_K(C_i)$

où $\mathcal{D}_K$ est la fonction de déchiffrement indicée par K, la clef.

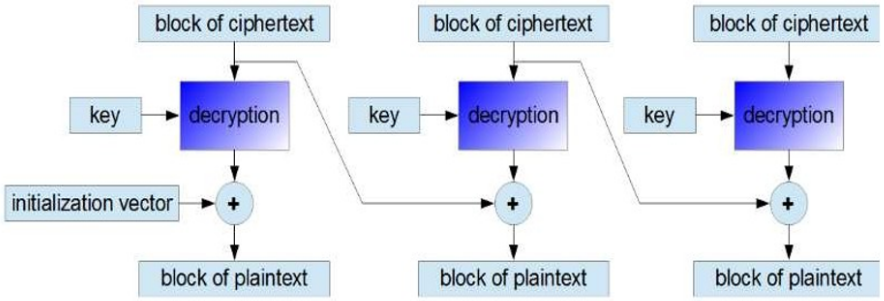
2.3.1 Travail à réaliser en C et main de synthèse

1. Implémenter la méthode CBC avec des blocs de 16 octets, un vecteur d'initialisation passé en paramètre qui devra faire lui aussi 16 octets (caractères [4]), et comme méthode

4. Il vaut mieux ici prendre des caractères non accentués pour avoir un octet par caractère en codage UTF8.



Cryptage en mode CBC



Décryptage en mode CBC

de chiffrement/déchiffrement un simple *xor*, c'est à dire que $\mathcal{E}_K = \mathcal{D}_K = xor$[5] avec une clé char * (ou byte *), c'est à dire un type chaîne de caractères en C[6]. Les méthodes de chiffrement et de déchiffrement étant différentes dans leurs protocoles, l'implémentation consistera en deux fonctions : une pour chiffrer et une pour déchiffrer.

2. Créer les fonctions *cbc-crypt* et *cbc-uncrypt*, implémentant la méthode 2 (avec xor). Les paramètres sont les noms des fichiers d'entrée et de sortie, la clef et le vecteur d'initialisation.
   - Ecrire un programme principal *sym_crypt* permettant d'appeler toutes les fonctions programmées jusqu'ici. Le message d'entrée (le clair) sera lu dans un fichier et le résultat sera écrit dans un autre fichier. Les noms de fichier et la clef seront passés en paramètre de la commande. On utilisera *getopt.h* pour gérer les paramètres de la ligne de commande :
     - -i suivie du nom du fichier contenant le message à chiffrer (présence obligatoire)
     - -o suivie du nom du fichier où l'on va écrire le chiffré (présence obligatoire)
     - -k suivie de la clef (présence obligatoire) ou -f suivi du nom de fichier contenant la clef.
     - -m suivie du nom de la méthode de chiffrement ou déchiffrement souhaitée parmi : xor, cbc-crypt, cbc-uncrypt, mask
     - -v suivie du nom du fichier contenant le vecteur d'initialisation (présence obligatoire

5. C'est à dire $\mathcal{E}_K(m) = m$ xor K, et même chose pour $\mathcal{D}_K$.
6. Les chaînes de caractères en C sont des tableaux dynamiques de caractères.

si la méthode est cbc crypt ou uncrypt)
  - -l suivie du nom du fichier de log. Présence facultative, auquel cas le fichier sera stdin.
  - -h affichant l'aide des commandes. Cette option, si elle est présente, annule toutes les autres.
- Il y a sur Moodle des scripts de test pour vérifier le bon fonctionnement de vos programmes. Ces tests sont minimums. Vous pouvez les étendre pour tester les fonctions qui ne sont pas prises en compte dans les versions fournies.

Exemple :

```
sym_crypt -i clair.txt -o crypt.txt -k taratata -m cbc-crypt -v iv.txt
sym_crypt -o clair.txt -i crypt.txt -f key.txt -m xor -l log.txt
sym_crypt -i clair.txt -o crypt.txt -f key.txt -m mask
```

3 Partie 2 : Le problème de l'échange de la clef secrète : échange de Diffie-Hellman

La cryptographie par clef symétrique est simple mais demande que les agents communiquant utilisent la même clef. Ils ont pu se mettre d'accord au préalable, en se rencontrant par exemple. Mais s'ils sont à distance et ne peuvent se mettre d'accord sur la clef sans être espionnés, comment faire ?

Whitfield Diffie et Martin Hellman, ont imaginé une méthode, publiée en 1976, par laquelle deux agents, nommés par convention Alice et Bob, peuvent se mettre d'accord sur un nombre (qu'ils peuvent utiliser comme clé pour chiffrer une conversation ou un message) sans qu'un troisième agent, appelé Ève puisse découvrir ce nombre, même en ayant écouté tous leurs échanges. Cette idée valut en 2015 aux deux auteurs le prix Turing (dixit Wikipédia).

Pour implémenter cette méthode, il nous faut un corps fini et un nombre premier p. Par exemple Z/pZ, l'ensemble des nombres relatifs $\{-(p-1), \ldots, -1, 0, 1, \ldots, p-1\}$

3.1 Principe d'un échange de clés Diffie-Hellman dans Z/pZ avec p premier

Remarque : La notation $[p]$ signifie *modulo p*

1. Alice et Bob ont choisi un groupe fini (Z/pZ) et un générateur g de ce groupe. **En clair ils choisissent p premier, calculent g et utilisent les nombres de 0 à $p-1$. Tous les calculs sont modulo p.**
2. Alice choisit un nombre a au hasard, $(1 \leq a \leq p-1)$ élève g à la puissance a, et communique à Bob ce nombre $A = g^a$ (calculé modulo p), comme montré dans l'exemple ci-dessous), c'est-à-dire le nombre A.
3. Bob fait de même avec le nombre b choisi au hasard $(1 \leq b \leq p-1)$. Il transmet à Alice le nombre $B = g^b[p]$;
4. Alice, en élevant le nombre B reçu de Bob à la puissance a, obtient $(g^b)^a[p]$.
5. Bob fait le calcul analogue avec le A reçu d'Alice et obtient $(g^a)^b$, qui est le même résultat que celui obtenu par Alice en 4., soit la clé commune aux deux.
6. L'astuce est qu'il est difficile d'inverser l'exponentiation dans un corps fini, c'est-à-dire de calculer le logarithme discret (trouver a tel que $h = g^a$ connaissant h et g), Ève ne peut découvrir ni a ni b, donc ne peut pas calculer $(g^a)^b[p]$ (à condition que le nombre p soit choisi assez grand). Finalement, Alice et Bob connaissent donc tous les deux le nombre $(g^a)^b[p]$ dont Ève n'a pas connaissance.

Exemple :

1. Alice et Bob ont choisi un nombre premier p et une base g. Par exemple, $p = 23$ et $g = 5$
2. Alice choisit un nombre secret $a = 6$
3. Elle envoie à Bob la valeur $A = g^a[p] = 5^6[23] = 15625[23] = 8$
4. Bob choisit à son tour un nombre secret $b = 15$
5. Bob envoie à Alice la valeur $B = g^b[p] = 5^{15}[23] = 19$
6. Alice peut maintenant calculer la clé secrète : $B^a[p] = 19^6[23] = 2$
7. Bob fait de même et obtient la même clé qu'Alice : $A^b[p] = 8^{15}[23] = 2$

Remarque importante : Dans la pratique, on prend un nombre premier p dit de Sophie Germain (tel que $q = 2p + 1$ premier lui aussi) de grande taille et un générateur g dans Z/pZ (g est donc premier avec $p - 1$). Cela évite de choisir un nombre $p = 2q + 1$ avec q non premier et décomposable en une suite de facteurs. On a en effet des méthodes pour calculer p à partir des facteurs et de réduire ainsi la puissance de calcul nécessaire.

3.2 Travail à réaliser en C et Python

On va utiliser un programme C et un programme Python. Le programme C va faire les calculs pour initialiser l'échange. Le programme Python fera la communication et le calcul de la clef.

1. Ecrire un programme C *dh_gen_group* permettant de générer un nombre premier de Sophie Germain, un générateur et admettant les options :
   — -o suivie d'un nom de fichier où les résultats suivants seront écrits : p le nombre premier généré et g le générateur,
   — -h donnant la syntaxe d'exécution sans autre action.

   Principe des calculs :
   — Calcul du nombre premier p (de Sophie Germain) avec la fonction :
   *long genPrimeSophieGermain(long min,long max,int *cpt)*
   — Calcul du générateur : *long seek_generator(long start,long p)*

   Remarque : Les fonctions cités sont données dans le fichier *dh_prime.c* sur Moodle. Des commentaires Doxygen sont suffisants pour en comprendre l'usage et l'action.

   Remarque : Normalement nous devrions faire les calculs en utilisant une bibliothèque permettant de faire des calculs en précision illimitée comme *GMP*. Pour simplifier ce projet nous nous limiterons à les *long*. Donc attention aux dépassements de capacité en C.
2. Ecrire un programme Python (*dh_genkey.py*) utilisant les packages Queue et Thread avec deux fonctions *alice* et *bob*, choisissant respectivement les paramètres a et b et calculant la clef. Le programme lira les paramètres dans le fichier précédemment généré. On utilisera la fonction Python :

   ```
   puissance_mod_n (a:int,  e:int,  n:int)->int
   ```

   calculant la puissance modulaire, $a^e[n]$

   La simulation devra préciser ce qui est connu d'Alice seulement, de Bob seulement, des deux et de Eve (l'espion). On affichera explicitement ce qui est transmis sur le réseau et ce qui est calculé en privé.

Exemple :

```
dh_gen_group -o param.txt

python3 dh_genkey.py -i param.txt -o key.txt
```

3.3 Séquence culturelle (ou apocalyptique suivant les avis) : Fondement mathématique

La méthode utilise la notion de groupe (multiplicatif), par exemple celui des entiers modulo p, où p est un nombre premier : dans ce cas, les opérations mathématiques (multiplication, puissance, division) sont utilisées telles quelles, mais le résultat doit être divisé par p pour ne garder que le reste, appelé modulo. Les groupes ayant la propriété de l'associativité, l'égalité $(g^b)^a = (g^a)^b$ est valide et les deux parties obtiennent bel et bien la même clé secrète.

La sécurité de ce protocole réside dans la difficulté du problème du logarithme discret : pour que Ève retrouve $(g^a)^b$ à partir de g^a et g^b, elle doit élever l'un ou l'autre à la puissance b ou à la puissance a respectivement. Mais déduire a (resp. b) de g^a (resp. g^b) est un problème que l'on ne sait pas résoudre efficacement. Ève est donc dans l'impossibilité (calculatoire) de déduire des seules informations g^a, g^b, g et p, la valeur de $(g^a)^b$.

Il faut toutefois que le groupe de départ soit bien choisi et que les nombres utilisés soient suffisamment grands pour éviter une attaque par recherche exhaustive. À l'heure actuelle, un nombre premier p de l'ordre de 300 chiffres ainsi que a et b de l'ordre de 100 chiffres sont tout simplement impossibles à casser [7] même avec les meilleurs algorithmes de résolution du logarithme discret.

4 Partie 3 : Cracker le code

Bon, tout ça est très rigolo, mais ce que tout le monde attend, c'est de pouvoir cracker un code. Nous y sommes. Vu que cela n'est pas si simple nous allons nous limiter au cas où le message a été chiffré à l'aide d'une méthode simple, c'est à dire le *xor* sur chaque caractère (Partie 1, méthode 1).

4.1 Attaque sur le message chiffré

Le processus par lequel on tente de décoder un message est appelé une attaque. Le chiffrage et le déchiffrage d'un fichier peut se faire sans se préoccuper de la nature des octets. Par contre, pour casser un chiffre nous devons connaître la langue dans laquelle le message a été écrit, ainsi que le codage informatique de ses caractères.

Le codage informatique utilisé sur la plupart des ordinateurs récents est le UTF-8. C'est un codage qui utilise entre un et quatre octets pour coder un caractère et qui est prévu pour représenter tous les caractères des langues de la planète. Il reprend le codage ASCII sur sept bits mais pour les caractères européens (et donc les caractères accentués du français), il utilise deux octets dont le premier (C3 en hexadécimal) sert de balise.

Du coup il faut faire un choix entre différentes options :

— Soit nous considérons que le texte en français est codé selon la norme Latin-1 sur un octet et non en UTF-8. Il faudra donc convertir ou enregistrer les textes à cette norme, ce qui est prévu dans la plupart des éditeurs de texte.

— Soit nous considérons que le texte est écrit en français sans accent ou en anglais, en UTF8. Pas d'accent, pas de problème.

Dans la suite nous allons considérer que nous décryptons un texte écrit en anglais ou en français sans accent et en UTF8 [8] car cela simplifie énormément les choses. Si vous voulez traiter le cas de texte écrits en français et codés selon la norme Latin-1, contactez votre tuteur.

7. Pour l'instant en 2024.

8. Le codage des caractères de la norme ASCII est le même dans la norme UTF8. On parlera d'ASCII dans la suite.

Nous présentons dans cette sections trois types d'attaque : C1, C2, C3. Nous allons les utiliser de manière hiérarchique C1, puis C2, puis C3. Ces attaques partent du principe que le message chiffré est un texte écrit en anglais. Pour toutes ces attaques, on va faire une liste des clefs possibles, dites clefs candidates, et chercher par différents moyens à éliminer les clefs impossibles. En dernier ressort, il faudra essayer chacune des clefs restantes pour décrypter le chiffré.

Remarque importante : Toujours dans un souci de simplification, nous allons considérer que la clef est une chaîne de caractères composée uniquement de caractères numériques ('0' à '9') et de caractères alphabétiques minuscules ou majuscules (de 'a' à 'z' et de 'A' à 'Z'). Donc, pas de ponctuation ou caractères spéciaux pour la clef.

4.1.1 Crack C1 : émondage des caractères possibles pour la clef

L'ensemble des lettres de l'anglais/français est représenté par un sous ensemble des octets codés sur 8 bits. Une première attaque qui permet de trouver un ensemble de clés valides, consiste à vérifier que les caractères décodés par une clé candidate sont valides dans la langue et le codage choisis. En clair, si x est un caractère du chiffré, et y un caractère de la clef (inconnue) qui a peut-être été utilisé pour chiffrer le texte alors $z = x$ xor y doit redonner le caractère d'origine, c'est à dire que le code de z doit correspondre à un caractère ASCII admissible, c'est à dire, un caractère de la langue, un chiffre ou un caractère de ponctuation.

On ne validera pas d'une manière globale chaque clef candidate, ce serait trop long. On validera l'un après l'autre chaque caractère valide des clefs. Précisément si la clef a pour longueur 3, le caractère clef[0] sert à coder les caractères $1, 4, ..., 1 + 3k, ...$ du message d'origine. On détermine pour commencer les valeurs possibles de clef[0] puis, de manière tout à fait indépendante, on déterminera les valeurs possibles de clef[1] et de clef[2].

On va considérer tous les caractères possibles pour chaque clef[i], $0 \leq i <$ longueur de la clé. Par exemple, si le chiffré est msg = "s(/1& !" et que la clef a trois caractères, alors clef[0] a servi à chiffrer msg[0] et msg[3]. Pour chaque valeur c candidate pour clef[0] on fait : msg[0] xor c = 's' xor c qui doit donner une valeur admissible de la table ASCII. De même, msg[3] xor c = '1' xor c qui doit aussi donner une valeur admissible de la table ASCII.

On va mémoriser l'ensemble des caractères admissibles pour chaque caractère de la clef. De là on peut établir la liste des clefs candidates en faisant une sorte de produit cartésien des listes de caractères.

Par exemple, si on obtient :
- $clef[0] \in ['5', '6', '8']$
- $clef[1] \in ['1']$
- $clef[2] \in ['0', '2']$
- alors on aura aisément la liste des $3 * 1 * 2 = 6$ clefs selon C1 :

$$clef \in ["510", "512", "610", "610", "810", "812"]$$

.

Les caractères valides pour la clef peuvent être stockés dans un tableau statique ou dynamique à deux dimensions *clef[tailleClef][maxCaracteres]* ou *tailleClef* est la taille de la clef recherchée, et *maxCaracteres* le nombre de caractères possible pour un caractères de la clef. Comme, dans cette partie, nous nous limitons à des clefs formées de caractères alphanumériques, *maxCaractères* est égal à 10+26+26 = 62 (caractères de 0 à 9, de a à z et de A à Z).

Au départ, chaque ligne du tableau (chaque caractère de la clef) contient tous les caractères possibles. Au fur et à mesure des tests, les caractères impossibles sont supprimés.

Attention : Il est important que le chiffré ne code que des caractères alphanumérique, espace ou ponctuation pour que cet émondage fonctionne. En effet si un caractère du chiffré vient du chiffrement d'un caractère de contrôle, l'émondage va éliminer tous les caractères potentiels de la clef pour ce caractère.

4.1.2 Crack C2 : statistiques des lettres dans la langue cible

Dans notre chiffrement caractère par caractère, une même lettre risque d'être chiffrée par le même nombre. L'analyse fréquentielle examine les répétitions des codes du message chiffré afin de trouver la clef. Elle est principalement utilisée contre les chiffrements mono-alphabétiques (substitution d'une lettre par une autre) qui présentent un biais statistique. Elle est moins pertinente pour un chiffrage par clef mais reste utile.

L'analyse fréquentielle est basée sur le fait que, dans chaque langue, certaines lettres ou combinaisons de lettres, apparaissent avec une certaine fréquence. Par exemple, en français, le e est la lettre la plus utilisée, suivie du a et du s. Inversement, le w est peu usité.

Le lien ci-dessous donne les fréquences pour diverses langues. On utilisera la colonne correspondant à la langue choisie.

`https://en.wikipedia.org/wiki/Letter_frequency`

Le tableau peut être téléchargé sur Moodle pour l'anglais et le français.

Dans ce tableau, les diverses formes des lettres sont prises en compte : par exemple pour le français et pour la lettre 'E' les formes 'e','E','é','è' et 'ê' sont prises en compte. Cet aspect est sans objet pour les textes en langue anglaise.

Avec chaque clef candidate obtenue par la méthode C1, on décode le cryptogramme et on calcule par comptage les diverses fréquences d'apparitions des lettres du texte décodé (mais pas forcément déchiffré) avec la clef. On obtient alors un tableau de fréquences noté $Freq$. La meilleure clef est celle pour laquelle les fréquences mesurées sont les plus proches des fréquences théoriques dans la langue cible : $Freq\_th$. Cette notion de proximité et donc de distance entre deux tableaux sera mesurée par la fonction d définie par :

$$d(Freq\_th, Freq) = \sum_{i=1}^{26} (Freq\_th[i] - Freq[i])^2$$

Cette répartition des fréquences des lettres n'est qu'approximative, cela dépend de nombreux paramètres tels que le niveau de langue du texte, ainsi que du style d'écriture. Une deuxième condition nécessaire pour appliquer cette technique est la longueur du cryptogramme. En effet, un texte trop court ne reflète pas obligatoirement la répartition générale des fréquences des lettres.

Quelquefois, on peut constater que la clef sélectionnée par cette méthode n'est pas la bonne. Cela provient du manque de représentativité du texte : trop petite taille, style trop particulier. On s'oriente alors vers une méthode utilisant un dictionnaire.

On cherche la clef avec le meilleur score. Pour cela, à partir des clefs valides pour C1, on "décode" les textes et on calcule les fréquences. On retient la clef avec les caractères offrant le meilleur score en suivant l'analyse fréquentielle. L'ensemble des clefs sera mémorisé dans un tableau et classé par ordre de score décroissant. Si plusieurs clés obtiennent le même score, on les classe entre elles avec un critère arbitraire (ordre lexicographique par exemple).

4.1.3 Crack C3 : utilisation d'un dictionnaire de la langue cible

Cette méthode consiste à chercher la clef pour laquelle le texte décodé contient exclusivement des mots présents dans un dictionnaire. Cela suppose qu'on dispose d'un dictionnaire contenant toutes les formes nominales et verbales ; c'est le cas du dictionnaire du scrabble qui vous est fourni. Si on suppose le dictionnaire complet et le message écrit sans faute, la bonne clef est celle pour laquelle tous les mots du message décodé sont dans le dictionnaire.

A partir de la liste des clefs valides calculée avec C1 et C2, on décode les textes et on cherche chaque mot du texte "décodé" dans le dictionnaire fourni [9]. On compte le nombre de mots valides suivant le dictionnaire et on classe les clefs par ordre décroissant de ce nouveau score.

Nota bene : On suppose que seule la première lettre d'un mot du message peut éventuellement être une majuscule. Pour chercher ce mot dans le dictionnaire il faudra remplacer cette majuscule par une minuscule.

4.1.4 Travail à faire

1. Créez les fonctions *break_code_c1.c*, *break_code_c2.c*, e *break_code_c3.c* implémentant chacune des recherches précédentes.
2. Créez un programme *break_code_main.c* permettant de lancer les fonctions ci-dessus, dans l'ordre, avec les options :
   -i fichier-à-décrypter
   -m c1 ou all (c1 lance la méthode c1 seulement, all les trois méthodes dans l'ordre)
   -k longueur-de-la-clef
   -d dictionnaire (cette option n'est utilisée que pour -m all qui lance les trois méthodes c1, c2, c3 dans cet ordre).
   -h donnant l'aide sans autre action
   -l fichier de log Le nom du programme compilé sera *break_code* et il affichera pour chaque clef candidate, son score en nombre de mots présents dans le dictionnaire.

Exemple :

```
./break_code -i Datas/Crypted/ringCxor.txt -m c1 -k 4 -l log.txt
./break_code -i Datas/Crypted/ringCxor.txt -m all -k 4 -d Dicos/english.txt
```

Remarque : La longueur de la clef peut être utilisée de deux manières :
- C'est la longueur exacte de la clef à trouver
- C'est la longueur maximale de la clef. Dans ce cas on devra tester incrémentalement les clefs jusqu'à la longueur maximale. Ce cas ne sera traité que si le premier cas fonctionne parfaitement.

Remarque : Le fichier *ctype.h* propose des fonctions permettant de tester si un caractère est alphanumérique, une ponctuation ou autre. Utilisez ces fonctions plutôt que de les refaire.

4.2 Attaque sur la réutilisation d'un masque jetable

Codez la commande *crack_mask* qui implémente le décryptage en cas de non unicité d'utilisation de la clef de chiffrage (formule (1) section 3). Les messages en entrée seront lus dans des fichiers

9. Le dictionnaire fourni est sans aucune majuscule.

données en ligne de commande et le message résultat sera écrit dans un fichier dont le nom sera également donné en ligne de commande [10] : *commande chiffré1 chiffré2 clair1 clairRésultat.*
Exemple : *./crack_mask chif1.txt chif2.txt test2.txt clair.txt* écrit dans *clair.txt* un message en clair qui doit être identique à *test1.txt*

5 Logiciel final

Rassembler toutes les fonctions dans un programme principal permettant d'invoquer les fonctions implémentées par l'intermédiaire d'une boucle de lecture de commande et exécution. Les commandes sont :

- help : donne la liste des commandes
- list-keys : donne la liste des clefs générées et disponibles et indique celle qui ont déjà été utilisée
- gen-key $< n >$: génère une clef de longueur n
- del-key $< key >$: supprime la clef $< key >$
- encrypt $< in >< out >< key >< method > [< vecteurd'initialisation >]$
- decrypt $< in >< out >< key >< method > [< vecteurd'initialisation >]$
- crack$< in >< out >< length >< dico >$
- quit

Un fichier *log* sera systématiquement utilisé.

10. Maximisez les copier/coller de la commande précédente.