MI44
Securité Informatique
Travaux Pratique 1: Cryptage par Bloc

ALLIOT Renaud

2 mai 2015

1 Introduction

Le but de ce travaux pratique seras de mettre en place un cryptage par réseau de Feistel de quatres tours. Pour ce faire nours procéderont en quatres étapes : tout d'abord il faudras créer des fonctions permettant de convertir les caractères de l'alphabet désiré en mot binaires de cinq bits, puis il faudras programmer la fonction de chiffrement par produit uttilisant la clé fournit par l'uttilisateur. Ensuite il faudras implémenter tout ceci dans le réseau Feistel et enfin programmer la fonction de décryptage.
Ayant choisis de coder en C++ ce projet, j'ai créer une classe "Msg" qui correspond au bloc sur lesquels nous travaillerons. Les précisions sur cette dernières seront présentant dans l'annexe A.

2 Conversion des caractère en binaire

Pour pouvoir travailler avec des bloc informatique on transpose tout d'abord chaques caractère de la manière suivante : A = "0" à Z = "25" et les caractère suivant : " " = 26, "." = 27, "," = 28, "'" = 29, "!" = 30 et "?" = 31. Ceci nous permet de convertir ces caractères en mot binaire de cinq bits correspondant aux nombres décimale lui correspondant ecrit en binaire.

Pour ce faire j'ai décidé de créer une méthode pour la classe msg : `void encrypt_binaire()` . Comme les lettres se suivent à la manière du tableau ASCII on récupère la valeur du caractère dans le tableau ASCII à laquel on retire 65 (A vaut 65 dans le tableau ASCII) puis l'on uttilise une boucle interative pour le combiner en binaire. Dans les cas des caractère spéciaux on test chaque code ASCII et on lui associe le mot binaire voulu.

La fonction inverse `void decrypt_binaire()` uttilise le même principe : On stocke les mots binaires des caractère spéciaux dans plusieurs variables, on effectue les tests pour chacun, si aucun ne reussis c'est que le mot est une lettre et dans ce cas la on calcule sa forme décimal plus 65 pour avoir sont code ASCII.

3 Fonction de chiffrement par produit

Cette fonction nécéssite deux opérations : le décalage binaire sur la gauche et l'addition en base deux qui correspond au ou exclusif.

Décalage binaire gauche

Pour cela j'ai créer une methode pour la classe msg : `msg shiftLeftMsg(int shiftValue) const` . Qui renvoie le bloc décalé de shiftValue vers la gauche. On uttilise la case du bits de poids faible comme support pour échanger chaque bits avec celui à sa gauche.

Addition en base deux

Pour cela on somme chaque entre bits et si l'on obtient un entier supérieur à 2, on le force à 0. On l'effectue pour chaque bloc.
Pour faciliter l'écriture du programme, j'ai décider de surcharger l'opérateur ou-exclusif (ˆ et ˆ = en C++).

Implémentation

J'ai donc décider d'implémenter tout cela dans une méthode : `msg function(const msg& key) const` . Elle renvoie le bloc crypté à l'aide de la clé "key" par la fonction de chiffrement par produit i.e. un décalage binaire du bloc suivis d'un ou-exclusif avec la clé.

4 Reseau de Feistel

Le reseau de Feistel crypte chaque bloc du message en quatres tours, il faudras donc coder un tour qu'on réitère quatres fois pour crypté chaques bloc. Mais aussi le découpage du message en bloc et de la clé en bloc.

Un tour

Pour un tour on effectue les opérations suivantes dans cette ordre :

1. $G_{i+1} = D_i$
2. $D_{i+1} = G_i \oplus function(D_i, K_{i+1})$

$D_i et G_i$ désigne respectivement les bloc droite et gauche du tour i, K_i désigne le bloc de la clé du tour i et function la fonction de chiffrement par produit.
On uttilise donc la fonction suivante : `void round(msg& leftBlock, msg& rightBlock, const msg& key)` . Seul la clé est conservé, leftBlock et rightBlock (respectivement D_i et G_i) sont modifiés par leur version crypté (respectivement D_{i+1} et G_{i+1}).

Découpe du message et de la clé

Pour la cas de la clé l'algorithmes est simple. Si l'on prend la clé suivante : "$C_1C_2C_3C_4$", on effectue trois itération pour récupéré C_1C_2 à C_3C_4 et pour le dernier cas on effectue l'opération à la main pour récupérer C_4C_1. La fonction uttiliser est `vector<msg> cutMsg(const string& key)` et renvoie donc la tableau de chaque sous-clé sans altérer la chaine de caractère de la clé
Pour le message s'il possède un nombre de caractère qui est un multiple de quatres on a aucun problème, cependant si ce n'est pas le cas, on doit compléter le message d'autant de caractère qu'il faut pour que sa taille devienne un multiple de quatres. J'ai donc décider d'ajouter des espaces, ces dernier n'apparaissant pas lors du décryptage il n'altère pas la lecture du message. On test donc la taille du message modulo 4 pour savoir combien de caractère à ajouter à chaque fois. On uttilise la fonction suivante : `vector<msg> cutStr(const string& str)` qui renvoie le tableau de bloc sans altérer le message non crypté.

Implémentation

On implémente donc tout cela de la manière suivante :
— On découpe la clé et le message en sous bloc avec `cutKey` et `cutStr` .
— On effectue autant d'itération que nécessaire pour parcourir le message. On enregistre les bloc courant dans `stackRight` et `stackLeft` (respectivement block droit et gauche). On applique quatres fois `round` à `stackRight` et `stackLeft` avec la bonne sous-clé.
— On convertit les bloc en caractère et on ajoute en fin de la chaine de caractère de retour.
On uttilise donc la fonction suivante : `string feistel(const string& message, const string& key)` qui n'altère ni le message, ni la clé et renvoie le message chiffré.

Exemple :

— "AAAA??BB" ↔ "".
— "??BBAAAA" ↔ "".
— "BBAABBAA" ↔ "".
— "AABBAABB" ↔ "".
— "CRYPTOGRAPHIE" ↔ "".
— "FEISTEL" ↔ "".

5 Décryptage du réseau

Le décryptage est assez simple, on effectue l'opération inverse lors des cycles de décryptage :

1. $D_i = G_{i+1}$
2. $G_i = D_{i+1} \oplus function(D_i, K_{i+1})$

On a donc simplement pour nouvelle fonction : `void decrypt_roud(msg& leftBlock, msg& rightBlock, const msg& ke` `string decrypt_feistel(const string& message, const string& key)` qui fonctionne de prenne en paramètre et renvoie les même variable que celle de cryptage.

6 Conclusion

Le programme fonctionne correctement, cependant il faudrait voir pour prendre en charge les minuscule et/ou le tableau ACII (hormis les caractère de 0 à 31 et 127 qui sont des caractère purement informatique et donc illisible). On peut aussi implémenter d'autre mode de cryptage, comme le CBC, permettre à l'uttilisateur de modifier le nombre de tour, d'entrer une clé de tailler quelconque.

A Classe "Msg" : Attribut et méthodes

La classe Msg correspond à un bloc de deux caractère en binaire. Elle est uttilisé lors de Feistel.

Attribut

`vector<int> firstChar` qui est le bloc des bits de poids faible (caractère de droite).
`vector<int> secondChar>` qui est le bloc des bits de poids fort (caractère de gauche).

Méthode

`msg()` Constructeur par défaut qui met chaque caractère égal à 'A'.
`msg(const vector<int>& a, const vector<int>& b)` Constructeur surchargé qui affecte 'a' à 'firstChar' et 'b' à 'secondChar'.
`msg(char a, char b)` Constructeur surchargé qui met a dans firstChar et b dans secondChar en les convertissant.
`msg(const msg& copy)` Constructeur par copie.

`msg& operator=(const msg& c)` Surcharge de l'opérateur d'affectation.
`bool isEqual(const msg& b) const` Methode itermédiaire pour le test d'égalité et d'inégalité.
`msg& operator^=(const msg& b)` Surcharge de l'opérateur XOR raccourcis

`vector<int> getFirstChar() const` Accesseur de firstChar
`void setFirstChar(vector<int> value)` Modificateur de firstChar avec un vector<int>.
`void setFirstChar(char c)` Modificateur de firstChar en convertissant un caractère c en binaire.
`void setFirstChar(int pos, int value)` Modificateur du bit "pos" de firstChar par "value".
`vector<int> getSecondChar() const` Accesseur de secondChar. `void setSecondChar(vector<int> value)` Modificateur de secondChar avec un vector<int>.
`void setSecondChar(char c)` Modificateur de secondChar avec un caractère c convertis en binaire.
`void setSecondChar(int pos, int value)` Modificateur du bit "pos" de firstChar par "value".

`msg shiftLeftMsg(int shiftValue) const` Renvoie *this décaler de "shiftValue" vers la gauche
`msg shiftRightMsg(int shiftValue) const` Renvoie *this décaler de "shiftValue" vers la droite.
`msg function(const msg& key) const` Renvoie function(*this, key).

`void encrypt_binaire(char a, char b)` Convertit les caractère 'a' et 'b' en mot binaire qu'il met à la place de secondChar et firstChar (dans cette ordre).
`void decrypt_binaire(char& a, char& b) const` Convertit firstChar et secondChar en caractère dans 'a' et 'b'.

`void displayMsg(ostream& stream)` Ecris firstChar et secondChar dans stream.

Fonction externe

`bool operator==(const msg& a, const msg& b)` Surcharge du test d'égalité pour les msg.
`bool operator!=(const msg& a, const msg& b)` Surcharge du test d'inégalité pour les msg.
`msg operator^(const msg& a, const msg& b)` Surcharge de l'opérator XOR pour les msg.
`ostream& operator<<(ostream& stream, msg& a)` Surcharge de l'opérateur flux de sortis pour msg.