INFO-F-105
Premier projet d'assembleur

Nikita Veshchikov

Année académique 2015–2016

1 Introduction

Le chiffrement est un des outils utilisés en cryptographie qui permettent de transférer et de stocker des messages de manière sécurisée. Un message *chiffré* ne peut être déchiffré, donc lu que par une personne qui possède la bonne *clé secrète*. L'algorithme de chiffrement a besoin de deux paramètres : la clé secrète et le message à chiffrer, aussi appelé le *texte clair* (ou *plaintext*, en anglais). Cet algorithme produit alors ce qu'on appelle le *texte chiffré* (ou *ciphertext*, en anglais).

Dans une affaire récente, le FBI demande à la société « Pear » de casser la sécurité d'un de ses produits connu sous le nom « jTelephone », en y introduisant une *backdoor* pour court-circuiter sa protection. En réaction, la société Pear voudrait encore améliorer la sécurité de ses produits en utilisant un nouvel algorithme de chiffrement. Elle vous demande d'implanter cet algorithme en assembleur pour atteindre une bonne efficacité. Cet algorithme de chiffrement que Pear demande d'implanter s'appelle *Ultra Light Blockcipher* (ULB).

2 Description d'algorithme Ultra Light Blockcipher

L'algorithme *Ultra Light Blockcipher* réalise ce qu'on appelle un *chiffrement par bloc* : il découpe le message en morceaux de taille fixe, appelés « blocs », et chiffre chacun indépendamment. Il utilise une clé de 32 bits et chiffre un bloc clair de 64 bits en produisant un bloc chiffré de 64 bits également.

Cet algorithme se décrit à l'aide des opérations suivantes :

- XOR, un ou exclusif bit à bit ;
- SWP, un échange (swap) entre les moitiés gauche et droite d'un bloc de 64 bits ;
- Inv, l'opposé modulaire dans $\mathbb{Z}_n$ où $n = 2^{32} - 1$;
- NOT, le complément bit à bit.

Normalement, vous connaissez bien les opérations XOR, NOT et SWP ; on ne va donc pas les revoir en détail.

Les opérations dans un groupe $\mathbb{Z}_n$ se font modulo n. Par exemple, si on travaille dans le groupe $\mathbb{Z}_7$, le résultat d'une opération est dans l'ensemble $\{0,1,2,3,4,5,6\}$. Voici deux exemples : $5 \equiv 2+3 \pmod 7$ ou $6 \equiv 5 \times 4 \pmod 7$.

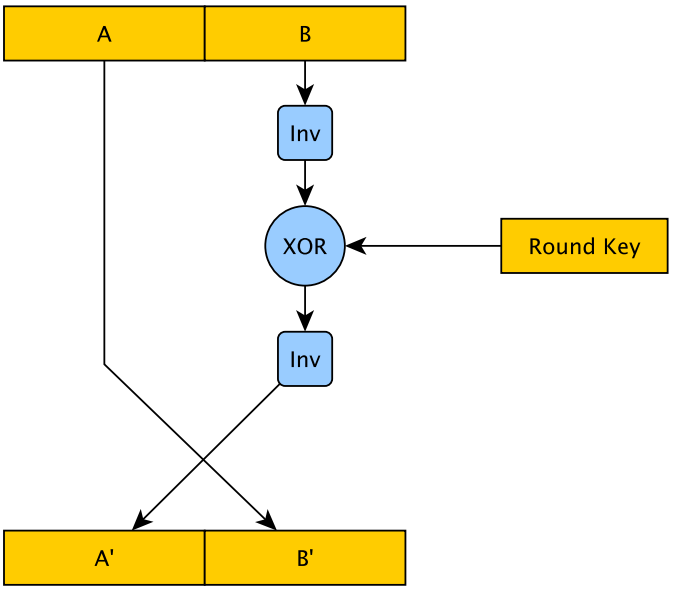

FIGURE 1 – Un round de chiffrement

L'opposé dans $\mathbb{Z}_n$ est, par définition, le nombre qu'il faut additionner dans $\mathbb{Z}_n$ pour obtenir 0. Par exemple, 3 est l'opposé de 4 dans $\mathbb{Z}_7$, puisque $3+4 \pmod 7$ vaut 0.

L'algorithme *Ultra Light Blockcipher* chiffre un bloc de message en répétant 4 fois la même transformation appelée « *tour* » (*round*, en anglais). Ces rounds sont numérotés de 0 à 3.

La figure 1 montre un tour de boucle de l'algorithme de chiffrement. Chaque moitié du bloc complet, notée A ou B, fait 32 bits. À la fin du dernier round les moitiés A et B sont échangées encore une fois de plus, i.e. un SWP supplémentaire.

Chaque round utilise une *sous-clé*, aussi appelée *round key*. Les sous-clés sont engendrées à partir de la clé secrète appelée *master key*. La Figure 2 montre comment générer une sous-clé à partir de la clé secrète ; l'exemple illustre la construction pour le round 1. On y note K_i les octets de la master key et rk_i ceux de la round key. Chaque sous-clé fait donc, comme la master key, 4 octets (ou 32 bits).

Voici un exemple de génération des sous-clés : si la clé secrète vaut `0xCAFEBABE`, alors les sous-clés pour le chiffrement valent (en hexadécimal) :

round 0 `0xCA35CA35`
round 1 `0xFE01FE01`
round 2 `0xBA45BA45`
round 3 `0xBE41BE41`

La Figure 3 montre comment les octets d'un message sont stockés dans l'état interne de l'algorithme au début du chiffrement [1]. La même représentation est utilisée pour stocker le résultat du chiffrement en mémoire (i.e. transférer les données des registres vers la mémoire).

Voici un exemple de chiffrement. Si l'on chiffre le message `'Assembly'` avec Ultra Light Blockcipher, la longueur du message fait exactement un bloc (8 octets ou 64 bits).

1. **Aide :** pensez à l'ordre des octets dans un mot de 4 octets, e.g. un unsigned

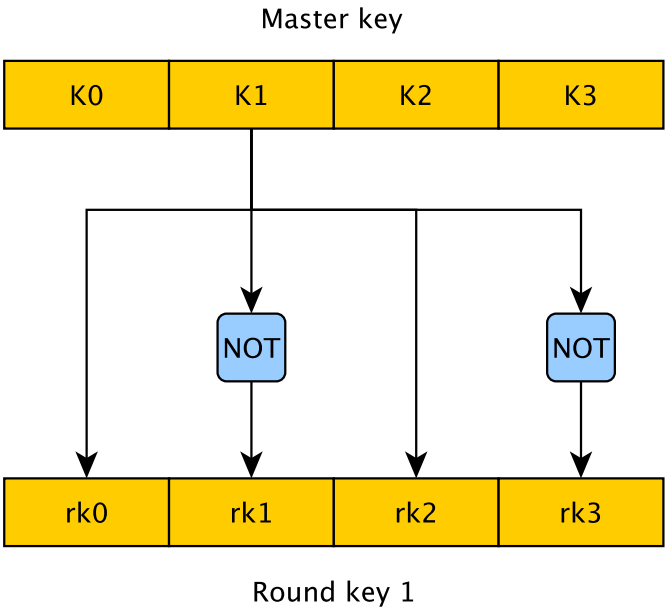

FIGURE 2 – Génération de la sous-clé pour le round 1

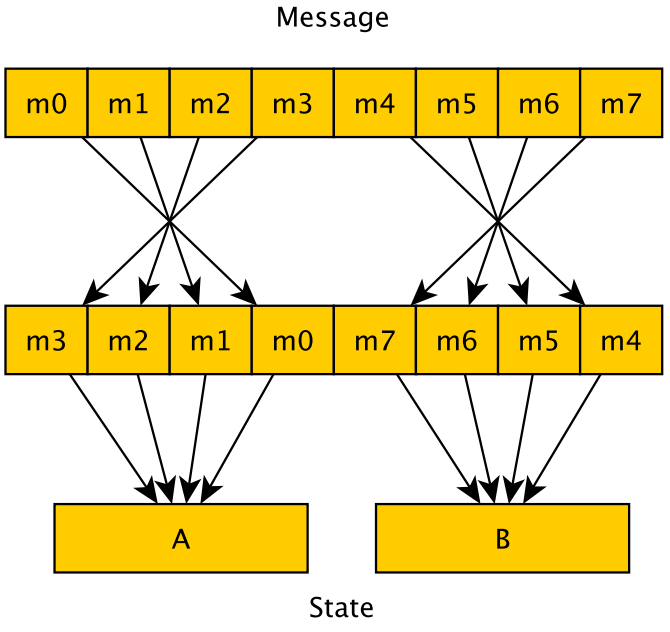

FIGURE 3 – Remplissage d'un bloc de 64 bits avec 8 octets du message

Les octets du message en hexadécimal sont :

```
0x41, 0x73, 0x73, 0x65, 0x6D, 0x62, 0x6C, 0x79
```

Après avoir rempli les parties *A* et *B* de l'état interne de l'algorithme du chiffrement on a :

```
A = 0x 6573 7341
B = 0x 796C 626D
```

Si on utilise la clé secrète `0xCAFEBABE` pour chiffrer le message, alors à la fin de l'algorithme de chiffrement, i.e. après l'application des 4 rounds du chiffrement et du dernier SWP supplémentaire, on a les octets du message chiffré :

```
0x1D, 0x12, 0x1C, 0x09, 0x01, 0x33, 0x33, 0x25
```

Remarquez que l'algorithme est construit de telle façon que le chiffrement est identique au déchiffrement, mais pour déchiffrer il faut y utiliser les sous-clés *dans l'ordre inverse*. Ceci revient simplement à inverser l'ordre des octets de la clé secrète. Donc, pour déchiffrer le message que l'on vient d'obtenir, il suffit d'exécuter le même algorithme de chiffrement avec la clé `0xBEBAFECA`.

3 Projet

On vous demande d'implanter l'algorithme Ultra Light Blockcipher en assembleur.

Pour illustrer son fonctionnement, votre programme doit déchiffrer et afficher le message suivant (de 184 octets) qui a été *chiffré* avec la clé `0xDEADBEEF`.

```
0x12, 0x01, 0x14, 0x15, 0x01, 0x2d, 0x2c, 0x25, 0x0f, 0x0e, 0x13, 0x41,
0x2e, 0x23, 0x36, 0x2b, 0x12, 0x40, 0x09, 0x0d, 0x62, 0x1b, 0x2d, 0x37,
0x05, 0x0e, 0x14, 0x01, 0x32, 0x2e, 0x27, 0x2f, 0x40, 0x09, 0x13, 0x40,
0x36, 0x2b, 0x2d, 0x2c, 0x05, 0x03, 0x14, 0x4e, 0x21, 0x2d, 0x30, 0x30,
0x47, 0x16, 0x05, 0x40, 0x62, 0x1b, 0x2d, 0x37, 0x05, 0x13, 0x13, 0x06,
0x31, 0x37, 0x21, 0x21, 0x40, 0x04, 0x05, 0x03, 0x37, 0x2e, 0x2e, 0x3b,
0x05, 0x04, 0x40, 0x14, 0x30, 0x3b, 0x32, 0x36, 0x05, 0x13, 0x13, 0x01,
0x2a, 0x27, 0x62, 0x2f, 0x39, 0x0f, 0x15, 0x12, 0x25, 0x27, 0x63, 0x48,
0x01, 0x12, 0x04, 0x40, 0x62, 0x30, 0x27, 0x35, 0x5a, 0x40, 0x23, 0x08,
0x28, 0x2d, 0x29, 0x27, 0x2e, 0x0f, 0x12, 0x12, 0x37, 0x21, 0x29, 0x62,
0x0f, 0x04, 0x05, 0x13, 0x2b, 0x31, 0x62, 0x21, 0x02, 0x09, 0x0e, 0x01,
0x62, 0x2b, 0x2c, 0x62, 0x0e, 0x04, 0x40, 0x17, 0x30, 0x3b, 0x62, 0x23,
0x13, 0x40, 0x01, 0x40, 0x30, 0x2b, 0x36, 0x27, 0x03, 0x0f, 0x04, 0x05,
0x01, 0x69, 0x69, 0x62, 0x01, 0x40, 0x04, 0x0f, 0x62, 0x23, 0x31, 0x62,
0x0e, 0x14, 0x01, 0x14, 0x21, 0x37, 0x2f, 0x27, 0x40, 0x5a, 0x4d, 0x49,
0x2b, 0x2d, 0x2c, 0x6c
```

Comme vous pouvez constater, 184 est un multiple de 8 (la taille du bloc). Votre code ne doit donc pas gérer le cas d'un message dont la taille n'est pas un multiple de la taille du bloc. Pour les curieux, dans ce cas on utiliserait un mécanisme qui s'appelle le « *padding* » (ou remplissage).

Le message chiffré doit être encodé dans la section `.data` de votre code assembleur tel qu'il est présenté ci-dessus.

Votre programme fera appel à un code `C++` pour l'affichage. Il ne doit afficher que le résultat du déchiffrement éventuellement avec un `endline` à la fin ; rien de plus. Un exemple d'utilisation de ce code `C++` pour afficher un message se trouve dans les fichiers `message.asm` et `print.cpp`.

Voici la suite de commandes qui seront utilisées pour tester votre projet (en supposant que le message secret est `"Hello, world!"`) :

```
~$ make
## compilation ##
~$ ./cipher
Hello, world!
```

4 Bonus

Voici encore un message chiffré avec une clé qu'on ne connais pas :

```
0x14, 0x16, 0x13, 0x16, 0x01, 0x22, 0x32, 0x21, 0x57, 0x2e, 0x18, 0x02,
0x22, 0x32, 0x21, 0x61, 0x12, 0x57, 0x11, 0x18, 0x60, 0x28, 0x21, 0x36,
0x03, 0x1f, 0x12, 0x57, 0x35, 0x2e, 0x24, 0x60, 0x12, 0x03, 0x57, 0x1c,
0x33, 0x25, 0x23, 0x32, 0x5a, 0x5e, 0x57, 0x24, 0x25, 0x39, 0x60, 0x7a,
0x1a, 0x12, 0x57, 0x16, 0x25, 0x2e, 0x24, 0x60, 0x1a, 0x16, 0x1e, 0x1b,
0x2e, 0x60, 0x25, 0x6d, 0x1f, 0x57, 0x03, 0x1f, 0x60, 0x37, 0x29, 0x34,
0x12, 0x04, 0x04, 0x16, 0x29, 0x33, 0x60, 0x2d, 0x18, 0x57, 0x10, 0x12,
0x27, 0x25, 0x60, 0x34, 0x15, 0x18, 0x19, 0x02, 0x34, 0x60, 0x21, 0x60,
0x1e, 0x19, 0x03, 0x56, 0x33, 0x60, 0x30, 0x2f
```

On vous propose de se mettre dans la peau d'un attaquant et utiliser la *cryptanalyse* pour retrouver la clé et ensuite déchiffrer le message (pour recevoir des points bonus).

Indications complémentaires

Vous devez soumettre un fichier compressé `.zip` qui contient un dossier. Ce dossier doit s'appeler `<nom>_<prenom>` (pas de majuscules ni de caractères accentués), le fichier zip doit porter le même nom. Par exemple, Chuck Norris doit soumettre un fichier `norris_chuck.zip` qui contient le dossier `norris_chuck` avec son projet. Les fichiers à soumettre :

- `print.cpp` qui contient le code que vous utilisez pour afficher le message déchiffré ;
- `cipher.asm` qui contient le code « main » avec l'implantation de l'algorithme de chiffrement ;
- `Makefile` qui produit un fichier exécutable `cipher`.

Bon travail.

Consignes pour la remise du projet

À respecter scrupuleusement !

1. Votre projet doit comporter **votre nom** et **votre section**.
2. Votre projet doit être **dactylographié**. Les projets écrits à la main ne seront **pas corrigés**.
3. Votre code doit être **commenté**.
4. Vous devez respecter les modalités suivantes :
   - Rendre une copie papier du projet au secrétariat étudiant.
   - Poster votre fichier avec le projet sur l'UV.
   - Date : **Le lundi 18 avril 2016**
   - Heure : **Avant 10 heures sur l'UV** et **entre 10 heures et 12 heures au secrétariat**

   Après ces heures de remise, les projets sont considérés comme en retard et **ne seront plus acceptés**.