DM: Implantation de l'attaque par le milieu contre un chiffrement par bloc

christina.boura@uvsq.fr

7 mars 2023

1 Généralités

Le but de ce devoir-maison est double. Il consiste d'abord en l'implantation en `Python` d'un chiffrement par bloc donné et en l'implantation ensuite de l'attaque par le milieu, vue en cours, sur la version double de ce chiffrement.

Modalités pratiques Vous pouvez travailler seuls ou en groupe de 2 personnes au plus. Bien évidement, chaque équipe devra travailler indépendamment des autres. La date limite pour rendre votre projet est le **mardi 4 avril** à **23 heures**. Aucun retard ne sera toléré et les projets rendus tardivement ne seront pas notés. Les projets doivent être envoyés par mail à l'adresse `christina.boura@uvsq.fr` sous la forme d'une archive compressée ayant comme nom votre nom de famille (ou les deux noms si vous travaillez en binôme). Cette archive doit contenir :

- tous vos fichiers `.py`
- un fichier `README` expliquant comment compiler et exécuter votre code (avec un fichier `Makefile` ou équivalent si nécessaire)
- un fichier texte supplémentaire d'une page maximum (sous forme `.txt`, `.pdf`, `.odt` ou `.docx`), expliquant les choix particuliers que vous avez fait pour l'implementation (optionnel) mais surtout contenant une description de votre attaque par le milieu (comment trouver les éléments communs de deux listes, combien des éléments communs vous avez trouvé, comment avez-vous fait pour trouver la bonne clé etc.)

Le DM est individualisé. Vous trouverez sur `eCampus` un fichier contenant pour chaque étudiant deux couples clairs-chiffrés à utiliser. Si vous travaillez en binôme, vous pouvez prendre les données de l'un ou de l'autre, mais il faudra noter explicitement dans le fichier `README` quelles données vous avez utilisé.

L'attaque par le milieu est une attaque qui vise à retrouver la clé secrète (k_1, k_2) utilisée pour un chiffrement double. Cette clé sera différente pour chaque étudiant. Le plus important dans ce projet est que votre attaque fonctionne correctement, dans le sens qu'elle renvoie la bonne clé secrète. Cependant, votre code doit être clair, lisible et très bien commenté. La bonne présentation du code sera notée favorablement. Des points seront également donnés pour des bons choix d'implementation du chiffrement par bloc et de l'attaque. Un code qui fonctionne et qui utilise peu de mémoire et/ou qui est rapide sera mieux noté qu'un code qui fonctionne correctement mais qui n'est pas très optimal. Il y a notamment plusieurs façons de coder le chiffrement. Vous êtes libres de choisir le mode d'implémentation qui vous semble le plus adapté. Cependant, les bons choix d'implémentation qui rendront votre code plus efficace et plus compact recevront des points supplémentaires.

2 Spécifications du chiffrement par bloc `PRESENT24`

Le chiffrement par bloc que vous devez implanter est une version réduite du chiffrement `PRESENT`. `PRESENT` est un chiffrement à bas coût, conçu en 2007 par Bogdanov et al. [1] et standardisé ensuite par l'ISO. La version de `PRESENT` que vous allez implanter, et qu'on appellera désormais `PRESENT24` prendra en entrée un message clair de 24 bits ainsi qu'une clé maître de 24 bits et produira en sortie un bloc chiffré de 24 bits également. Cette version est décrite parmi d'autres dans [2]. `PRESENT` appartient à la famille des chiffrements dits SPN (Substitution-Permutation Network). Cette construction est la deuxième famille largement utilisée pour concevoir un chiffrement par bloc, l'autre étant les réseaux de Feistel. Le principe d'un chiffrement SPN est simple : chaque tour du chiffrement est constitué de trois couches :

— Une addition (XOR) de la sous-clé du tour à l'état ;
— une couche de substitution pour assurer la confusion ;
— une couche linéaire pour assurer la diffusion.

Le nombre de tours de `PRESENT24` est fixé à `10`.

Algorithme de cadencement de clé La clé de `PRESENT24` est constituée de 24 bits. On l'appellera *clé maître*. Un algorithme, appelé algorithme de cadencement de clé (ou key schedule en anglais) est appliqué à la clé maître afin de produire à partir de celle-ci 11 sous-clés K_i, $1 \leq i \leq 11$, de 24 bits chacune. Les 10 premières sous-clés seront utilisées pour l'opération de l'addition de la sous-clé aux 10 premiers tours, tandis que la dernière sous-clé K_{11} sera additionnée à l'état final pour produire le chiffré.

On note `Etat` le registre de 24 bits qui contiendra l'état. Le **bit de poids faible** de ce registre, ainsi que de toutes les autres valeurs utilisées, est considéré à **droite**. On appelle également `Substitution` la fonction non-linéaire et `Permutation` la fonction linéaire. Chacune de ces fonctions, décrites dans les sections suivantes, prennent en entrée le registre `Etat` et donnent en sortie le même registre après l'avoir mis à jour. L'algorithme suivant donne une description de l'algorithme de chiffrement.

Algorithme 1 : Fonction de chiffrement

Données : Un message m de 24 bits et 11 sous-clés K_i, $1 \leq i \leq 11$, produites par l'algorithme de cadencement de clé
Sortie : *Un message chiffré c de 24 bits*

`Etat` $\leftarrow m$
pour $i = 1$ *jusqu'à* 10 **faire**
  `Etat` $\leftarrow$ `Etat` $\oplus K_i$
  `Etat` $\leftarrow$ `Substitution(Etat)`
  `Etat` $\leftarrow$ `Permutation(Etat)`
`Etat` $\leftarrow$ `Etat` $\oplus K_{11}$
$c \leftarrow$ `Etat`
retourner c

2.1 Addition de la sous-clé à l'état

Étant donnée la sous-clé du tour i, $K_i = \kappa_{23}^i \kappa_{22}^i \cdots \kappa_0^i$ pour $1 \leq i \leq 10$ et l'état actuel `Etat` $= b_{23} b_{22} \cdots b_1 b_0$, cette étape consiste juste en l'opération

$$b_j \leftarrow b_j \oplus \kappa_j^i,$$

où $\oplus$ représente l'opération XOR.

2.2 La fonction `Substitution`

La substitution est effectuée à l'aide d'une boîte-S, qu'on notera S, qui prend en entrée 4 bits et qui donne en sortie 4 bits, décrite par le tableau ci-dessous :

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S[x]	c	5	6	b	9	0	a	d	3	e	f	8	4	7	1	2

Chaque entrée et sortie de la boîte-S est constituée de 4 bits et peut être donc représentée par un chiffre hexadécimal. C'est cette notation hexadécimale qui est utilisée pour décrire l'action de la boîte-S. Par exemple, $\texttt{0011}_2 = \texttt{3}_{16} \rightarrow \texttt{b}_{16} = \texttt{1011}_2$.

Pour cette étape, l'état de 24 bits $b_{23} b_{22} \cdots b_1 b_0$ est divisé en 6 mots de 4 bits $w_5 w_4 \cdots w_0$, où $w_i = b_{4i+3} || b_{4i+2} || b_{4i+1} || b_{4i}$ pour $0 \leq i \leq 5$. La boîte-S est appliquée alors à chaque mot w_i de l'état, transformant tous les mots w_i en $S[w_i]$, pour $0 \leq i \leq 5$.

2.3 La fonction `Permutation`

La couche linéaire est effectuée à l'aide d'une permutation bit-à-bit P. Le bit i de l'état bouge après la couche linéaire à la position $P(i)$, comme décrit par la table suivante :

i	0	1	2	3	4	5	6	7	8	9	10	11
$P(i)$	0	6	12	18	1	7	13	19	2	8	14	20
i	12	13	14	15	16	17	18	19	20	21	22	23
$P(i)$	3	9	15	21	4	10	16	22	5	11	17	23

2.4 L'algorithme de cadencement de clé

L'algorithme de cadencement de clé de `PRESENT24` est directement repris de celui de la version de `PRESENT` qui utilise une clé de `80` bits. Pour cette raison on utilise, pour cette version aussi, un registre K de 80 bits, qu'on représente comme $k_{79}k_{78}\dots k_0$, pour sauvegarder la clé maître. Au départ, la clé maître qui fait 24 bits est copiée dans les 24 bits de poids fort du registre K (bits $k_{79}\dots k_{56}$). Les 56 bits restants sont complétés par des 0.

Exemple Si la clé maître est `f34ab7`, le registre K au départ de l'algorithme contiendra la valeur (en notation hexadécimale)

`f34ab700000000000000`.

Ici, comme partout ailleurs, les bits de poids faible sont à droite.

Au tour i, la sous-clé K_i de 24 bits et constituée de 24 bits suivants du registre K :

$$K_i = k_{39}k_{38}\dots k_{17}k_{16}.$$

Après avoir extrait la sous-clé du tour, le registre $K = k_{79}k_{78}\dots k_0$ est mis à jour de la façon suivante :

1. $[k_{79}k_{78}\dots k_1k_0] = [k_{18}k_{17}\dots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus i$

Dans la première étape le registre est simplement pivoté de 61 positions vers la gauche. Dans l'étape suivante la même boîte-S que celle utilisée pour le chiffrement est appliquée aux 4 bits les plus à gauche du registre. À la dernière étape, on additionne (avec un XOR) les bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ du registre à la représentation binaire d'une valeur i qui correspond au numéro du tour que nous sommes en train de traiter. Donc pour le tour 1 on ferra un XOR avec $i = 1 = (\texttt{00001})$, pour le tour 2 avec $i = 2 = (\texttt{00010})$ etc.

2.5 Deux tours du chiffrement

La figure 1 représente graphiquement deux tours de `PRESENT24` (tours i et $i+1$).

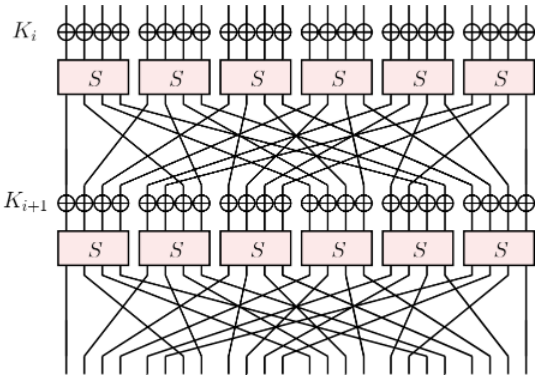

FIGURE 1 – 2 tours du chiffrement `PRESENT24`

2.6 Vecteurs de test

Les vecteurs de test suivants sont donnés en notation hexadécimale. Le bit le plus faible de chaque valeur est à droite.

Message clair	Clé maître	Message chiffré
000000	000000	bb57e6
ffffff	000000	739293
000000	ffffff	1b56ce
f955b9	d1bd2d	47a929

De manière plus détaillée, on donne pour la clé 000000 et le message 000000 l'évolution de l'état à chaque tour. L'état (deuxième colonne) est donné à l'entrée du tour i et donc avant l'addition avec la sous-clé. La dernière ligne (ligne 11) contient l'état juste avant le XOR avec la dernière sous-clé K_{11} ainsi que la valeur de celle-ci. Le chiffré final est alors produit en faisant un XOR entre les entrées de la ligne 11. Dans cet exemple, le chiffré sera donc bb57e6.

Tour i	Etat	Sous-clé K_i
1	000000	000000
2	fff000	000000
3	1c7e00	000001
4	2bb02d	000001
5	727880	400062
6	a19d6e	80002a
7	2fcb17	c00033
8	14a4a1	40005b
9	7492dd	00064c
10	fab2b5	800284
11	fb54b3	400355

2.7 À vos machines

1. Implantez le chiffrement PRESENT24 en Python. Vérifiez en utilisant les vecteurs de test ci-dessus que votre code fonctionne correctement.
2. Implantez la fonction de déchiffrement en Python dans un fichier séparé. Pour que le déchiffrement fonctionne correctement, il faudra considérer les fonctions inverses de la boîte-S et de la permutation bit-à-bit P. Noter les inverses de ces fonctions dans le fichier texte qui accompagne votre projet. Vérifiez que votre déchiffrement marche bien, en chiffrant un message quelconque avec PRESENT24 et en déchiffrant par la suite le chiffré obtenu.

3 Attaque par le milieu contre le chiffrement 2PRESENT24

À l'instar du DoubleDES on considère le chiffrement double 2PRESENT24. Ce chiffrement par bloc prend en entrée deux clés k_1 et k_2 et chiffre un message m de 24 bits une première fois avec PRESENT24 en utilisant la clé k_1 et ensuite il chiffre le résultat une deuxième fois avec le même chiffrement mais avec la clé k_2, pour produire le chiffré c :

$$c = \texttt{2PRESENT24}_{k_1,k_2}(m) = \texttt{PRESENT24}_{k_2}(\texttt{PRESENT24}_{k_1}(m)).$$

3.1 À vos machines

1. Implantez en Python l'attaque par le milieu vue en cours contre 2PRESENT24. Pour cela vous aurez besoin de deux couples clair-chiffré (m_1, c_1) et (m_2, c_2) produits avec une clé secrète (k_1, k_2). Les couples clair-chiffré sont donnés dans un fichier séparé sur eCampus et sont différents pour tous. En particulier, une clé différente a été utilisée pour chaque étudiant pour les générer.
2. Expliquez brièvement comment avez vous fait pour trouver les éléments communs de vos deux listes. Combien d'éléments communs avez vous trouvé ? Comment avez-vous fait pour déterminer la bonne clé ?

3. Appliquez votre attaque en utilisant les couples clair-chiffré donnés et notez dans le fichier texte la clé secrète (k_1, k_2) que vous avez trouvé. Si votre programme vous retourne plusieurs clés candidates, notez-les toutes.

Références

[1] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, C. Vikkelsoe, Charlotte, *PRESENT : An Ultra-Lightweight Block Cipher*, in Proceedings of CHES 2007, p. 450–466, *Lecture Notes in Computer Science*, 4727, Springer (2007)
`https://iacr.org/archive/ches2007/47270450/47270450.pdf`

[2] G. Leander, *Small Scale Variants Of The Block Cipher PRESENT* (2010)
`https://eprint.iacr.org/2010/143.pdf`