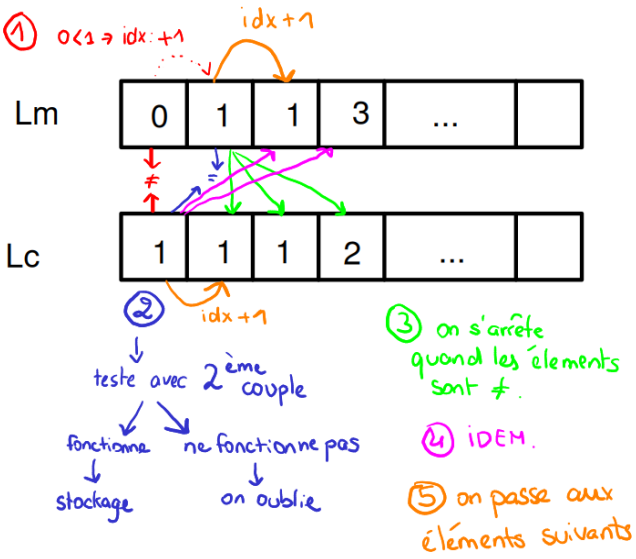
Implémentation de l'attaque par le milieu contre 2PRESENT24

par GODET Chloé & GROSJACQUES Marwane

Python n'étant pas le langage de programmation le plus performant (en temps et en mémoire), nous avons décidé d'utiliser uniquement des entiers et des opérations sur les bits afin d'essayer d'obtenir les meilleures performances. Pour plus d'informations, vous pouvez consulter le README : *https://github.com/chloegdt/PRESENT24*

Pour l'attaque, nous nous sommes inspirés de l'attaque vue en cours sur le double DES. Nous utilisons le premier couple clair-chiffre que l'on chiffre avec toutes les clés possibles (de 0x0 à 0xffffff) et l'on stocke dans la liste Lm le message chiffré obtenu ainsi que la clé utilisée. On réalise la même chose en partant du chiffré, on le déchiffre et on stocke les résultats dans une liste Lc. Nos listes contiennent $2^{24} + 1$ tuples (sous forme : (message intermédiaire, clé)).



Dessin des étapes de la fonction cherchant les éléments communs (attention, pour plus de simplicité dans cet exemple, nous avons représenté le tuple (message intermédiaire, clé) par un entier

Nous cherchons ensuite les éléments communs de ces deux listes, c'est-à-dire les messages obtenus (peu importe que les clés soient les mêmes ou non). Pour cela, nous trions les listes par ordre croissant de message à l'aide de la fonction sort() de python car c'est la plus rapide (complexité O(nlog(n))).

Une fois les listes triées, nous partons du début des deux listes. On regarde si les messages (premier élément du tuple) sont identiques.

- Si ce n'est pas le cas, on incrémente l'indice de la liste qui a l'élément le plus petit (étape1).
- Si c'est le cas, nous testons directement les deux clés candidates à l'aide du deuxième couple clair-chiffré afin d'économiser de la mémoire et de ne pas stocker l'entièreté des clés candidates possibles (étape2). Si les deux clés candidates testées permettent de passer du clair au chiffré (du deuxième couple respectivement) alors on mémorise ces clés candidates dans une liste.

De plus, pour chacune des deux listes, nous testons les prochains éléments si ces derniers sont les mêmes que l'élément à l'indice que nous sommes en train de tester (étape 3 & 4).

Nous retournons ensuite les tuples de clés qui ont fonctionné (et qui ont été stockés dans une liste) à l'issue de l'étape précédente s'ils existent.

Nous avons testés nos deux paires de couples et nous avons obtenus les résultats suivants :

À gauche pour Marwane

```
ceres[14:57:55]:~/Downloads/cours/crypto/dm/present24$ python attaque.py -m
Début de l'attaque avec le message clair : 0xc1a0d et le chiffré : 0x783e29
Création des listes...
[##################]
Listes crées en 9.022363185882568 secondes.

Liste_m triée en 3.3685524463653564 secondes.
Liste_c triée en 3.334977865219116 secondes.

Suite de l'attaque avec le message clair : 0x8556cc et le chiffré : 0x5e51d4
Recherche et test des elements communs entre les deux listes...
[##################]
16774162 elements communs trouvés et testés en 12.196969270706177 secondes.

1 couple(s) de clés trouvé(s) :
k1 = 0xc44276 | k2 = 0xa47390
Attaque terminée en 27.924314737319946 secondes.
```

À droite pour Chloé

```
ceres[14:57:03]:~/Downloads/cours/crypto/dm/present24$ python attaque.py -c
Début de l'attaque avec le message clair : 0x4efbd3 et le chiffré : 0x9c45fa
Création des listes...
[##################]
Listes crées en 9.041048526763916 secondes.

Liste_m triée en 3.376155138015747 secondes.
Liste_c triée en 3.5440845489501953 secondes.

Suite de l'attaque avec le message clair : 0xfdd4e0 et le chiffré : 0x4ae29b
Recherche et test des elements communs entre les deux listes...
[##################]
16782676 elements communs trouvés et testés en 12.586406469345093 secondes.

1 couple(s) de clés trouvé(s) :
k1 = 0x673e22 | k2 = 0xae673a
Attaque terminée en 28.54923987388611 secondes.
```

Pour obtenir des résultats plus rapides nous avons utilisé le module numba qui transforme le code python en langage machine afin de le compiler lors de l'exécution. En effet, sans son utilisation nous mettions environ 32 minutes à trouver notre paire de clés et utilisions 3X plus de RAM.

De plus, si nous avons écrit les fonctions de chiffrement et de déchiffrement dans des fichiers séparés de la façon la plus lisible possible, nous avons choisi d'utiliser pour notre attaque un fichier contenant les deux fonctions complètes (sans appel à d'autres fonctions) ce qui a permis une exécution deux fois plus rapide (~60 secondes à moins de 30 secondes).

Ordinateur utilisé pour les tests : AMD Ryzen7 5800U, 16Go RAM