4CS02 – Cryptologie
Travaux Pratiques
AES et RSA

Année 2024

1 Attaques sur AES

L'algorithme AES complet utilise 10 ou 14 tours (pour des clés respectives de 128 et 256 bits). Cependant, il est courant d'étudier des versions réduites de l'AES (i.e., avec moins de tours) pour comprendre certaines vulnérabilités potentielles. Plus précisément, nous allons utiliser la cryptanalyse différentielle, vue en cours.

La cryptanalyse différentielle s'effectue en général dans un contexte de texte clair choisi, ce qui signifie que l'attaquant est en mesure d'obtenir les résultats chiffrés de textes clairs de son choix. Une telle cryptanalyse repose sur des paires de textes clairs qui ont une différence constante. L'attaquant calcule ensuite les différences qu'il voit dans les textes chiffrés correspondants, afin d'en extraire des motifs pouvant indiquer un biais. Les différences en sortie du chiffrement sont nommées des différentielles.

Ce type d'attaque peut parfois être utilisé par un attaquant pour être capable de distinguer une exécution d'AES d'une génération aléatoire. Dans ce contexte, l'attaquant peut choisir des messages de son choix et obtenir soit le chiffré correspondant, soit une valeur aléatoire. Quand il le souhaite, l'attaquant peut alors décider s'il a affaire au vrai AES ou à une génération aléatoire.

Dans cette première partie, vous allez réaliser une cryptanalyse différentielle sur AES réduit à 1, 2 et 3 tours. Vous allez examiner le comportement de certaines différences d'entrée au fil des tours.

Récupérez le fichier TP_AES.py qui contient l'ensemble des opérations basiques d'AES (SubBytes, ShiftRows, MixColumns, AddRoundKey), la génération de la clé étendue, ainsi que d'autres fonctions basiques qui pourront vous êtes utiles.

1. Implémentez un chiffrement AES complet avec un nombre de tours au choix, pour un bloc de 128 bits (16 octets). La génération des clés étant fournie, il vous reste donc à compléter le chiffrement et le déchiffrement.
2. Générez deux blocs de texte clair P et P', où P et P' ne diffèrent que par un seul bit (par exemple, un bit dans le premier octet) et chiffrez P

et P' en utilisant AES réduit à 1 tour pour obtenir les chiffrés C_1 et C'_1. Calculez la différence binaire (XOR) entre les textes chiffrés C_1 et C'_1. Que constatez-vous ? Qu'en déduisez vous ?

3. Même questions avec 2 tours, puis 3 tours.

2 Attaque d'AES sur 3 tours (facultatif)

Nous allons maintenant voir une attaque un peu plus complexe sur un AES un 3 tours.

1. Implémenter une fonction simple_swap qui prend en entrées deux états AES et qui échange la première colonne différente entre chaque état et retourne les deux états changés. Soient $[a_1, a_2, a_3, a_4]$ et $[b_1, b_2, b_3, b_4]$ ces deux états. Si $a_1 \neq b_1$, alors retourner $[b_1, a_2, a_3, a_4]$ et $[a_1, b_2, b_3, b_4]$, si $a_1 = b_1$ et $a_2 \neq b_2$, alors retourner $[a_1, b_2, a_3, a_4]$ et $[b_1, a_2, b_3, b_4]$, etc.
2. Utiliser cette fonction pour créer un différenciateur d'un AES 3 tours. Ce dernier prend en entrée deux textes clairs p_0 et p_1 qui diffèrent de 3 bits et sort Il fonctionne de la façon suivante :
   (a) On applique inv_shift_rows à p_0 et p_1.
   (b) On calcule $c_0 = \mathsf{encrypt}(p_0, \mathsf{key}, 3)$ et $c_1 = \mathsf{encrypt}(p_1, \mathsf{key}, 3)$.
   (c) On applique inv_mix_columns à c_0 et c_1.
   (d) On applique simple_swap à c_0 et c_1.
   (e) On applique mix_columns à c_0 et c_1.
   (f) On calcule $a_0 = \mathsf{decrypt}(c_0, \mathsf{key}, 3)$ et $a_1 = \mathsf{decrypt}(c_1, \mathsf{key}, 3)$.
   (g) On applique shift_rows à a_0 et a_1.
   (h) Si $a_0 \oplus a_1 = p_0 \oplus p_1$) sortir 1.
   (i) Sortir -1.

3 Utilisation pratique d'AES

Nous allons maintenant considérer une exécution d'AES-128 sur les 10 tours préconisés, et voir comme l'utiliser en pratique pour assurer la confidentialité, l'intégrité et l'authenticité pour un message de n'importe quelle taille.

Pour un message de taille quelconque, il faut dans un premier temps pouvoir exécuter AES-128 sur des blocs pleins (pour rappel, AES utilise des blocs de taille 128 bits, c'est à dire 16 octets). Il faut rajouter un *padding*. Il existe plusieurs façons de faire mais nous allons ici utiliser celui décrit par le standard PKCS#7. Celui fonctionne de la façon suivante : s'il manque N octets pour arriver à un multiple de 16, rajouter N fois l'octet N. Par exemple, s'il manque 4 octets, il faudra rajouter 0x04‖0x04‖0x04‖0x04 à la fin du message initial.

1. Implémentez un tel padding, ainsi que la padding inverse.

2. Implémentez le mode CBC que vous utiliserez avec AES pour calculer un MAC. Comment faut-il vérifier ce MAC ?
3. Implémentez le mode CTR que vous utiliserez avec AES pour chiffrer et déchiffrer un message.
4. Utiliser ces deux dernières étapes pour implémenter un Encrypt-then-MAC avec AES et les modes CTR (pour le "encrypt") et CBC (pour le "MAC").

4 Implémentation de RSA-OAEP

Récupérez le fichier 4CS02_TP_RSA.py qui contient un squelette de l'algorithme RSA OAEP.

1. Commencez par implémenter les différentes étapes de l'algorithme RSA (génération des clés, chiffrement et déchiffrement).
2. En vous aidant de ce qui est fourni dans le fichier, complétez votre code pour obtenir le padding OAEP, et ainsi chiffrer et déchiffrer avec RSA-OAEP.