TP1 - ChaCha20

(ChaCha20) **Exercise 1.** *PRF - ChaCha20*

L'objectif de ce premier exercice est de découvrir et implémenter en python le chiffrement à flot Chacha20, recommandé par l'ANSSI. Il est utilisé dans TLS 1.3 comme alternative de certain modes de l'AES (CCM et GCM) pour le chiffrement des communications sur internet (représente le 's' de `https`), également dans SSH, et dans le générateur aléatoire Linux.

1. Les spécifications de ChaCha20 se trouve sur la RFC 8439. Trouvez cette RFC pour les vecteurs tests et les notations du TP.

Le chiffrement par flot ChaCha20 opère sur des mots de 32 bits (combien d'octets ?) et utilise en particulier une fonction de rotation circulaire à gauche sur ces mots. Pour la suite, pensez à convertir les flots de bits en suites de mots de 32 bits.

2. Écrire une fonction `rot(x,n)` qui réalise la rotation à gauche de n bits pour un mot x de 4 octets. Réalisez un test avec `rot(0x7998bfda,7) = 0xcc5fed3c`.

Pour la suite du TP, vous utiliserez la RFC précédente afin d'implémenter et tester les fonctions demandées. La fonction de base de `ChaCha` opère sur 4 mots a, b, c, d de 32 bits.

3. En vous aidant de la RFC, implémenter et vérifiez la sortie de la fonction `quaterround`.

L'état initial de `ChaCha` a une longueur de 512 bits, où 4 mots sont modifiés par l'appel à la fonction précédente.

4. Écrire une fonction `Qround(state, i1, i2, i3, i4)` où `state` est un tableau de 16 mots et `i1, i2, i3, i4` sont les indices des mots à changer. Tester cette fonction avec les vecteurs tests de la RFC.

La mise à jour de l'état interne du générateur correspond à 8 appels successifs à la fonction précédente, avec une séquence particulière d'entiers.

5. Écrire une fonction `inner_block(state)` qui retounre l'état interne après ces 8 appels. Tester cette fonction avec les vecteurs tests.

`ChaCha` génère des blocs de 512 bits à partir d'une clé secrète de 256 bits, un compteur de 32 bits et un *nonce* aléatoire de 96 bits. L'état initial débute avec 4 mots de 32 bits constants. Leqsuels ? Suivi de la clé secrète, le compteur et finalement, le *nonce*. L'un de vous peut passer au tableau afin de faire un schéma de l'état intial, avec les tailles de chacun des éléments pour mieux représenter les choses. La clé et le *nonce* sont représentes par un tableau d'octet au format *little endian*. Que cela signifie-t-il ?

6. Écrire une fonction `le_bytes2num(b,t,n)` qui prend en entrée un tableau b de $t * n$ octets correspondant au format *little endian* et qui vient réaliser la transformation des octets en t mots de n octets.

7. Écrire une fonction `key_setup(key,counter,nonce)` qui retourne l'état initial comme décrit ci-dessus, les variables `key` et `nonce` données en représentation *little endian*. Testez votre fonction avec l'état intial composé des vecteurs tests.

8. Écrire une fonction `serialize(state)` qui prend en entrée un tableau de mots de 32 bits et qui retourne un tableau d'octet en *litlle endian*.

La fonction de chiffrement génère un bloc de 512 bits en appelant 10 fois la fonction `inner_block` sur l'état initial.

9. Écrire la fonction `ChaCha_block` prenant en entrée une clé `key` de 256 bits (combien de mots de 4 octets ?), un compteur `counter` de 32 bits et un nonce de 3 mots de 4 octets (combien de bits ?), et qui retourne l'état en *little_endian* après 10 appels à la fonction `inner_block` et son addition à l'état initial.

Le chiffrement `ChaCha` chiffre des blocs de 512 bits. Afin de s'assurer de la bonne longueur du message, il faut ajouter un pad de sorte que le message contienne un nombre entier de block. Le pad fonctionne de différentes manières suivant le chiffrement utilisé : ajoute des 0, ajouter la taille du message (en octets) avec un mot de 4 octets suivi ou précédé de 0 afin de vérifier qu'aucun paquet n'est perdu, et ainsi de suite. Le chiffrement que l'on considère ici étant un chiffrement par flot, chaque bloc est chiffré presqu'indépendemment. Seul le compteur relie deux bloc. Ainsi, considérer la taille permet de s'assurer que tout les paquets sont arrivés correctement, et qu'aucun ne manque à l'appel. Pour plus de facilité dans les tests, on considère ici un pad qui ajoute bêtement des 0 jusqu'à obtenir un multiple de 512 bits.

10. Écrire une fonction `padding` qui prend en entrée un message `plaintext` et qui retourne ce même message `plaintext_pad` avec des 0 supplémentaires afin de le padder.

Afin de chiffrer le texte clair `plaintext`, `ChaCha` realise le `XOR` de la suite chiffrante générée par la fonction `ChaCha20_block`. Le compteur caractérise le numéro du block à chiffré en cours. Dans ce cadre, il augmente sa valeur de 1 à chaque chiffrement de block de 512 bits.

11. Écrire la fonction `ChaCha_encrypt` qui prend en entrée la clé `key`, le compteur initial `counter` qui s'incrémentera à chaque utilisation, le nonce `nonce` et le message `plaintext`.

L'algorithme précédent s'appelle en réalité `ChaCha`, le nom de `ChaCha20` étant donné à l'algorithme qui applique 20 tours.

12. De la même manière que la question précédente, implémentez l'algorithme de chiffrement `ChaCha20_encrypt` qui chiffre un message `plaintext` avec 20 tour appliqués à l'état initial. Vous pourrez choisir d'appliquer la fonction `ChaCha_block` 2 fois de suite, l'adapter afin de lui préciser la variante `ChaCha20` qui prendra ensuite le bon nombre de tour, ou bien implémenter une nouvelle fonction `ChaCha20_block`.