SAE3 - Chiffrement RC4 et fonction de hachage

Plan :

1. L'algorithme RC4

1.1 L'algorithme en python

Dans un premier temps nous avons choisi, afin de mieux comprendre le fonctionnement de l'algorithme RC4, de d'abord le programmer dans un langage qui nous est plus familier. Nous nous sommes donc mis d'accord pour utiliser le langage python :

```
import base64

def encode(sh):
    return base64.b16encode(bytes(sh, 'latin')).upper()

def decode(sh):
    return base64.b16decode(sh, casefold=True).decode('latin')

def rc4_init(key):
    S = [i for i in range(256)]
    j = 0

    for i in range(256):
        j = (j + S[i] + ord(key[i % len(key)])) % 256
        S[i], S[j] = S[j], S[i]
    return S

def bytes_to_text(byteList):
    s = ''
    for byte in byteList:
          s += chr(byte)
    return s

def rc4_encrypt(key, input):
    key_stream = []
    key = rc4_init(key)
    j = i = 0
    for x in range(256):
        i = (i + 1) % 256
        j = (j + key[i]) % 256
        key[i], key[j] = key[j], key[i]
```

```
            key_stream.append(key[(key[i] + key[j]) % 256])
        ciphertext = ''
        i = 0
        for char in input:
            enc = chr(ord(char) ^ key_stream[i])
            ciphertext += enc
            i+=1
        ciphertext = encode(ciphertext)
        return bytes_to_text(ciphertext)

    def rc4_decrypt(key, input):
         return decode(rc4_encrypt(key, decode(input)))
```

1.2 L'algorithme en lui même

Historique :

Premièrement le RC4 a été conçu par Ronald Rivest en 1987 par la suite cet algorithme a pris en partie le nom de son créateur (Rivest Cipher 4). Initialement, RC4 était gardé comme un secret industriel, mais il a rapidement gagné en popularité en raison de ses performances et de sa simplicité.

Explication de son fonctionnement :

RC4 est un algorithme de chiffrement à clé symétrique, ce qui signifie que la même clé est utilisée pour le chiffrement et le déchiffrement. Lors de l'initialisation de l'algorithme, la clé fournie en paramètre subit une série d'opérations d'addition modulaire et de permutation sur un tableau de 256 octets, créant ainsi un état interne initial. Ensuite, des échanges d'octets sont effectués pour mélanger davantage les données. La génération du keystream (une génération de nombre pseudo aléatoire) intervient ensuite, où l'état interne est modifié itérativement pour produire une séquence de nombre pseudo aléatoire. Enfin, l'opération XOR (Code Vernam) est appliquée entre le keystream généré et le texte à chiffrer, produisant ainsi le texte chiffré.

1.3 L'algorithme en PHP

Il était nécessaire de coder ce système de cryptage expliqué au dessus en langage PHP, pour l'intégrer à notre site nous avons donc utilisé le code ci-dessous tel quel dans notre application web

```
<?php


function rc4Encrypt($text)
{
    $json = file_get_contents('json/key.json'); // import de la clé de cryptage
stocké dans un JSON dans le repo de notre site web
    $data = json_decode($json, true);
    $key = $data["key"];

    $keyStream = initializeKeyStream($key);
    $encryptedText = '';
```

```
    $i = $j = 0;
    $textLength = strlen($text);

    for ($k = 0; $k < $textLength; $k++) {
        $i = ($i + 1) % 256;
        $j = ($j + $keyStream[$i]) % 256;

        $temp = $keyStream[$i];
        $keyStream[$i] = $keyStream[$j];
        $keyStream[$j] = $temp;

        $char = ord($text[$k]) ^ $keyStream[($keyStream[$i] + $keyStream[$j]) %
256];
        $encryptedText .= chr($char);
    }

    return bin2hex($encryptedText);
}




function rc4Decrypt($key, $encryptedText) {
    $keyStream = initializeKeyStream($key);
    $encryptedText = hex2bin($encryptedText);
    $decryptedText = '';

    $i = $j = 0;
    $textLength = strlen($encryptedText);

    for ($k = 0; $k < $textLength; $k++) {
        $i = ($i + 1) % 256;
        $j = ($j + $keyStream[$i]) % 256;

        $temp = $keyStream[$i];
        $keyStream[$i] = $keyStream[$j];
        $keyStream[$j] = $temp;

        $char = ord($encryptedText[$k]) ^ $keyStream[($keyStream[$i] +
$keyStream[$j]) % 256];
        $decryptedText .= chr($char);
    }

    return $decryptedText;
}

function initializeKeyStream($key) {
    $keyLength = strlen($key);
    $keyStream = range(0, 255);
    $j = 0;

    for ($i = 0; $i < 256; $i++) {
        $j = ($j + $keyStream[$i] + ord($key[$i % $keyLength])) % 256;
```

```
            $temp = $keyStream[$i];
            $keyStream[$i] = $keyStream[$j];
            $keyStream[$j] = $temp;
        }

        return $keyStream;
    }

    ?>
```

2. Recherche bibliographique

Fonction de hachage

Une fonction de hachage est une fonction qui utilise des fichiers de très grande taille de toute sorte pour le transformer en une séquence de taille fixe. Le but d'une fonction de hachage est de donner pour quelconque fichier et sa taille, une signature (le résultat de la fonction de hachage) tout en minimisant les collisions. Une collision survient quand deux fichiers différents produisent la même signature avec la fonction de hachage. Une fonction de hachage a pour particularité le déterminisme, un même fichier donnera toujours la même signature, il doit être rapide, peu importe la taille des données traitées, résistant à la préimage, c'est-à-dire qu'il doit être difficile de retrouver l'entrée originale à partir de sa signature, puis résistant aux collisions.

Une fonction de hachage cryptographique possède ces particularités, à la différence près qu'elle est à sens unique, il doit être facile d'obtenir l'empreinte, mais difficile, voire, dans le meilleur des cas, impossible de retrouver l'entrée originale.

Le MD5

Le MD5 pour Message Digest 5 est une fonction de hachage cryptographique ayant une forte probabilité que les empreintes produites soient différentes. Une empreinte MD5 est représentée en une séquence de 128 bits, ou de 32 caractères hexadécimaux.

MD5 utilise un message divisé en blocs de 512 bits, sur le dernier bloc, on réalise un remplissage, le remplissage MD5 ajoute à la fin du message un 1, suivi d'une séquence de 0 jusqu'au 448ème bit du dernier bloc, les 64 bits restants pour faire un bloc de 512 bits sont dédiés à un entier donnant la taille du message. Dans le cas où le message est divisible par 512, un remplissage est quand même réalisé jusqu'au bloc suivant.

Ainsi, on a des blocs de 512 bits contenant chacun 16 mots de 32 bits.

MD5 utilise 64 constantes K de 32 bits notées:

$$K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$$

Ces constantes peuvent être obtenues par la formule $K_i^{\{256\}} = \lfloor |2^{32} \times \sin(i+1)| \rfloor$

Nous pouvons alors traduire chaque mot en différents K.

Dans son algorithme principal, MD5 va, à partir des différents K obtenus, utiliser le message en groupe de 128 bits, soit, 4 mots de 32 bits pour obtenir sa signature. Ces 4 mots sont notés respectivement A, B, C, D.

Les opérations se déroulent en 4 étapes se déroulant sur 4 blocs de 32 bits, chacune des étapes étant composée de 16 opérations, changeant selon l'étape, des 4 fonctions non linéaires (ET logique, OU logique, XOR logique et NON logique), ainsi qu'une addition ou une rotation vers la gauche. Enfin, chaque bloc est additionné à son étape respective, et on répète ces opérations tant qu'il reste des blocs de 512 bits.
L'empreinte est obtenue par concaténation des 4 blocs de 32 bits