**Implémentation de la fonction de hashage SHA-1**

**SAADI Brahim**

**Elève Ingénieur en 3ème année DSIA**

**Ecole Nationale Polytechnique**

Avenue des Frères Oudek, Hassan Badi, B.P.182, El-Harrach, 16200 Alger,

ALGERIE

[brahim.saadi@g.enp.edu.dz](mailto:brahim.saadi@g.enp.edu.dz),

**Introduction**

SHA-1 (Secure Hash Algorithm 1) est un algorithme de hachage cryptographique qui prend un message en entrée et génère une valeur de hachage de 160 bits, souvent représentée par une chaîne hexadécimale de 40 caractères. Son objectif est de créer une empreinte unique pour un message donné. Il découpe le message en blocs et utilise des opérations mathématiques complexes pour produire la valeur de hachage. Bien que largement utilisé par le passé, il est aujourd'hui obsolète en raison de vulnérabilités. Si vous avez un code à expliquer, veuillez le fournir, et je vous aiderai étape par étape.

**Les étapes de l’algorithme sha-1**

**Étape 1 : Prétraitement du Message**

L'algorithme SHA-1 commence par prétraiter le message d'entrée pour qu'il ait la bonne longueur et structure requises. Cela implique les étapes suivantes :

* Calculez la longueur originale du message en bits.
* Ajoutez un bit '1' au message, suivi de zéros pour que la longueur totale du message soit un multiple de 512 bits.
* Ajoutez la longueur originale du message (sur 64 bits) à la fin du message prétraité.

**Étape 2 : Découpage du Message en Blocs**

Le message prétraité est maintenant divisé en blocs de 512 bits (64 octets). Chaque bloc est ensuite divisé en 16 mots de 32 bits, ce qui nous donne un total de 512 bits.

**Étape 3 : Opérations de Compression**

Pour chaque bloc de 512 bits, les opérations de compression sont effectuées. Voici ce qui se passe à l'intérieur de cette boucle :

* Les 16 premiers mots (0 à 15) sont copiés directement à partir du bloc.
* Ensuite, 64 mots supplémentaires (16 à 79) sont calculés à l'aide de rotations et de combinaisons des mots précédents.
* C'est à ce stade que l'algorithme effectue les opérations non linéaires en fonction de la position dans le traitement. Les opérations de type "et", "ou exclusif" et "ou" sont appliquées, ainsi que des décalages et rotations de bits.

**Étape 4 : Mise à Jour des Valeurs de Hachage**

Après avoir effectué les opérations de compression pour chaque bloc, les valeurs de hachage intermédiaires (**h0**, **h1**, **h2**, **h3**, et **h4**) sont mises à jour :

* Les valeurs actuelles de **h0**, **h1**, **h2**, **h3**, et **h4** servent de base pour les opérations suivantes.
* Les opérations d'addition modulo 2^32 sont utilisées pour mettre à jour ces valeurs.

**Étape 5 : Répétition**

Les étapes 2 à 4 sont répétées pour chaque bloc de 512 bits du message. Le hachage final est obtenu à la fin de ce processus.

**Étape 6 : Valeur de Hachage Finale**

La valeur de hachage finale est généralement représentée sous forme de chaîne hexadécimale de 40 caractères (160 bits). Elle représente l'empreinte numérique unique du message d'origine.

**Code de l'Implémentation**

import struct

def left\_rotate(n, b):

    return ((n << b) | (n >> (32 - b))) & 0xFFFFFFFF

def preprocess\_message(message):

    original\_byte\_len = len(message)

    original\_bit\_len = original\_byte\_len \* 8

    message += b'\x80'

    while len(message) % 64 != 56:

        message += b'\x00'

    message += struct.pack('>Q', original\_bit\_len)

    return message

def sha1(message):

    h0 = 0x67452301

    h1 = 0xEFCDAB89

    h2 = 0x98BADCFE

    h3 = 0x10325476

    h4 = 0xC3D2E1F0

    message = preprocess\_message(message)

    for i in range(0, len(message), 64):

        chunk = message[i:i + 64]

        words = list(struct.unpack('>16I', chunk))

        for j in range(16, 80):

            words.append(left\_rotate(words[j - 3] ^ words[j - 8] ^ words[j - 14] ^ words[j - 16], 1))

        a, b, c, d, e = h0, h1, h2, h3, h4

        for j in range(80):

            if 0 <= j < 20:

                f = (b & c) | ((~b) & d)

                k = 0x5A827999

            elif 20 <= j < 40:

                f = b ^ c ^ d

                k = 0x6ED9EBA1

            elif 40 <= j < 60:

                f = (b & c) | (b & d) | (c & d)

                k = 0x8F1BBCDC

            else:

                f = b ^ c ^ d

                k = 0xCA62C1D6

            temp = (left\_rotate(a, 5) + f + e + k + words[j]) & 0xFFFFFFFF

            e = d

            d = c

            c = left\_rotate(b, 30)

            b = a

            a = temp

        h0 = (h0 + a) & 0xFFFFFFFF

        h1 = (h1 + b) & 0xFFFFFFFF

        h2 = (h2 + c) & 0xFFFFFFFF

        h3 = (h3 + d) & 0xFFFFFFFF

        h4 = (h4 + e) & 0xFFFFFFFF

    return '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)

**Exemple d'Utilisation**

message = "hello.friend"

hashed\_message = sha1(message.encode('utf-8'))

print(f"Message: {message}")

print(f"Hachage SHA-1: {hashed\_message}")

**resultats**

Message: hello.friend

Hachage SHA-1: b57e221136555e425959aacc79e9b51eeff859d0

**Comparaison des résultats avec sha-1 du bibliothèque hashlib**

import hashlib

def sha1\_hash(input\_string):

    sha1 = hashlib.sha1()

    sha1.update(input\_string.encode('utf-8'))

    return sha1.hexdigest()

print(sha1\_hash(message))

sha1\_hash(message) == hashed\_message

**resultats**

Message: hello.friend

Hachage SHA-1: b57e221136555e425959aacc79e9b51eeff859d0

b57e221136555e425959aacc79e9b51eeff859d0

True

**Références**

Pseudo code de Wikipedia: <https://en.wikipedia.org/wiki/SHA-1#SHA-1_pseudocode>

Article sur meduim : <https://infosecwriteups.com/breaking-down-sha-1-algorithm-c152ed353de2>