Experiment No. 7

Aim: For varying message sizes, generate message digest using SHA-1 algorithm and check the integrity of message

Theory:

There are two main kinds of encryption algorithms: Symmetric Key Algorithm (SKA) and Asymmetric Key Algorithm (AKA). In additional, there is also another assistant algorithm called: Hash, which compresses message of any length to certain fixed length (message-digest). This process is irreversible. Hash function can be used in many fields such as: digital signature, message integrity test, and message originality etc. Hash algorithm mainly includes: MD×(Message－Digest Algorithm), SHA×(Secure Hash Algorithms), N-Hash, RIPE-MD, and HAVAL etc.

SHA1 Algorithm:

SHA1 also is another main hash algorithm, which is primarily based on MD4 principle. Because it produces 160-bit output, so SHA1 needs five 32-bit registers. But the method of message digest and data filling works just like MD5 algorithm. SHA has 4 main rounds iterative, and each round has 20 steps operations. There are only some minute differences including: rotate left, addition constant, and non-linear function. Here are the five initialized variables:

$$H0 = 0x67452301$$

$$H1 = 0xEFCDAB89$$

$$H2 = 0x98BADCFE$$

$$H3 = 0x10325476$$

$$H4 = 0xC3D2E1F0$$

The constants used in process are stated as follows:

$$k_t = \begin{cases} 5a827999 & 0 \le t \le 19 \\ 6ed9eba1 & 20 \le t \le 39 \\ 8f1bbcdc & 40 \le t \le 59 \\ Ca62c1d6 & 60 \le t \le 79 \end{cases}$$

We can use these constants to judge whether one procedure has adopted SHA1 algorithm. After finishing initialization, we then start the main rotation of algorithm.

SHA1 algorithm consists of 6 tasks:

Task 1. Appending Padding Bits. The original message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. The padding rules are:

- The original message is always padded with one bit "1" first.
- Then zero or more bits "0" are padded to bring the length of the message up to 64 bits less than a multiple of 512.

Task 2. Appending Length. 64 bits are appended to the end of the padded message to indicate the length of the original message in bytes. The rules of appending length are:

- The length of the original message in bytes is converted to its binary format of 64 bits. If overflow happens, only the low-order 64 bits are used.
- Break the 64-bit length into 2 words (32 bits each).
- The low-order word is appended first and followed by the high-order word.

Task 3. Preparing Processing Functions. SHA1 requires 80 processing functions defined as:

$$f(t;B,C,D) = \begin{cases} (BandC)or((notB)andD) & 0 \le t \le 19 \\ BxorCxorD & 20 \le t \le 39 \\ (BandC)or(BandD)or(CandD) & 40 \le t \le 59 \\ BxorCxorD & 60 \le t \le 79 \end{cases}$$

Task 6. Processing Message in 512-bit Blocks. This is the main task of SHA1 algorithm, which loops through the padded and appended message in blocks of 512 bits each. For each input block, a number of operations are performed. This task can be described in the following pseudo code slightly modified from the RFC 3174's method 1:

Input and predefined functions:

```
  M[1, 2, ..., N]: Blocks of the padded and appended message

  f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): Defined as above

  K(0), K(1), ..., K(79): Defined as above

  H0, H1, H2, H3, H4: Word buffers with initial values
```

Algorithm:

```
  For loop on k = 1 to N

    (W(0),W(1),...,W(15)) = M[k] /* Divide M[k] into 16 words */

    For t = 16 to 79 do:

        W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)) <<< 1

    A = H0, B = H1, C = H2, D = H3, E = H4

    For t = 0 to 79 do:

        TEMP = A<<<5 + f(t;B,C,D) + E + W(t) + K(t)

        E = D, D = C, C = B<<<30, B = A, A = TEMP

    End of for loop
```

H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E

End of for loop

Output:

H0, H1, H2, H3, H4: Word buffers with final message digest

Implementation:

```
import hashlib

class SHA1:

    def __init__(self):

        self.__keys = ["Adnan is a human", "Humans are mamals", "Adnan is a mamal"]

        self.digest_keys = [self.digest(key) for key in self.__keys]

        for key, digest_key in zip(self.__keys,self.digest_keys):

            print(f"key = {key}\nEquivalent Sha1 hexadecimal digest = {digest_key}\n")

    def digest(self,string):

        return hashlib.sha1(string.encode()).hexdigest()

    def check_in_digest_keys(self,string):

        return self.digest(string) in self.digest_keys

    def digest_strings(self,strings):

        for x in strings:

            print(f"Given string: {x}\nSha1 hexadecimal digest value of given string: {self.digest(x)}")
```

```
        print(f"present in digest_keys {self.check_in_digest_keys(x)}\n")

obj = SHA1()

given_strings = [

            "Adnan is a human", "Adnan is not a human",

            "Humans are reptile", "Humans are mamals",

            "Adnan is a mamal", "Adnan is a reptile"

            ]

obj.digest_strings(given_strings)
```

Output:

```
key = Adnan is a human
Equivalent Sha1 hexadecimal digest = b534ccf13fdcdf68ef0ba785c891f3ce6b082ab6

key = Humans are mamals
Equivalent Sha1 hexadecimal digest = 01692710f02ef3368ff45b21c96975ac492dc059

key = Adnan is a mamal
Equivalent Sha1 hexadecimal digest = ecda2d5fcb7338c1418adef04862a16c05e70f27

Given string: Adnan is a human
Sha1 hexadecimal digest value of given string: b534ccf13fdcdf68ef0ba785c891f3ce6b082ab6
present in digest_keys True

Given string: Adnan is not a human
Sha1 hexadecimal digest value of given string: 220c34a00d4e3010da7de455d40682fc94cb549d
present in digest_keys False

Given string: Humans are reptile
Sha1 hexadecimal digest value of given string: 461b008a39d7f033f1f001851ee0bd1ae1c448f3
present in digest_keys False

Given string: Humans are mamals
Sha1 hexadecimal digest value of given string: 01692710f02ef3368ff45b21c96975ac492dc059
present in digest_keys True

Given string: Adnan is a mamal
Sha1 hexadecimal digest value of given string: ecda2d5fcb7338c1418adef04862a16c05e70f27
present in digest_keys True

Given string: Adnan is a reptile
Sha1 hexadecimal digest value of given string: aaa8c10433d9b74a0f64c66a4445be451f459574
present in digest_keys False
```