Name : Omkar Gurav
Roll no : 8048

Assignment No: 3

Title: Write a program in Java to implement SHA-1 algorithm using Libraries (API)

Objective: To study Message Digest of simple Plaintext.

Theory:

The **Secure Hash Algorithm** is one of a number of cryptographic hash functions published by the National Institute of Standards and Technology as a U.S. Federal Information Processing Standard:

SHA-1

A retronym applied to the original version of the 160-bit hash function published in 1993 under the name "SHA". It was withdrawn shortly after publication due to an undisclosed "significant flaw" and replaced by the slightly revised version SHA-1.
It is a 160-bit hash function which resembles the earlier MD5 algorithm. This was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm.

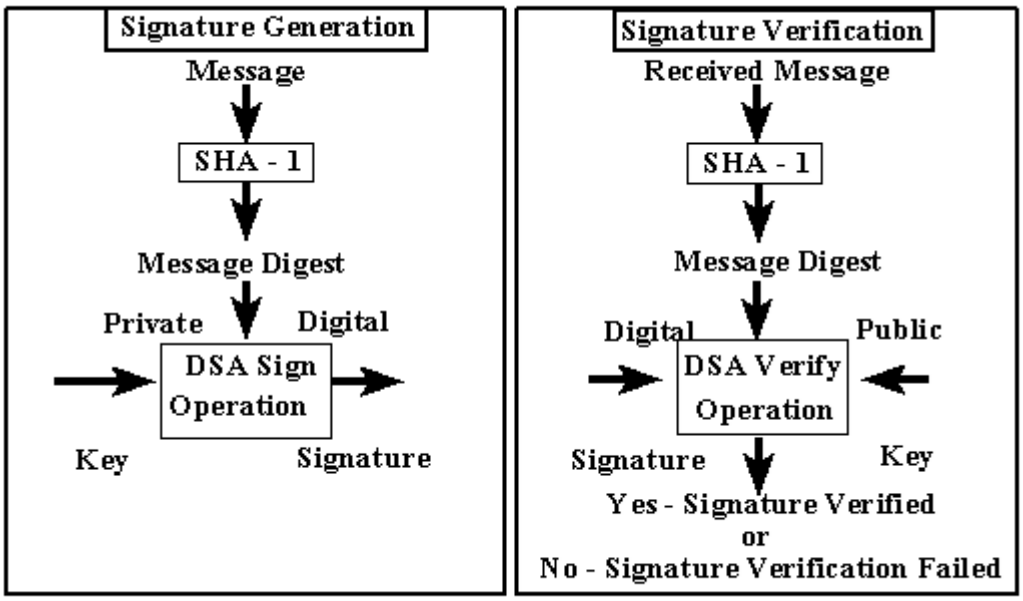

Figure 1: Using the SHA-1 with the DSA

Applications: The SHA-1 may be used with the DSA in electronic mail, electronic funds transfer, software distribution, data storage, and other applications which require data integrity assurance and data origin authentication. The SHA-1 may also be used whenever it is necessary to generate a condensed message.

Implementations: The SHA-1 may be implemented in software, firmware, hardware, or any combination thereof. Only implementations of the SHA-1 that are validated by NIST will be considered as complying with this standard.
Information about the requirements for validating implementations of this:

1. INTRODUCTION

The Secure Hash Algorithm (SHA-1) is required for use with the Digital Signature Algorithm (DSA) as specified in the Digital Signature Standard (DSS) and whenever a secure hash algorithm is required for federal applications. For a message of length < 2^64 bits, the SHA-1 produces a 160-bit condensed representation of the message called a message digest.

The message digest is used during generation of a signature for the message. The SHA-1 is also used to compute a message digest for the received version of the message during the process of verifying the signature. Any change to the message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.
The SHA-1 is designed to have the following properties: it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest.

2. BIT STRINGS AND INTEGERS

The following terminology related to bit strings and integers will be used:
a. A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}. A hex digit is the representation of a 4-bit string. **Examples:** 7 = 0111, A = 1010.

b. A word equals a 32-bit string which may be represented as a sequence of 8 hex digits. To convert a word to 8 hex digits each 4-bit string is converted to its hex equivalent as described in (a) above. **Example:**

1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

c. An integer between 0 and 2^{32} - 1 inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. **Example:** the integer 291 = $2^8+2^5+2^1+2^0$ = 256+32+2+1 is represented by the hex word, 00000123.

If z is an integer, 0 <= z < 2^{64}, then z = 2^{32}x + y where 0 <= x < 2^{32} and 0 <= y < 2^{32}. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X,Y).

d. block = 512-bit string. A block (e.g., B) may be represented as a sequence of 16 words

3. OPERATIONS ON WORDS

The following logical operators will be applied to words:
a. Bitwise logical word operations

X ^ Y = bitwise logical "and" of X and Y.

X ⋁ Y = bitwise logical "inclusive-or" of X and Y.

X XOR Y = bitwise logical "exclusive-or" of X and Y.

~ X = bitwise logical "complement" of X.

Example:

```
    01101100101110011101001001111011
XOR 01100101110000010110100110110111
    --------------------------------
```

= 00001001011110001011101111001100

b. The operation X + Y is defined as follows: words X and Y represent integers x and y, where $0 <= x < 2^{32}$ and $0 <= y < 2^{32}$. For positive integers n and m, let n mod m be the remainder upon dividing n by m. Compute

$z = (x + y) \bmod 2^{32}$.

Then $0 <= z < 2^{32}$. Convert z to a word, Z, and define Z = X + Y.

c. The circular left shift operation $S^n(X)$, where X is a word and n is an integer with $0 <= n < 32$, is defined by

$S^n(X) = (X << n) OR (X >> 32-n)$.

In the above, X << n is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus $S^n(X)$ is equivalent to a circular shift of X by n positions to the left.

4. MESSAGE PADDING

The SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512. The SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64-bit integer are appended to the end of the message to produce a padded message of length 512 * n. The 64-bit integer is l, the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.

Suppose a message has length $l < 2^{64}$. Before it is input to the SHA-1, the message is padded on the right as follows:

a. "1" is appended. **Example:** if the original message is "01010000", this is padded to "010100001".

b. "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Example: Suppose the original message is the bit string

01100001 01100010 01100011 01100100 01100101.

After step (a) this gives

01100001 01100010 01100011 01100100 01100101 1.

Since l = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)

61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000.

c. Obtain the 2-word representation of l, the number of bits in the original message. If $l < 2^{32}$ then the first word is all zeroes. Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then l = 40 (note that l is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex

61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028.

The padded message will contain 16 * n words for some n > 0. The padded message is regarded as a sequence of n blocks M_1 , M_2, ... , M_n, where each M_i contains 16 words and M_1 contains the first characters (or bits) of the message.

5. FUNCTIONS USED

A sequence of logical functions f_0, f_1,..., f_{79} is used in the SHA-1. Each f_t, 0 <= t <= 79, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f_t(B,C,D) is defined as follows: for words B, C, D,
f_t(B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)

f_t(B,C,D) = B XOR C XOR D (20 <= t <= 39)

f_t(B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <= 59)

f_t(B,C,D) = B XOR C XOR D (60 <= t <= 79).

6. CONSTANTS USED

A sequence of constant words K(0), K(1), ... , K(79) is used in the SHA-1. In hex these are given by
K = 5A827999 (0 <= t <= 19)

K_t = 6ED9EBA1 (20 <= t <= 39)

K_t = 8F1BBCDC (40 <= t <= 59)

K_t = CA62C1D6 (60 <= t <= 79).

7. COMPUTING THE MESSAGE DIGEST

The message digest is computed using the final padded message. The computation uses two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. The words of the first 5-word buffer are labeled A,B,C,D,E. The words of the second 5-word buffer are labeled H_0, H_1, H_2, H_3, H_4. The words of the 80-word sequence are labeled W_0, W_1,..., W_{79}. A single word buffer TEMP is also employed.
To generate the message digest, the 16-word blocks M_1, M_2,..., M_n defined in Section 4 are processed in order. The processing of each M_i involves 80 steps.
Before processing any blocks, the {H_i} are initialized as follows: in hex,
H_0 = 67452301

H_1 = EFCDAB89

H_2 = 98BADCFE

H_3 = 10325476

H_4 = C3D2E1F0.
Now M_1, M_2, ... , M_n are processed. To process M_i, we proceed as follows:
a. Divide M_i into 16 words W_0, W_1, ... , W_{15}, where W_0 is the left-most word.

b. For t = 16 to 79 let W_t = S^1(W_{t-3} XOR W_{t-8} XOR W_{t-14} XOR W_{t-16}).

c. Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.

d. For t = 0 to 79 do
$TEMP = S^5(A) + f_t(B,C,D) + E + W_t + K_t$;

$E = D$; $D = C$; $C = S^{30}(B)$; $B = A$; $A = TEMP$;

e. Let $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.
After processing M_n, the message digest is the 160-bit string represented by the 5 words
H_0 H_1 H_2 H_3 H_4.

Steps to implement SHA1 using JAVA API.

➢ Sun's implementation of SHA1 can be accessed through a generic class called MessageDigest. Here are the main methods of MessageDigest class:

- getInstance("SHA1") - Returns a message digest object represents a specific implementation of SHA1 algorithm from the default provider, Sun.
- getProvider() - Returns the provider name of the current object.
- update(bytes) - Updates the input message by appending a byte array at the end.
- digest() - Performs SHA1 algorithm on the current input message and returns the message digest as a byte array. This method also resets the input message to an empty byte string.
- reset() - Resets the input message to an empty byte string.

Input:

- Message

Output:

- Message Digest in Hexadecimal format
- Message Digest in Binary format

Algorithm:

1. Start
2. Input message.
3. Initialize instance of SHA-1.
4. Calculate message digest and store it in byte array.
5. Convert byte array into signum representation.

6. Convert message digest into hex value.
7. Add preceding 0s to make it 32 bit.
8. Covert message digest into binary value.
9. Add preceding 0s to make it 32 bit.
10. Show hexadecimal value.
11. Show binary value.
12. Stop.

Conclusion: We have implemented SHA-1 algorithm using Libraries JAVA (API).

SHA.java

```
// Importing classes
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;

public class SHA {

	public static void encrypt_string(String input)
	{
		try {

			MessageDigest md = MessageDigest.getInstance("SHA-1");

			// Calculating message digest and storing it in byte array
			byte[] messageDigest = md.digest(input.getBytes());

			// Converting byte array into signum representation
```

```
                BigInteger no = new BigInteger(1, messageDigest);

                // Converting message digest into hex value
                String hashtext = no.toString(16);

                // Adding preceding 0s to make it 32 bit
                while (hashtext.length() < 32) {
                        hashtext = "0" + hashtext;
                }

                // Converting message digest into binary
                String binarytext = no.toString(2);

                // Adding preceding 0s to make it 32 bit
                while (binarytext.length() < 32) {
                        binarytext = "0" + binarytext;
                }

                System.out.println("\nMessage Digest in Hexadecimal format : " +
hashtext);

                System.out.println("\nMessage Digest in Binary format : " +
binarytext);

        }

        catch (NoSuchAlgorithmException e) {
                throw new RuntimeException(e);
```

```
        }
    }

    public static void main(String args[]) throws
                                                    NoSuchAlgorithmException
    {

        System.out.println("Enter message : ");

        Scanner sc = new Scanner(System.in);

        String msg = sc.nextLine();

        encrypt_string(msg);

        sc.close();
    }
}
```

Output :

Enter message :
PUNE

Message Digest in Hexadecimal format : c03c7f2b3126c7b25ea179d1c2f1eaad1d299008

Message Digest in Binary format :
11000000001111000111111100101011001100010010011011000111101100100101111010100001011110
01110100011100001011110001111010101010110100011101001010011001000000001000