St. Francis Institute of Technology, Mumbai-400 103
Department Of Information Technology

A.Y. 2025-2026
Class: BE-IT A/B, Semester: VII

Subject: Secure Application Development Lab

Student Name: Charis Fernandes Student Roll No: 26

Experiment – 10 : Study and Implement Hashing Algorithm

Aim: To study and implement hashing algorithm
Objectives: After study of this experiment, the student will be able to

- understand role of hashing function in secure coding
- understand the concept of hashing function
- understand practical applications of hashing algorithm

Lab objective mapped: ITL703.6 To **apply** secure coding for cryptography

Prerequisite: Basic knowledge of hashing function in cryptography.

Requirements: C/C++/JAVA/PYTHON

Pre-Experiment Theory:

Hash Function is a function that has a huge role in making a System Secure as it converts normal data given to it as an irregular value of fixed length. When we put data into this function it outputs an irregular value. The Irregular value it outputs is known as **"Hash Value"**.
Hash Values are simply numbers but are often written in Hexadecimal. Computers manage values as Binary. The hash value is also data and is often managed in Binary. A hash function is basically performing some calculations in the computer. Data values that are its output are of fixed length. Length always varies according to the hash function. Value doesn't vary even if there is a large or small value.
If given the same input, two hash functions will invariably produce the same output. Even if input data entered differs by a single bit, huge change in their output values. Even if input data entered differs huge, there is a minimal chance that the hash values produced will be identical. If they are equal, it is known as **"Hash Collision".** Converting Hash Codes to their original value is an impossible task to perform. This is the main difference between Encryption as a Hash Function.

Features of hash functions in system security:

One-way function: Hash functions are designed to be one-way functions, meaning that it is easy to compute the hash value for a given input, but difficult to compute the input for a given hash value. This property makes hash functions useful for verifying the integrity of data, as any changes to the data will result in a different hash value.

Deterministic: Hash functions are deterministic, meaning that given the same input, the output will always be the same. This makes hash functions useful for verifying the authenticity of data, as any changes to the data will result in a different hash value.
Fixed-size output: Hash functions produce a fixed-size output, regardless of the size of the input. This property makes hash functions useful for storing and transmitting data, as the hash value can be stored or transmitted more efficiently than the original data.
Collision resistance: Hash functions should be designed to be collision resistant, meaning that it is difficult to find two different inputs that produce the same hash value. This property ensures that attackers cannot create a false message that has the same hash value as a legitimate message.
Non-reversible: Hash functions are non-reversible, meaning that it is difficult or impossible to reverse the process of generating a hash value to recover the original input. This property makes hash functions useful for storing passwords or other sensitive information, as the original input cannot be recovered from the hash value.

Advantages:

Data integrity: Hash functions are useful for ensuring the integrity of data, as any changes to the data will result in a different hash value. This property makes hash functions a valuable tool for detecting data tampering or corruption.
Message authentication: Hash functions are useful for verifying the authenticity of messages, as any changes to the message will result in a different hash value. This property makes hash functions a valuable tool for verifying the source of a message and detecting message tampering.
Password storage: Hash functions are useful for storing passwords in a secure manner. Hashing the password ensures that the original password cannot be recovered from the hash value, making it more difficult for attackers to access user accounts.
Fast computation: Hash functions are designed to be fast to compute, making them useful for a variety of applications where efficiency is important.

Disadvantages:

Collision attacks: Hash functions are vulnerable to collision attacks, where an attacker tries to find two different inputs that produce the same hash value. This can compromise the security of hash-based protocols, such as digital signatures or message authentication codes.
Rainbow table attacks: Hash functions are vulnerable to rainbow table attacks, where an attacker precomputes a table of hash values and their corresponding inputs, making it easier to crack password hashes.
Hash function weaknesses: Some hash functions have known weaknesses, such as the MD5 hash function, which is vulnerable to collision attacks. It is important to choose a hash function that is secure for the intended application.
Limited input size: Hash functions produce a fixed-size output, regardless of the size of the input. This can lead to collisions if the input size is larger than the hash function output size.

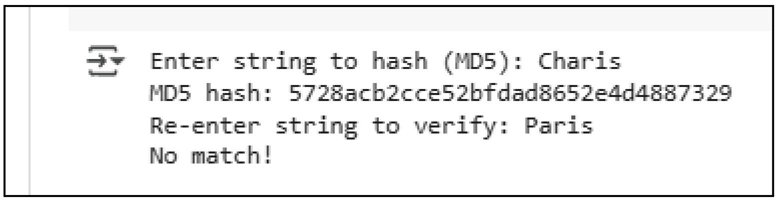
Procedure:

MD5- HASHING

```
import struct
import math
```

```
def left_rotate(x, c):
    return ((x << c) | (x >> (32 - c))) & 0xFFFFFFFF

def md5(message):
    # Initialize variables:
    A = 0x67452301
    B = 0xefcdab89
    C = 0x98badcfe
    D = 0x10325476

    # Preprocessing:
    msg = bytearray(message, 'utf-8')
    orig_len_bits = (8 * len(msg)) & 0xffffffffffffffff
    msg.append(0x80)
    while (len(msg) * 8) % 512 != 448:
        msg.append(0)
    msg += struct.pack('<Q', orig_len_bits)

    # Constants for MD5
    s = [7,12,17,22]*4 + [5,9,14,20]*4 + [4,11,16,23]*4 + [6,10,15,21]*4
    K = [int(abs(math.sin(i + 1)) * (2**32)) & 0xFFFFFFFF for i in range(64)]

    for offset in range(0, len(msg), 64):
        chunk = msg[offset:offset+64]
        M = list(struct.unpack('<16I', chunk))
        a, b, c, d = A, B, C, D

        for i in range(64):
            if i < 16:
                f = (b & c) | (~b & d)
                g = i
            elif i < 32:
                f = (d & b) | (~d & c)
                g = (5*i + 1) % 16
            elif i < 48:
                f = b ^ c ^ d
                g = (3*i + 5) % 16
            else:
                f = c ^ (b | ~d)
                g = (7*i) % 16
            f = (f + a + K[i] + M[g]) & 0xFFFFFFFF
            a, d, c, b = d, c, b, (b + left_rotate(f, s[i])) & 0xFFFFFFFF

        A = (A + a) & 0xFFFFFFFF
        B = (B + b) & 0xFFFFFFFF
        C = (C + c) & 0xFFFFFFFF
        D = (D + d) & 0xFFFFFFFF

    return ''.join(f'{x:02x}' for x in struct.pack('<4I', A, B, C, D))
```

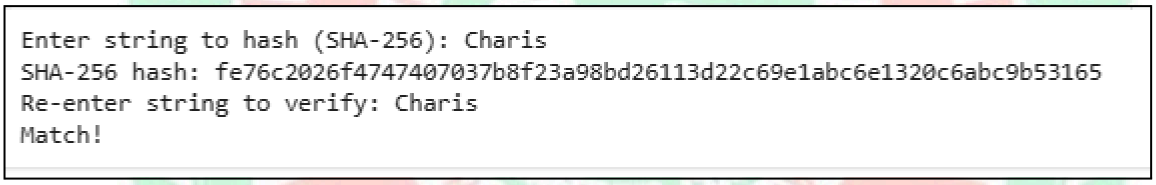
```
# Simple verification logic
original = input("Enter string to hash (MD5): ")
hash1 = md5(original)
print("MD5 hash:", hash1)

verify = input("Re-enter string to verify: ")
hash2 = md5(verify)
print("Match!" if hash1 == hash2 else "No match!")
```



```
Enter string to hash (MD5): Charis
MD5 hash: 5728acb2cce52bfdad8652e4d4887329
Re-enter string to verify: Paris
No match!
```

SHA- HASHING

```
import struct

def right_rotate(x, n):
    return (x >> n) | ((x << (32 - n)) & 0xFFFFFFFF)

def sha256(message):
    # Initial hash values (first 32 bits of fractional sqrt primes)
    H = [
        0x6a09e667, 0xbb67ae85, 0x3c6ef372,
        0xa54ff53a, 0x510e527f, 0x9b05688c,
        0x1f83d9ab, 0x5be0cd19,
    ]

    K = [
        0x428a2f98, 0x71374491, 0xb5c0fbcf,
        0xe9b5dba5, 0x3956c25b, 0x59f111f1,
        0x923f82a4, 0xab1c5ed5, 0xd807aa98,
        0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
        0xc19bf174, 0xe49b69c1, 0xefbe4786,
        0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
        0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8,
        0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
        0x06ca6351, 0x14292967, 0x27b70a85,
        0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e,
        0x92722c85, 0xa2bfe8a1, 0xa81a664b,
        0xc24b8b70, 0xc76c51a3, 0xd192e819,
    0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c,
    0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
    0x5b9cca4f, 0x682e6ff3, 0x748f82ee,
    0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7,
    0xc67178f2,
]

msg = bytearray(message, 'utf-8')
orig_len_bits = (8 * len(msg)) & 0xffffffffffffffff
msg.append(0x80)
while (len(msg)*8) % 512 != 448:
    msg.append(0)
msg += struct.pack('>Q', orig_len_bits)

for offset in range(0, len(msg), 64):
    chunk = msg[offset:offset+64]
    w = list(struct.unpack('>16L', chunk)) + [0]*48

    for i in range(16, 64):
        s0 = right_rotate(w[i-15],7) ^ right_rotate(w[i-15],18) ^ (w[i-15] >> 3)
        s1 = right_rotate(w[i-2],17) ^ right_rotate(w[i-2],19) ^ (w[i-2] >> 10)
        w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFF

    a,b,c,d,e,f,g,h = H

    for i in range(64):
        S1 = right_rotate(e,6) ^ right_rotate(e,11) ^ right_rotate(e,25)
        ch = (e & f) ^ ((~e) & g)
        temp1 = (h + S1 + ch + K[i] + w[i]) & 0xFFFFFFFF
        S0 = right_rotate(a,2) ^ right_rotate(a,13) ^ right_rotate(a,22)
        maj = (a & b) ^ (a & c) ^ (b & c)
        temp2 = (S0 + maj) & 0xFFFFFFFF

        h = g
        g = f
        f = e
        e = (d + temp1) & 0xFFFFFFFF
        d = c
        c = b
        b = a
        a = (temp1 + temp2) & 0xFFFFFFFF

    H = [
        (H[0]+a) & 0xFFFFFFFF,
        (H[1]+b) & 0xFFFFFFFF,
        (H[2]+c) & 0xFFFFFFFF,
        (H[3]+d) & 0xFFFFFFFF,
            (H[4]+e) & 0xFFFFFFFF,
            (H[5]+f) & 0xFFFFFFFF,
            (H[6]+g) & 0xFFFFFFFF,
            (H[7]+h) & 0xFFFFFFFF,
        ]

    return ''.join(f'{x:08x}' for x in H)

# Simple verification logic
original = input("Enter string to hash (SHA-256): ")
hash1 = sha256(original)
print("SHA-256 hash:", hash1)

verify = input("Re-enter string to verify: ")
hash2 = sha256(verify)
print("Match!" if hash1 == hash2 else "No match!")
```



```
Enter string to hash (SHA-256): Charis
SHA-256 hash: fe76c2026f4747407037b8f23a98bd26113d22c69e1abc6e1320c6abc9b53165
Re-enter string to verify: Charis
Match!
```

Post-Experiments Exercise:

Extended Theory: Nil

Post Experimental Exercise:

Questions:
- Explain the difference between hashing and encryption. In what scenarios would you use one over the other, and why?
- Provide examples of practical applications of hashing functions in computer science beyond data integrity and password storage.
- Briefly explain how hashing functions are used in the creation and verification of digital signatures. Why is this important in cybersecurity and secure communication?
- What is the purpose of salting passwords before hashing them?

Conclusion:
- Write what was performed in the experiment.
- Write the significance of the topic studied in the experiment.

References:
http://cse29-iiith.vlabs.ac.in/