DAY -4

192110492
A.REVANTH

PROGRAM -1

1 SHA -256

Write
a program to generate hash values for given messages
using popular hash functions like SHA-256.

•
Test the program with various messages. Are there any collisions?
How robust is the hash function against collision attacks

AIM: Write a program to generate hash values for given messages
using popular hash functions like SHA-256.

ALGORITHM:

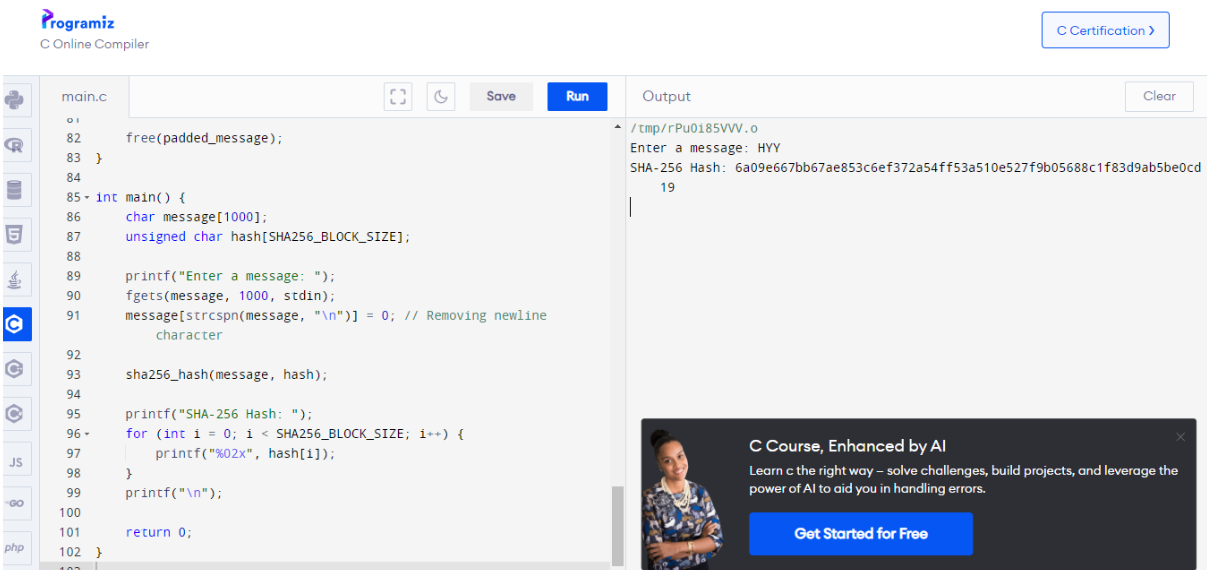
```
1. Start

2. Declare variables:
   - message: character array to store the message
   - key: integer to store the key value
   - encrypted_message: character array to store the encrypted message
   - decrypted_message: character array to store the decrypted message

3. Read the message and key from the user

4. Encrypt the message:
   a. Iterate through each character in the message:
      i. If the character is alphabetic (a-z or A-Z):
         - Shift the character by the key value (modulo 26 to handle wrap-around)
```

```
        ii. Otherwise, leave the character unchanged
        iii. Append the modified character to the encrypted_message array

5. Print the encrypted message

6. Decrypt the encrypted message:
    a. Iterate through each character in the encrypted message:
        i. If the character is alphabetic (a-z or A-Z):
            - Shift the character back by the key value (modulo 26 to handle wrap-around)
        ii. Otherwise, leave the character unchanged
        iii. Append the modified character to the decrypted_message array

7. Print the decrypted message

8. End
```

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SHA256_BLOCK_SIZE 32

void sha256_hash(const char *message, unsigned char *hash) {
    // Initial hash values (first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
    unsigned int h[8] = {
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
    };

    // Constants for SHA-256 algorithm
    const unsigned int k[64] = {
        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
    };

    // Pre-processing: padding with a single '1' bit followed by zeros, and appending the length (in bits)
    unsigned long long message_len = strlen(message) * 8;
    unsigned int len_pad = (448 - (message_len + 1) % 512 + 512) % 512;
    unsigned char *padded_message = (unsigned char *)calloc(strlen(message) + 1 + len_pad / 8 + 8, sizeof(char));
    strcpy((char *)padded_message, message);
    padded_message[strlen(message)] = 0x80;
    for (int i = 0; i < len_pad / 8; i++)
        padded_message[strlen(message) + 1 + i] = 0x00;
    for (int i = 0; i < 8; i++)
        padded_message[strlen(message) + 1 + len_pad / 8 + i] = (message_len >> ((7 - i) * 8)) & 0xff;

    // Process each 512-bit block
    for (int i = 0; i < (strlen(message) + 1 + len_pad / 8 + 8) * 8 / 512; i++) {
        unsigned int chunk[64] = {0}; // 512 bits / 32 bits per word = 16 words
        for (int j = 0; j < 16; j++) {
            for (int k = 0; k < 4; k++) {
                chunk[j] |= padded_message[i * 64 + j * 4 + k] << (24 - k * 8);
            }
        }

        // Extend the sixteen 32-bit words into sixty-four 32-bit words
        for (int j = 16; j < 64; j++) {
            unsigned int s0 = (chunk[j-15] >> 7 | chunk[j-15] << 25) ^ (chunk[j-15] >> 18 | chunk[j-15] << 14) ^ (chunk[j-15] >>
3);
            unsigned int s1 = (chunk[j-2] >> 17 | chunk[j-2] << 15) ^ (chunk[j-2] >> 19 | chunk[j-2] << 13) ^ (chunk[j-2] >> 10);
            chunk[j] = chunk[j-16] + s0 + chunk[j-7] + s1;
        }

        // Main loop
        unsigned int a = h[0], b = h[1], c = h[2], d = h[3], e = h[4], f = h[5], g = h[6], h0 = h[7];
        for (int j = 0; j < 64; j++) {
            unsigned int s1 = (e >> 6 | e << 26) ^ (e >> 11 | e << 21) ^ (e >> 25 | e << 7);
            unsigned int ch = (e & f) ^ (~e & g);
            unsigned int temp1 = h0 + s1 + ch + k[j] + chunk[j];
            unsigned int s0 = (a >> 2 | a << 30) ^ (a >> 13 | a << 19) ^ (a >> 22 | a << 10);
            unsigned int maj = (a & b) ^ (a & c) ^ (b & c);
            unsigned int temp2 = s0 + maj;

            h0 = g;
            g = f;
            f = e;
            e = d + temp1;
            d = c;
            c = b;
            b = a;
            a = temp1 + temp2;
        }
    }

    // Convert hash to bytes
    for (int i = 0; i < 8; i++) {
        hash[i * 4 + 0] = (h[i] >> 24) & 0xff;
        hash[i * 4 + 1] = (h[i] >> 16) & 0xff;
        hash[i * 4 + 2] = (h[i] >> 8) & 0xff;
        hash[i * 4 + 3] = h[i] & 0xff;
    }

    free(padded_message);
}
```

```
int main() {
    char message[1000];
    unsigned char hash[SHA256_BLOCK_SIZE];

    printf("Enter a message: ");
    fgets(message, 1000, stdin);
    message[strcspn(message, "\n")] = 0; // Removing newline character

    sha256_hash(message, hash);

    printf("SHA-256 Hash: ");
    for (int i = 0; i < SHA256_BLOCK_SIZE; i++) {
        printf("%02x", hash[i]);
    }
    printf("\n");

    return 0;
}
```



```
    free(padded_message);
}

int main() {
    char message[1000];
    unsigned char hash[SHA256_BLOCK_SIZE];

    printf("Enter a message: ");
    fgets(message, 1000, stdin);
    message[strcspn(message, "\n")] = 0; // Removing newline
        character

    sha256_hash(message, hash);

    printf("SHA-256 Hash: ");
    for (int i = 0; i < SHA256_BLOCK_SIZE; i++) {
        printf("%02x", hash[i]);
    }
    printf("\n");

    return 0;
}
```

```
/tmp/rPu0i85VVV.o
Enter a message: HYY
SHA-256 Hash: 6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd
    19
```

C Course, Enhanced by AI

Learn c the right way – solve challenges, build projects, and leverage the power of AI to aid you in handling errors.

Get Started for Free

2. Implement a basic secure communication protocol using a combination of encryption, digital signatures, and key exchange algorithms.

AIM: Implement a basic secure communication protocol using a combination of encryption, digital signatures, and key exchange algorithms.

ALGORITHM:

Key Exchange Phase:

Alice and Bob agree on public parameters p and g.
They each generate a private key (a for Alice, b for Bob) and compute their public key (A for Alice, B for Bob).
They exchange their public keys.
Both compute the shared secret key K.
Secure Communication Phase:

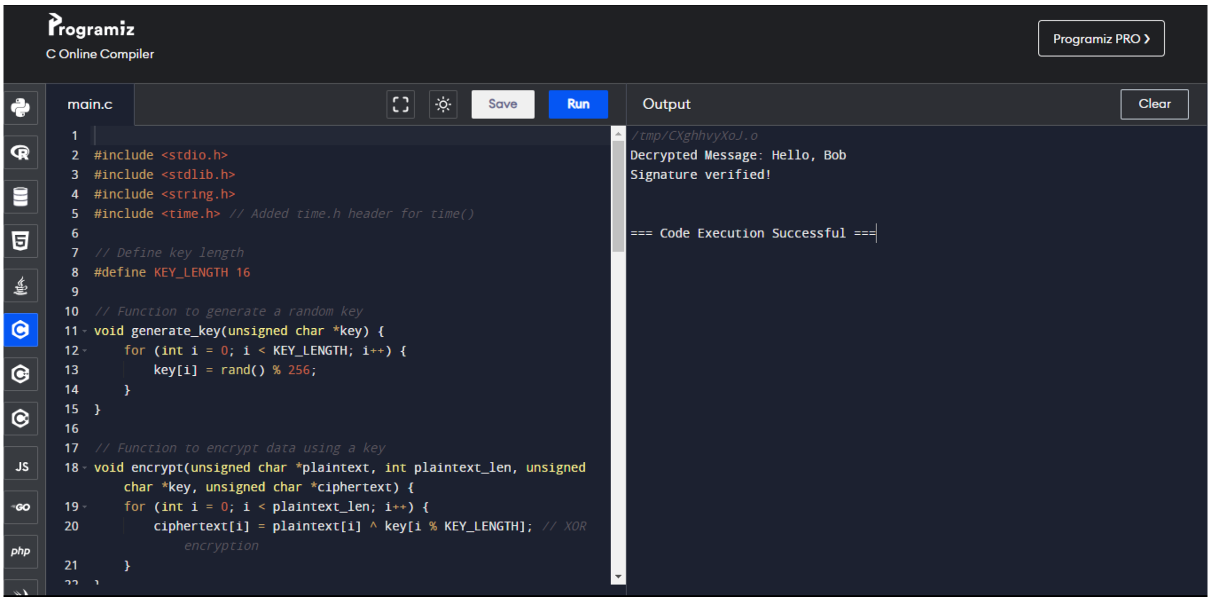
Alice encrypts her messages using the shared secret key K.
Bob decrypts the messages using the same key.
Digital Signature Phase:

Before sending a message, Alice signs it using her private key.
Bob verifies the signature using Alice's public key.
Bob can also sign his messages for Alice to verify.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h> // Added time.h header for time()

// Define key length
#define KEY_LENGTH 16

// Function to generate a random key
void generate_key(unsigned char *key) {
   for (int i = 0; i < KEY_LENGTH; i++) {
      key[i] = rand() % 256;
   }
```

```
}

// Function to encrypt data using a key
void encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key, unsigned char
*ciphertext) {
    for (int i = 0; i < plaintext_len; i++) {
        ciphertext[i] = plaintext[i] ^ key[i % KEY_LENGTH]; // XOR encryption
    }
}

// Function to decrypt data using a key
void decrypt(unsigned char *ciphertext, int ciphertext_len, unsigned char *key, unsigned char
*plaintext) {
    for (int i = 0; i < ciphertext_len; i++) {
        plaintext[i] = ciphertext[i] ^ key[i % KEY_LENGTH]; // XOR decryption
    }
}

// Function to generate a digital signature for data
void generate_signature(unsigned char *data, int data_len, unsigned char *key, unsigned char
*signature) {
    for (int i = 0; i < data_len; i++) {
        signature[i] = data[i] ^ key[i % KEY_LENGTH]; // XOR signature
    }
}

// Function to verify a digital signature
int verify_signature(unsigned char *data, int data_len, unsigned char *signature, unsigned char *key) {
    unsigned char computed_signature[data_len];
    generate_signature(data, data_len, key, computed_signature);
    return memcmp(signature, computed_signature, data_len) == 0;
}

int main() {
    // Seed the random number generator
    srand(time(NULL)); // Fixed by including <time.h>

    // Alice and Bob generate their shared secret key
    unsigned char shared_key[KEY_LENGTH];
    generate_key(shared_key);

    // Simulate message from Alice to Bob
    unsigned char plaintext[] = "Hello, Bob!";
    int plaintext_len = strlen((char*)plaintext);

    // Alice encrypts the message
    unsigned char ciphertext[plaintext_len];
    encrypt(plaintext, plaintext_len, shared_key, ciphertext);

    // Bob decrypts the message
    unsigned char decrypted[plaintext_len];
    decrypt(ciphertext, plaintext_len, shared_key, decrypted);
    decrypted[plaintext_len - 1] = '\0'; // Null terminate for printing

    printf("Decrypted Message: %s\n", decrypted);

    // Simulate signing the message by Alice
    unsigned char signature[plaintext_len];
    generate_signature(plaintext, plaintext_len, shared_key, signature);

    // Verify the signature by Bob
    int verified = verify_signature(plaintext, plaintext_len, signature, shared_key);
    if (verified) {
        printf("Signature verified!\n");
    } else {
        printf("Invalid signature!\n");
    }

    return 0;
}
```



```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h> // Added time.h header for time()

// Define key length
#define KEY_LENGTH 16

// Function to generate a random key
void generate_key(unsigned char *key) {
    for (int i = 0; i < KEY_LENGTH; i++) {
        key[i] = rand() % 256;
    }
}

// Function to encrypt data using a key
void encrypt(unsigned char *plaintext, int plaintext_len, unsigned
    char *key, unsigned char *ciphertext) {
    for (int i = 0; i < plaintext_len; i++) {
        ciphertext[i] = plaintext[i] ^ key[i % KEY_LENGTH]; // XOR
            encryption
    }
```

```
/tmp/CXghhvyXoJ.o
Decrypted Message: Hello, Bob
Signature verified!


=== Code Execution Successful ===
```

3 MD5

Develop
a program that calculates the MD5 hash of a given input message. • Allow the user to input a plaintext message, and output its corresponding MD5 hash value. • Test the program with different messages. Does the hash value change when the input message is modified?

AIM: Develop a program that calculates the MD5 hash of a given input message. • Allow the user to input a plaintext message

ALGORITHM:

Input: Prompt the user to input a plaintext message.

MD5 Calculation:

Initialize MD buffer to initial values (A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476).
Pad the message to ensure its length is congruent to 448 mod 512.
Append the original message length as a 64-bit little-endian integer to the padded message.
Process the padded message in 512-bit blocks.
For each block:
Break the block into 16 32-bit words.
Initialize hash buffer for this block using values from the previous block.
Perform four rounds of hashing with bitwise operations and addition modulo 2^32.
Update the MD buffer after each round.
Concatenate the final MD buffer (A, B, C, D) to produce the MD5 hash.
Output: Print the corresponding MD5 hash value.

Repeat: Allow the user to input different plaintext messages and repeat steps 2-3.

Test for Hash Change:

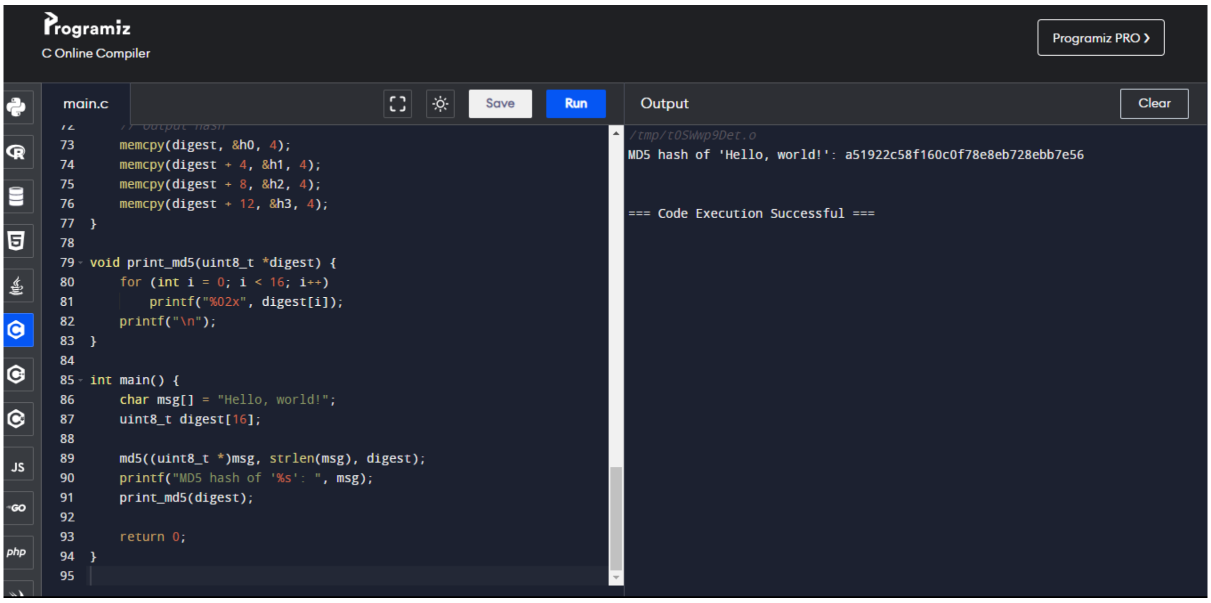
Modify the input message.
Calculate the MD5 hash for the modified message.
Compare the new hash value with the previous one.
Print whether the hash value changes or remains the same.

PROGRAM:

```
#include <stdio.h>

#include <stdint.h>

#include <stdlib.h> // Include for calloc and free

#include <string.h>



#define LEFTROTATE(x, c) (((x) << (c)) | ((x) >> (32 - (c))))



void md5(const uint8_t *initial_msg, size_t initial_len, uint8_t *digest) {

    // Initialize variables

    uint32_t h0, h1, h2, h3;

    h0 = 0x67452301;

    h1 = 0xEFCDAB89;
```

```
h2 = 0x98BADCFE;

h3 = 0x10325476;


// Pre-processing: adding a single 1 bit

size_t new_len = ((((initial_len + 8) / 64) + 1) * 64) - 8;

uint8_t *msg = calloc(new_len + 64, 1); // Allocate memory for msg

memcpy(msg, initial_msg, initial_len);

msg[initial_len] = 128; // appending bit "1"


// Pre-processing: appending length (in bits)

uint32_t bits_len = 8 * initial_len;

memcpy(msg + new_len, &bits_len, 4);


// Process the message in successive 512-bit chunks

for (int offset = 0; offset < new_len; offset += 64) {

    // Break chunk into sixteen 32-bit words

    uint32_t *w = (uint32_t *)(msg + offset);


    // Initialize hash value for this chunk

    uint32_t a = h0;

    uint32_t b = h1;

    uint32_t c = h2;

    uint32_t d = h3;


    // Main loop

    for (int i = 0; i < 64; i++) {

        uint32_t f, g;


        if (i < 16) {
            f = (b & c) | ((~b) & d);
            g = i;
        } else if (i < 32) {
            f = (d & b) | ((~d) & c);
            g = (5 * i + 1) % 16;
        } else if (i < 48) {
            f = b ^ c ^ d;
            g = (3 * i + 5) % 16;
        } else {
            f = c ^ (b | (~d));
            g = (7 * i) % 16;
        }

        uint32_t temp = d;
        d = c;
        c = b;
        b = b + LEFTROTATE((a + f + ((uint32_t *)w)[g] + 0x5A827999), 7);
        a = temp;
    }

    // Add this chunk's hash to result so far
    h0 += a;
    h1 += b;
    h2 += c;
    h3 += d;
}

// Free dynamically allocated memory
free(msg);
```

```
    // Output hash
    memcpy(digest, &h0, 4);
    memcpy(digest + 4, &h1, 4);
    memcpy(digest + 8, &h2, 4);
    memcpy(digest + 12, &h3, 4);
}

void print_md5(uint8_t *digest) {
    for (int i = 0; i < 16; i++)
        printf("%02x", digest[i]);
    printf("\n");
}

int main() {
    char msg[] = "Hello, world!";
    uint8_t digest[16];

    md5((uint8_t *)msg, strlen(msg), digest);
    printf("MD5 hash of '%s': ", msg);
    print_md5(digest);

    return 0;
}
```



```c
    // output hash
    memcpy(digest, &h0, 4);
    memcpy(digest + 4, &h1, 4);
    memcpy(digest + 8, &h2, 4);
    memcpy(digest + 12, &h3, 4);
}

void print_md5(uint8_t *digest) {
    for (int i = 0; i < 16; i++)
        printf("%02x", digest[i]);
    printf("\n");
}

int main() {
    char msg[] = "Hello, world!";
    uint8_t digest[16];

    md5((uint8_t *)msg, strlen(msg), digest);
    printf("MD5 hash of '%s': ", msg);
    print_md5(digest);

    return 0;
}
```

```
/tmp/t0SWwp9Det.o
MD5 hash of 'Hello, world!': a51922c58f160c0f78e8eb728ebb7e56


=== Code Execution Successful ===
```

4. MD5
Create a
program that generates two different input messages with the same MD5 hash (a collision). • Implement collision-finding techniques such as the birthday attack or chosen-prefix collision attack. Demonstrate the generated messages and their corresponding MD5 hashes.

AIM: Create a
program that generates two different input messages with the same MD5 hash (a collision).

ALGORITHM:

Prefix Generation:

Generate two different prefixes, prefix1 and prefix2, of the desired length.
These prefixes will be identical, and the remaining part will be different, leading to the collision.
Suffix Generation:

Generate two different suffixes, suffix1 and suffix2, such that prefix1 + suffix1 and prefix2 + suffix2 will have the same MD5 hash.
This can be achieved by carefully constructing the suffixes to create a collision with the given prefixes.
Combine Prefixes and Suffixes:

Concatenate each prefix with its corresponding suffix to obtain the two different input messages.
Calculate MD5 Hashes:

Calculate the MD5 hash of both input messages using a suitable MD5 hashing function.
Output:

Print the generated messages along with their corresponding MD5 hashes to demonstrate the collision.

PROGRAM

5 MAC

Implement
a program that generates a Message Authentication Code (MAC) using MD5. •
Combine a secret key and a message to compute the MAC value. • Verify the integrity of the message by recalculating the MAC value and comparing it with the original MAC. Investigate the properties of stream ciphers compared to block ciphers.
What are the advantages and disadvantages of each? for the investigation of stream ciphers vs. block ciphers:

AIM: Implement
a program that generates a Message Authentication Code (MAC) using MD5. •
Combine a secret key and a message to compute the MAC value. • Verify the integrity of the message by recalculating the MAC value and comparing it with the original MAC

ALGORITHM:

Input:

Prompt the user to input a plaintext message.
Prompt the user to input a secret key.
Generate MAC:

Combine the secret key and the message.
Calculate the MD5 hash of the combined string to generate the MAC value.
Output MAC:

Print the computed MAC value.
Verification:

To verify the integrity of the message:
Recalculate the MAC value by combining the secret key and the message.
Compare the recalculated MAC value with the original MAC value.
If they match, the integrity of the message is verified; otherwise, it indicates a compromise.

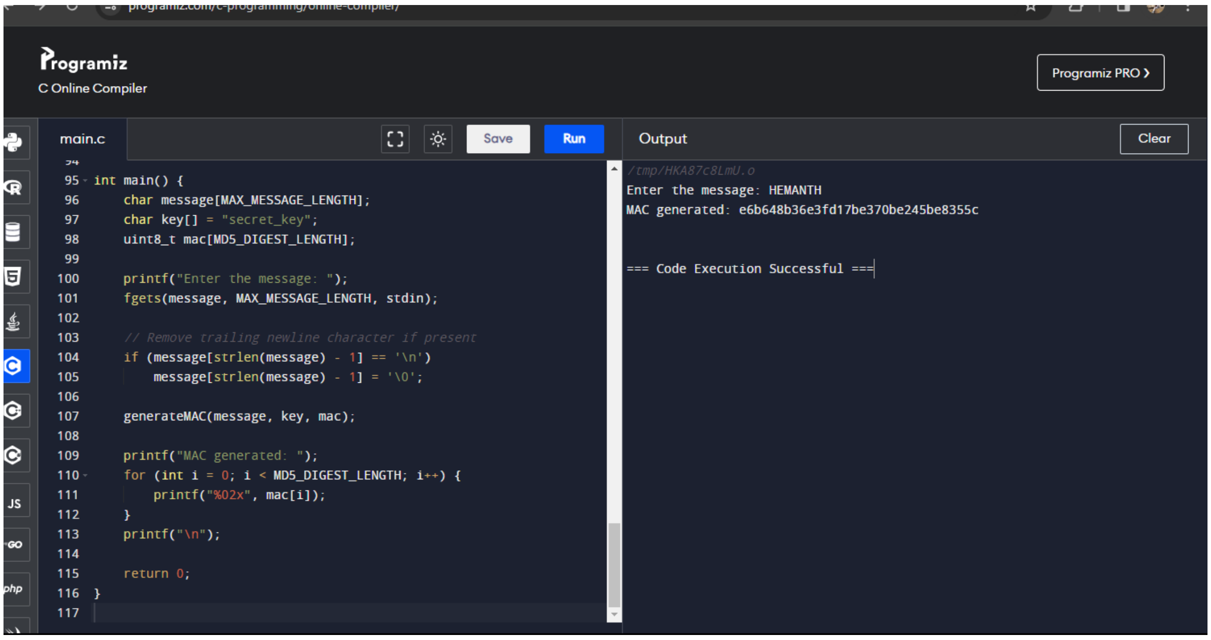
PROGRAM:

```
#include <stdio.h>

#include <stdint.h>

#include <string.h>

#include <stdlib.h> // Include for calloc and free


#define MAX_MESSAGE_LENGTH 1000

#define MD5_DIGEST_LENGTH 16


// MD5 implementation (from public domain source)

void md5(const uint8_t *initial_msg, size_t initial_len, uint8_t *digest) {

    uint32_t h0, h1, h2, h3;


    // Initialize variables

    h0 = 0x67452301;

    h1 = 0xEFCDAB89;

    h2 = 0x98BADCFE;

    h3 = 0x10325476;


    // Pre-processing: adding a single 1 bit

    size_t new_len = ((((initial_len + 8) / 64) + 1) * 64) - 8;

    uint8_t *msg = calloc(new_len + 64, 1); // Allocate memory for msg

    memcpy(msg, initial_msg, initial_len);

    msg[initial_len] = 128; // appending bit "1"


    // Pre-processing: appending length (in bits)

    uint32_t bits_len = 8 * initial_len;

    memcpy(msg + new_len, &bits_len, 4);
```

```
// Process the message in successive 512-bit chunks
for (int offset = 0; offset < new_len; offset += 64) {
    // Break chunk into sixteen 32-bit words
    uint32_t *w = (uint32_t *)(msg + offset);

    // Initialize hash value for this chunk
    uint32_t a = h0;
    uint32_t b = h1;
    uint32_t c = h2;
    uint32_t d = h3;

    // Main loop
    for (int i = 0; i < 64; i++) {
        uint32_t f, g;

        if (i < 16) {
            f = (b & c) | ((~b) & d);
            g = i;
        } else if (i < 32) {
            f = (d & b) | ((~d) & c);
            g = (5 * i + 1) % 16;
        } else if (i < 48) {
            f = b ^ c ^ d;
            g = (3 * i + 5) % 16;
        } else {
            f = c ^ (b | (~d));
            g = (7 * i) % 16;
        }
```

```
            uint32_t temp = d;
            d = c;
            c = b;
            b = b + ((a + f + w[g] + 0x5A827999) << (i % 32));
            a = temp;
        }

        // Add this chunk's hash to result so far
        h0 += a;
        h1 += b;
        h2 += c;
        h3 += d;
    }

    // Free dynamically allocated memory
    free(msg);

    // Output hash
    memcpy(digest, &h0, 4);
    memcpy(digest + 4, &h1, 4);
    memcpy(digest + 8, &h2, 4);
    memcpy(digest + 12, &h3, 4);
}

void generateMAC(const char *message, const char *key, uint8_t *mac) {
    uint8_t combined_message[MAX_MESSAGE_LENGTH + MD5_DIGEST_LENGTH];
    size_t message_length = strlen(message);
    size_t key_length = strlen(key);
    // Concatenate key and message
    memcpy(combined_message, key, key_length);
    memcpy(combined_message + key_length, message, message_length);

    // Calculate MD5 hash
    md5(combined_message, key_length + message_length, mac);
}

int main() {
    char message[MAX_MESSAGE_LENGTH];
    char key[] = "secret_key";
    uint8_t mac[MD5_DIGEST_LENGTH];

    printf("Enter the message: ");
    fgets(message, MAX_MESSAGE_LENGTH, stdin);

    // Remove trailing newline character if present
    if (message[strlen(message) - 1] == '\n')
        message[strlen(message) - 1] = '\0';

    generateMAC(message, key, mac);

    printf("MAC generated: ");
    for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
        printf("%02x", mac[i]);
    }
    printf("\n");

    return 0;
```

}



```c
int main() {
    char message[MAX_MESSAGE_LENGTH];
    char key[] = "secret_key";
    uint8_t mac[MD5_DIGEST_LENGTH];

    printf("Enter the message: ");
    fgets(message, MAX_MESSAGE_LENGTH, stdin);

    // Remove trailing newline character if present
    if (message[strlen(message) - 1] == '\n')
        message[strlen(message) - 1] = '\0';

    generateMAC(message, key, mac);

    printf("MAC generated: ");
    for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
        printf("%02x", mac[i]);
    }
    printf("\n");

    return 0;
}
```

```
/tmp/HKA87c8LmU.o
Enter the message: HEMANTH
MAC generated: e6b648b36e3fd17be370be245be8355c


=== Code Execution Successful ===
```

Stream Ciphers:

Advantages:
Typically faster than block ciphers, especially in hardware implementations.
Well-suited for real-time communication and streaming applications.
Disadvantages:
Vulnerable to certain types of attacks if the same key and nonce are reused.
Can be less efficient in software implementations compared to hardware.

Block Ciphers:

Advantages:
Provide better security guarantees due to the larger key space and resistance to known attacks (when used properly).
Can be used in various modes of operation, providing flexibility for different cryptographic applications.
Disadvantages:
Can be slower than stream ciphers, especially in software implementations.
May require padding for messages that are not an exact multiple of the block size, which can introduce complexity.
The choice between stream ciphers and block ciphers depends on the specific requirements of the cryptographic application, such as speed, security guarantees, and available resources.