```c
// Name- Adarsh Maddheshiya
// Id-   202151004

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define P 1021
#define A 449
#define B 233
#define SHA256_ROTATE(x, n) (((x) >> (n)) | ((x) << (32 - (n))))

typedef struct
{
    uint16_t x;
    uint16_t y;
} Point;


int M[4][4] = {{1, 4, 4, 5}, {5, 1, 4, 4}, {4, 5, 1, 4}, {4, 4, 5, 1}};
int M_inv[4][4] = {{165, 7, 26, 115}, {115, 165, 7, 26}, {26, 115, 165, 7},
 {7, 26, 115, 165}};

void SHA256(unsigned char *message, size_t length, unsigned char *digest) {
    // Initialize hash value
    uint32_t H[8] = {
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
    };

    // Initializes array of round constants
    uint32_t 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 the message
size_t initial_len = length;
// Pre-processing, appending a single '1' bit
message[length++] = 0x80;
// Pre-processing, appending enough '0' bits
while (length % 64 != 56) {
    message[length++] = 0x00;
}
// Pre-processing, appending length of message (before pre-processing), in
 bits, as 64-bit big-endian integer
uint64_t bit_length = initial_len * 8;
for (int i = 0; i < 8; ++i) {
    message[length++] = (bit_length >> ((7 - i) * 8)) & 0xFF;
}

// Processing the message in successive 512-bit chunks
for (size_t chunk = 0; chunk < length / 64; ++chunk) {
    // Break chunk into sixteen 32-bits big-endian words
    uint32_t W[64];
    for (int i = 0; i < 16; ++i) {
        W[i] = (message[chunk * 64 + i * 4] << 24) | (message[chunk * 64 +
         i * 4 + 1] << 16) |
               (message[chunk * 64 + i * 4 + 2] << 8) | (message[chunk *
                64 + i * 4 + 3]);
    }

    // Extend the sixteen 32-bit words into sixty-four 32-bits words
    for (int i = 16; i < 64; ++i) {
        uint32_t s0 = SHA256_ROTATE(W[i - 15], 7) ^ SHA256_ROTATE(W[i -
         15], 18) ^ (W[i - 15] >> 3);
        uint32_t s1 = SHA256_ROTATE(W[i - 2], 17) ^ SHA256_ROTATE(W[i -
         2], 19) ^ (W[i - 2] >> 10);
        W[i] = W[i - 16] + s0 + W[i - 7] + s1;
    }

    // Initialize working variables to current hash values
    uint32_t a = H[0], b = H[1], c = H[2], d = H[3], e = H[4], f = H[5], g
     = H[6], h = H[7];

    // Compression function main loop.
    for (int i = 0; i < 64; ++i) {
        uint32_t S1 = SHA256_ROTATE(e, 6) ^ SHA256_ROTATE(e, 11) ^
         SHA256_ROTATE(e, 25);
        uint32_t ch = (e & f) ^ ((~e) & g);
        uint32_t temp1 = h + S1 + ch + K[i] + W[i];
        uint32_t S0 = SHA256_ROTATE(a, 2) ^ SHA256_ROTATE(a, 13) ^
         SHA256_ROTATE(a, 22);
        uint32_t maj = (a & b) ^ (a & c) ^ (b & c);
        uint32_t temp2 = S0 + maj;
```

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

        // Add the compressed chunk to the current hash values
        H[0] += a;
        H[1] += b;
        H[2] += c;
        H[3] += d;
        H[4] += e;
        H[5] += f;
        H[6] += g;
        H[7] += h;
    }

    // Produceing the final hash value (big-endian)
    for (int i = 0; i < 8; ++i) {
        digest[i * 4] = (H[i] >> 24) & 0xFF;
        digest[i * 4 + 1] = (H[i] >> 16) & 0xFF;
        digest[i * 4 + 2] = (H[i] >> 8) & 0xFF;
        digest[i * 4 + 3] = H[i] & 0xFF;
    }
}

Point EL_add(Point a, Point b)
{
    int L = ((b.y - a.y) * (int)pow(a.x - b.x, P - 2)) % P; // pow() to
     (int)pow()
    int x = (L * L - a.x - b.x + P) % P; //  calculation for modulo operation
    int y = (L * (a.x - x) - a.y + P) % P; //  calculation for modulo operation
    return (Point){x, y};
}

// Function to perform scalar multiplication of a point 'a' on an elliptic
 curve by an integer n
// Use double-and-add algorithm for efficient computation
// Return the result of the scalar multiplication
Point EL_mul(Point a, int n)
{
    Point b = {1, 1};
    for (int i = 0; i < 16; i++)
    {
        for (int j = 0; j < 4; j++)
        {
```

```
            if (n & (1 << (15 - i)))
            {
                b = EL_add(b, a);
            }
            a = EL_add(a, a);
        }
    }
    return b;
}

// Function to generate a random point on an elliptic curve
// Randomly generate x-coordinate within the curve's field and calculate
 corresponding y-coordinate
Point EL_rand()
{
    Point p;
    do
    {
        p.x = rand() % P;
        p.y = (uint16_t)((p.x * p.x * p.x + A * p.x + B) % P); //  calculation
         for y coordinate.
    } while (p.y == 0);
    return p;
}

int Subbyte_prime(int x)
{
    static int sbox[256] = {
        0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
         0x2b, 0xfe, 0xd7, 0xab, 0x76,
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2,
         0xaf, 0x9c, 0xa4, 0x72, 0xc0,
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5,
         0xf1, 0x71, 0xd8, 0x31, 0x15,
        0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80,
         0xe2, 0xeb, 0x27, 0xb2, 0x75,
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6,
         0xb3, 0x29, 0xe3, 0x2f, 0x84,
        0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe,
         0x39, 0x4a, 0x4c, 0x58, 0xcf,
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02,
         0x7f, 0x50, 0x3c, 0x9f, 0xa8,
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda,
         0x21, 0x10, 0xff, 0xf3, 0xd2,
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e,
         0x3d, 0x64, 0x5d, 0x19, 0x73,
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8,
         0x14, 0xde, 0x5e, 0x0b, 0xdb,
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac,
         0x62, 0x91, 0x95, 0xe4, 0x79,
```

```
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4,
         0xea, 0x65, 0x7a, 0xae, 0x08,
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74,
         0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57,
         0xb9, 0x86, 0xc1, 0x1d, 0x9e,
        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87,
         0xe9, 0xce, 0x55, 0x28, 0xdf,
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d,
         0x0f, 0xb0, 0x54, 0xbb, 0x16};
    return sbox[(221 * x) + 125];
}

// Function to performing matrix multiplication between the fixed M matrix and
 the state matrix
void Mixcolumn_prime(int state[4])
{
    int temp[4];
    int M[4][4] = {{1, 4, 4, 5}, {5, 1, 4, 4}, {4, 5, 1, 4}, {4, 4, 5, 1}}; //
     Moved M matrix declaration here
    for (int i = 0; i < 4; i++)
    {
        temp[i] = 0;
        for (int j = 0; j < 4; j++)
        {
            temp[i] ^= M[i][j] * state[j];
        }
    }
    for (int i = 0; i < 4; i++)
    {
        state[i] = temp[i];
    }
}

// Function to generates round keys using the key expansion algorithm
void Key_schedule(int key[16], int k[10][4])
{
    for (int i = 0; i < 16; i++)
    {
        k[0][i] = key[i];
    }
    for (int r = 1; r < 10; r++)
    {
        for (int i = 0; i < 4; i++)
        {
            k[r][4 * i] = k[r - 1][4 * i] ^ Subbyte_prime(k[r - 1][4 * i + 1])
             ^ k[r - 1][4 * i + 2] ^ k[r - 1][4 * i + 3];
            k[r][4 * i + 1] = Subbyte_prime(k[r - 1][4 * i]) ^ k[r - 1][4 * i
             + 1] ^ k[r - 1][4 * i + 2] ^ k[r - 1][4 * i + 3];
```

```
            k[r][4 * i + 2] = Subbyte_prime(k[r - 1][4 * i]) ^
             Subbyte_prime(k[r - 1][4 * i + 1]) ^ k[r - 1][4 * i + 2] ^ k[r -
             1][4 * i + 3];
            k[r][4 * i + 3] = Subbyte_prime(k[r - 1][4 * i]) ^
             Subbyte_prime(k[r - 1][4 * i + 1]) ^ Subbyte_prime(k[r - 1][4 * i
             + 2]) ^ k[r - 1][4 * i + 3];
        }
    }
}

// Function to performs AES encryption using the given state and round keys
void AES_prime_encrypt(int state[4], int k[10][4])
{
    for (int r = 0; r < 10; r++)
    {
        for (int i = 0; i < 4; i++)
        {
            state[i] = Subbyte_prime(state[i]);
            for (int j = 0; j < 4; j++)
            {
                state[i] ^= k[r][4 * j + i];
            }
        }
        Mixcolumn_prime(state);
    }
}

// Function to perform triple AES encryption on input using three keys and an
 initialization vector IV
void Triple_AES_prime_encrypt(unsigned char *input, unsigned char *output, int
 key[3][16], int iv[4])
{
    int state[4];
    int SK[4] = {0}; // SK to be an array of integers.
    for (int i = 0; i < 16; i += 4)
    {
        state[0] = (input[i] << 24) | (input[i + 1] << 16) | (input[i + 2] <<
         8) | input[i + 3];
        state[1] = (input[i + 4] << 24) | (input[i + 5] << 16) | (input[i + 6]
         << 8) | input[i + 7];
        state[2] = (input[i + 8] << 24) | (input[i + 9] << 16) | (input[i +
         10] << 8) | input[i + 11];
        state[3] = (input[i + 12] << 24) | (input[i + 13] << 16) | (input[i +
         14] << 8) | input[i + 15];
        AES_prime_encrypt(state, key[0]);
        for (int j = 0; j < 4; j++)
        {
            output[i + j] = (state[j] >> 24) & 0xff;
            output[i + j + 16] = (state[j] >> 16) & 0xff;
            output[i + j + 32] = (state[j] >> 8) & 0xff;
            output[i + j + 48] = state[j] & 0xff;
```

```
        }
        AES_prime_encrypt(state, key[1]);
        for (int j = 0; j < 4; j++)
        {
            state[j] ^= iv[j];
        }
        AES_prime_encrypt(state, key[2]);
        for (int j = 0; j < 4; j++)
        {
            iv[j] = state[j];
        }
    }
}

void Triple_AES_prime_decrypt(unsigned char *input, unsigned char *output, int
 key[3][16], int iv[4]) {
    // Triple_AES_prime_decrypt implementation
    // Decrypt using Triple AES
    // Perform decryption with the third key first
    unsigned char temp[16];
    for (int i = 0; i < 16; i++) {
        temp[i] = input[i] ^ key[2][i];
    }

    // then use the second keys
    for (int i = 0; i < 16; i++) {
        output[i] = temp[i] ^ key[1][i];
    }

    // finally use the first key and XOR with IV.
    for (int i = 0; i < 16; i++) {
        output[i] ^= key[0][i] ^ iv[i % 4];
    }
}

void MAC_generate(unsigned char *key, unsigned char *input, unsigned char
 *mac) {
    // MAC_generate implementation
    // Generate MAC using SHA-256
    unsigned char KA[32];
    SHA256(key, 16, KA);
    unsigned char temp[48];
    memcpy(temp, input, 16);
    memcpy(temp + 16, KA, 32);
    SHA256(temp, 48, mac);
}

int main()
{
    // Initialize the random number generator.
    srand(time(NULL));
```

```
// Generate the point alpha
Point alpha = EL_rand();
printf("alpha = (%d, %d)\n", alpha.x, alpha.y);

// Get the private keys nA and nB
int nA, nB;
printf("Enter Alice's private key (nA): ");
scanf("%d", &nA);
printf("Enter Bob's private key (nB): ");
scanf("%d", &nB);

// Compute the shared secret key SK
Point SK = EL_mul(alpha, nA);
printf("SK = (%d, %d)\n", SK.x, SK.y);

// Compute the keys KA and KB
unsigned char KA[32];
unsigned char KB[32];
SHA256((unsigned char *)&SK.x, sizeof(SK.x), KA);
SHA256((unsigned char *)&SK.x, sizeof(SK.x), KB);

// Print KA and KB
printf("KA = ");
for (int i = 0; i < 32; i++)
{
    printf("%02x ", KA[i]);
}
printf("\n");
printf("KB = ");
for (int i = 0; i < 32; i++)
{
    printf("%02x ", KB[i]);
}
printf("\n");
// Get Alice's message MA
unsigned char MA[16]; // Changed to array
printf("Enter Alice's message (MA) in hexadecimal (space separated): ");
for (int i = 0; i < 16; i++)
{
    scanf("%2x", &MA[i]);
}
// Encrypt the message MA using Triple-AES'-128
unsigned char CA[16]; // Changed to array
unsigned char KA_prime[32]; // Changed to array
memcpy(KA_prime, KA, sizeof(KA)); // Copy KA to KA_prime
Triple_AES_prime_encrypt(MA, CA, KA_prime, SK.x);
// Generate the MAC for MA
unsigned char MACA[32]; // Changed to array
MAC_generate(KA, MA, MACA);
// print the ciphertext CA and the MACA
```

```
printf("CA = ");
for (int i = 0; i < 16; i++)
{
    printf("%02x ", CA[i]);
}
printf("\n");
printf("MACA = ");
for (int i = 0; i < 32; i++)
{
    printf("%02x ", MACA[i]);
}
printf("\n");
// Bob receive the ciphertext CA and the MACA
unsigned char CB[16]; // Changed to array
unsigned char MACB[32]; // Changed to array
printf("Enter Bob's received ciphertext (CA) in hexadecimal (space
 separated): ");
for (int i = 0; i < 16; i++)
{
    scanf("%2x", &CB[i]);
}
printf("Enter Bob's received MACA in hexadecimal (space separated): ");
for (int i = 0; i < 32; i++)
{
    scanf("%2x", &MACB[i]);
}
// Bob decrypts the ciphertext CA using Triple-AES'-128
unsigned char MB[16]; // Changed to array
Triple_AES_prime_decrypt(CA, MB, KB, SK.x);
// Bob generates the MAC for MB
unsigned char MACC[32]; // Changed to array
MAC_generate(KB, MB, MACC);
// Check if the MACs match
int match = 1;
for (int i = 0; i < 32; i++)
{
    if (MACA[i] != MACC[i] || MACB[i] != MACC[i])
    {
        match = 0;
        break;
    }
}
// Print the decrypted message MB and the MACC
printf("MB = ");
for (int i = 0; i < 16; i++)
{
    printf("%02x ", MB[i]);
}
printf("\n");
printf("MACC = ");
for (int i = 0; i < 32; i++)
```

```
    {
        printf("%02x ", MACC[i]);
    }
    printf("\n");
    // Check if the keys match
    match &= (memcmp(KA, KB, 16) == 0);
    // Check if the messages match
    match &= (memcmp(MA, MB, 16) == 0);
    // Check if the MACs match
    match &= (memcmp(MACA, MACB, 32) == 0);
    // Print the result
    if (match)
    {
        printf("The keys, messages, and MACs match!\n");
    }
    else
    {
        printf("The keys, messages, or MACs do not match!\n");
    }
    return 0;
}
```