Cryptography and Network Security Lab

Assignment 6
Implementation and Understanding of Data Encryption Standard (DES) Cipher

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Data Encryption Standard (DES)

Aim: To Study, Implement and Demonstrate the Data Encryption Standard (DES)

- Part A- Implementation of DES using Virtual Lab
- Part B- Implementation of DES using C/C++/Java/Python or any other programming language

Theory:

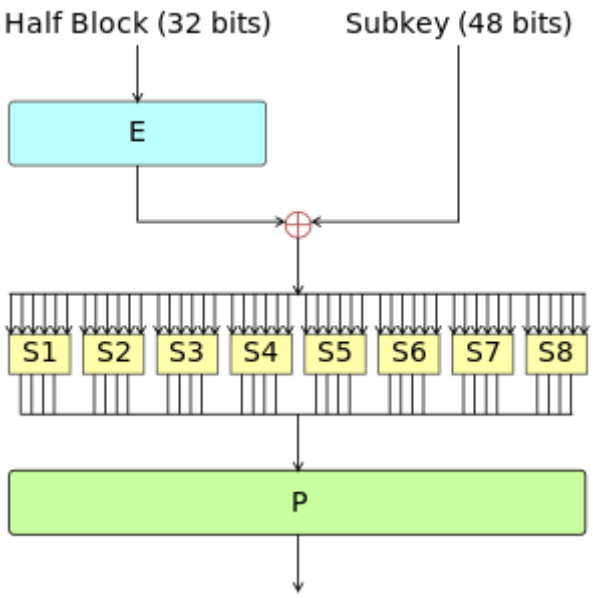
The Data Encryption Standard is a symmetric-key algorithm for the encryption of digital data. Although its short key length of 56 bits makes it too insecure for modern applications, it has been highly influential in the advancement of cryptography.

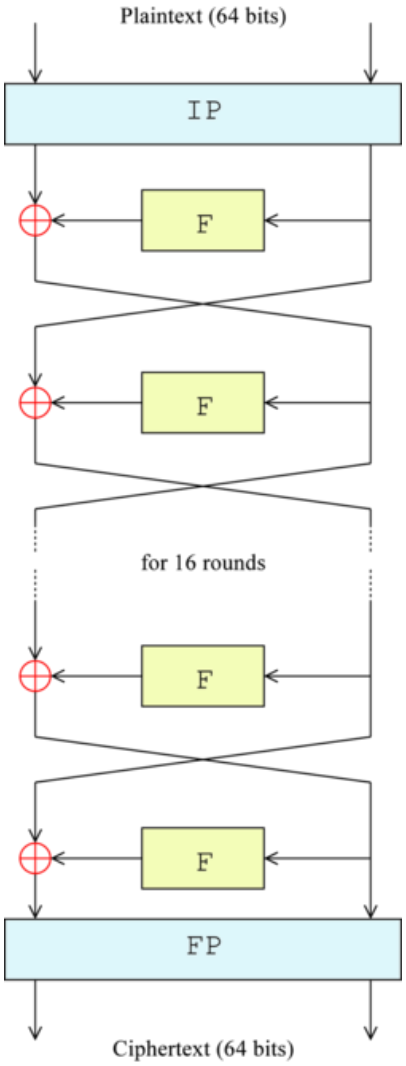
The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST). DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56bits, since 8 of the 64 bits of the key are not used by the encryption algorithm.

DES is the archetypal block cipher—an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity, and are thereafter discarded. Hence the effective key length is 56 bits. The key is nominally stored or transmitted as 8 bytes, each with odd parity.

One bit in each 8-bit byte of the KEY may be utilized for error detection in key generation, distribution, and storage. Bits 8, 16, ... , 64 are for use in ensuring that each byte is of odd parity. Like other block ciphers, DES by itself is not a secure means of encryption, but must instead be used in a mode of operation. FIPS-81 specifies several modes for use with DES. Further comments on the usage of DES are contained in FIPS-74.

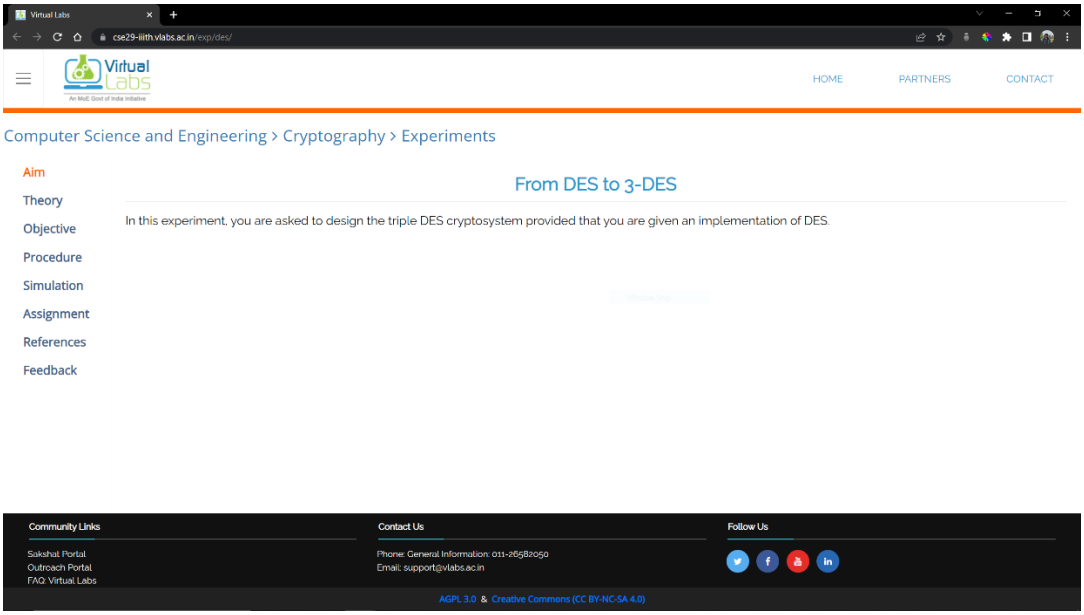
Decryption uses the same structure as encryption, but with the keys used in reverse order. (This has the advantage that the same hardware or software can be used in both directions.)



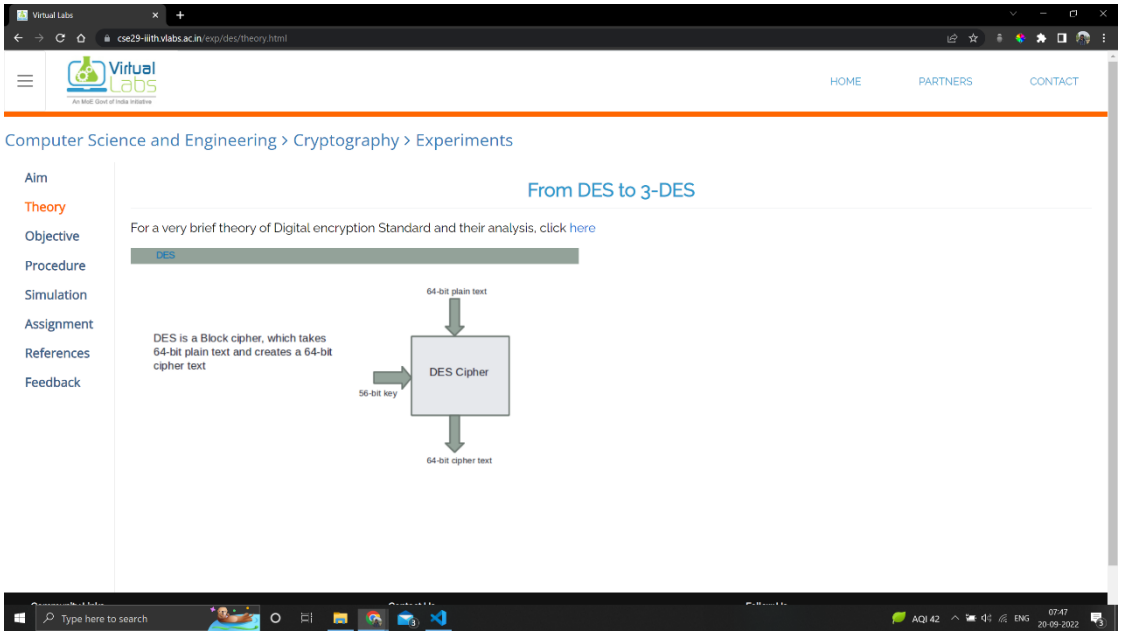


V-Lab Implementation:

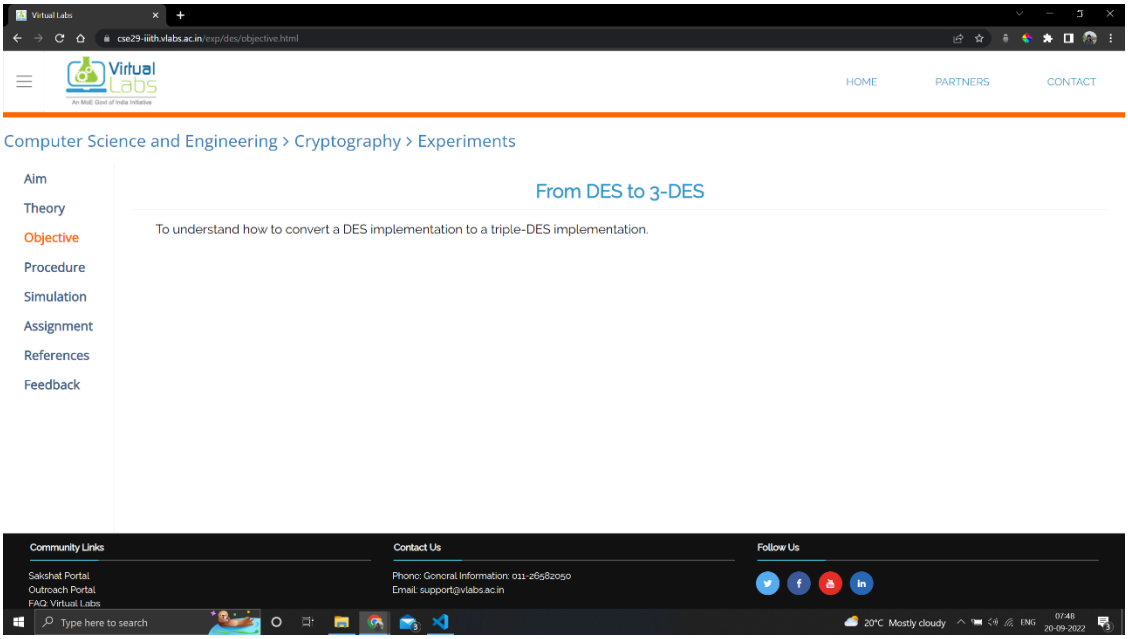
Let us work on the simulator. This simulator is performing the 3DES algorithm.



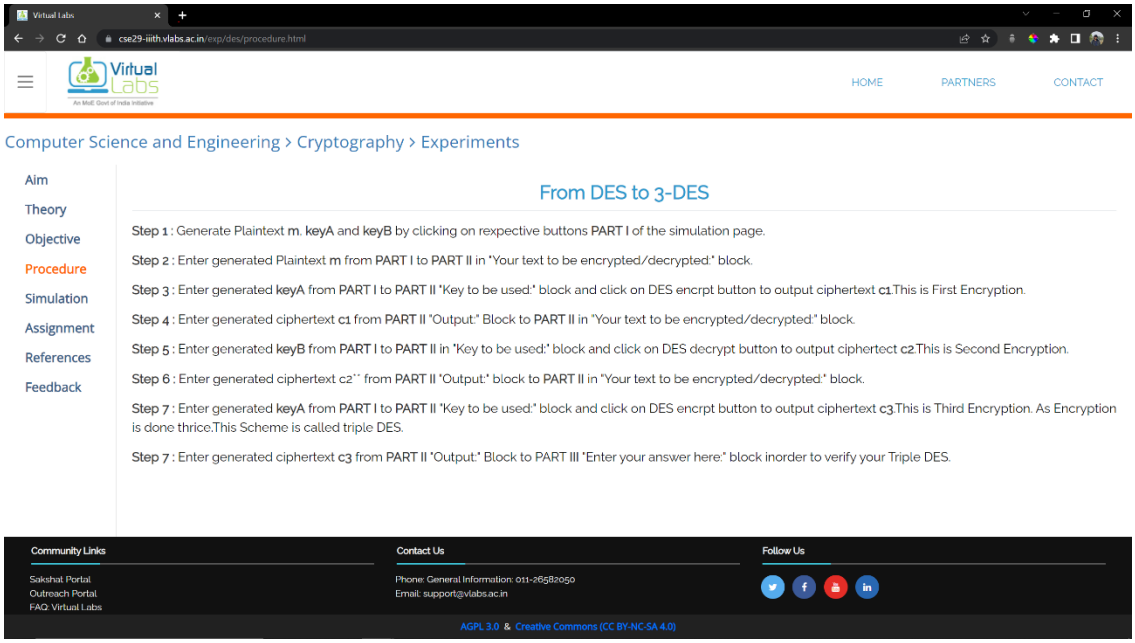
Looking at the theory:



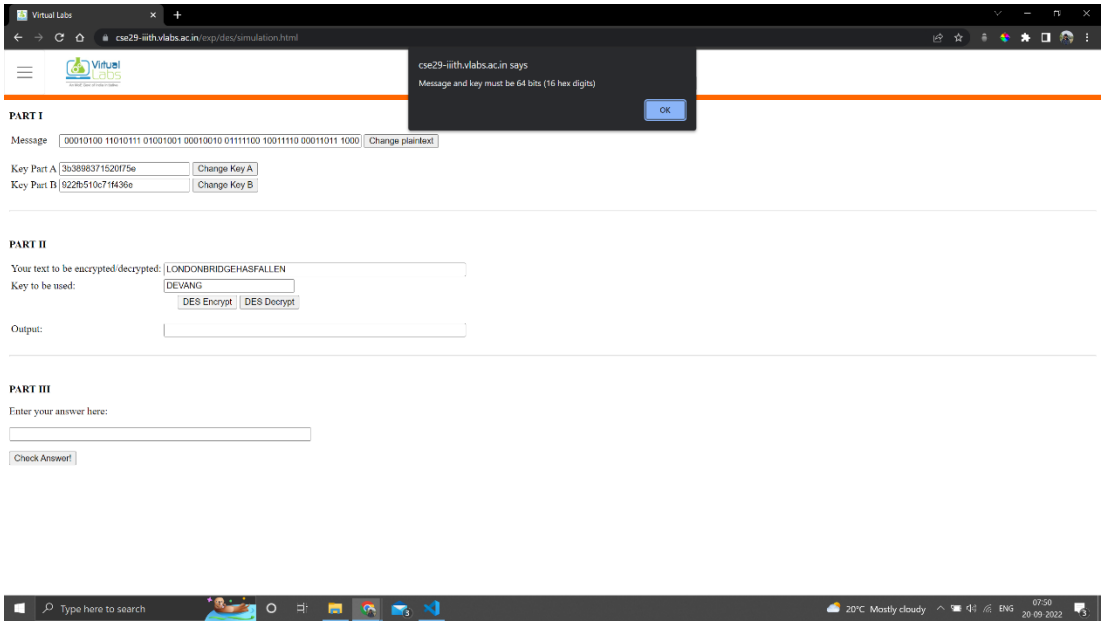
Objective:



Procedure



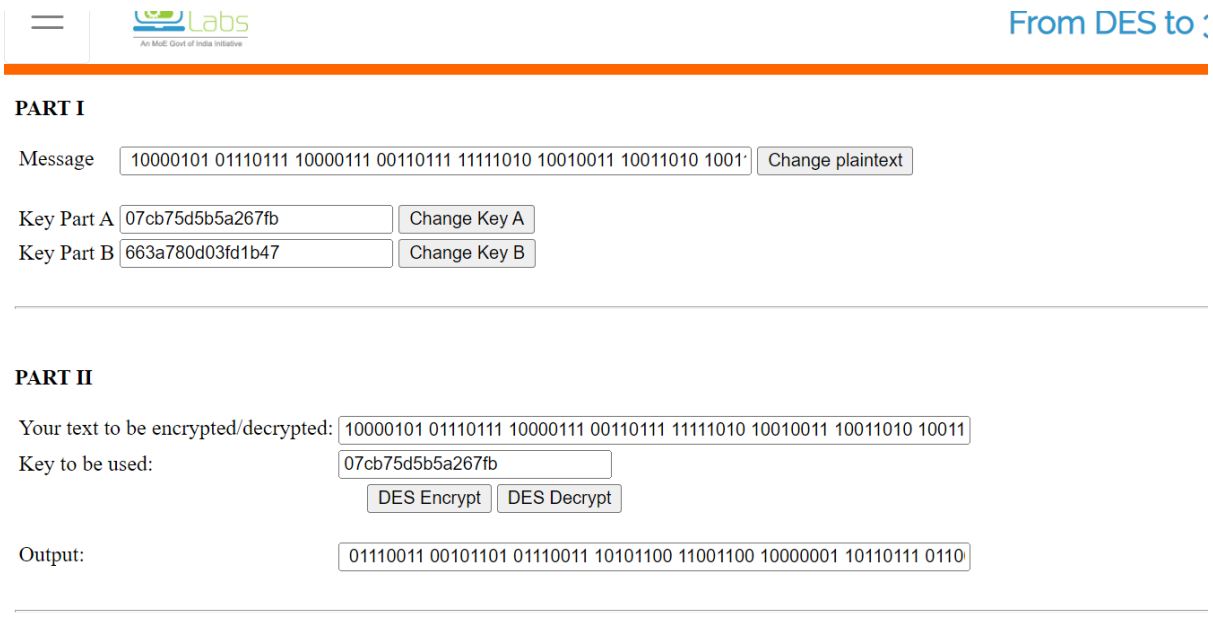
We try entering a value



Simple DES

Plaintext: 10000101 01110111 10000111 00110111 11111010 10010011 10011010 1001111

Key: 07cb75d5b5a267fb



EncText: 01110011 00101101 01110011 10101100 11001100 10000001 10110111 01100010

Decryption

Plaintext: 10000101 01110111 10000111 00110111 11111010 10010011 10011010 10011110

PART I

Message: 10000101 01110111 10000111 00110111 11111010 10010011 10011010 1001 [Change plaintext]

Key Part A: 07cb75d5b5a267fb [Change Key A]

Key Part B: 663a780d03fd1b47 [Change Key B]

PART II

Your text to be encrypted/decrypted: 01110011 00101101 01110011 10101100 11001100 10000001 10110111 0110

Key to be used: 07cb75d5b5a267fb

[DES Encrypt] [DES Decrypt]

Output: 10000101 01110111 10000111 00110111 11111010 10010011 10011010 1001

Let's do once from Part 1 values

DES -> 3DES

Plaintext -> 00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000001

Key A -> 3b3898371520f75e

Key B -> 922fb510c71f436e

PART II

Your text to be encrypted/decrypted: 593428AE137D8346

Key to be used: 975321BA72BA9361

[DES Encrypt] [DES Decrypt]

Output: 95438200 31173864 00000000 10010011 10000010 00001101 00100011 011

Text - 593428AE137D8346
Key - 975321BA72BA9361

Encryption: 95438200 31173864 00000000 10010011 10000010 00001101 00100011 01100100

Then we take this encrypted text and encrypt with another key.

PART II

Your text to be encrypted/decrypted: 95438200 31173864 00000000 10010011 10000010 00001101 00100011 011
Key to be used: 975321BA72BA9361
DES Encrypt | DES Decrypt

Output: 59342800 13708346 00000000 00000000 00000000 00000000 00000000 00(

Plaintext -> 00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000001

Key A -> 3b3898371520f75e

Key B -> 922fb510c71f436e

An MoE Govt of India Initiative

PART I

Message 00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000 Change plaintext

Key Part A 3b3898371520f75e Change Key A
Key Part B 922fb510c71f436e Change Key B

PART II

Your text to be encrypted/decrypted: 00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000(
Key to be used: 3b3898371520f75e
DES Encrypt | DES Decrypt

Output: 00111110 11010100 11010111 01101101 10000110 11100111 00010001 01111

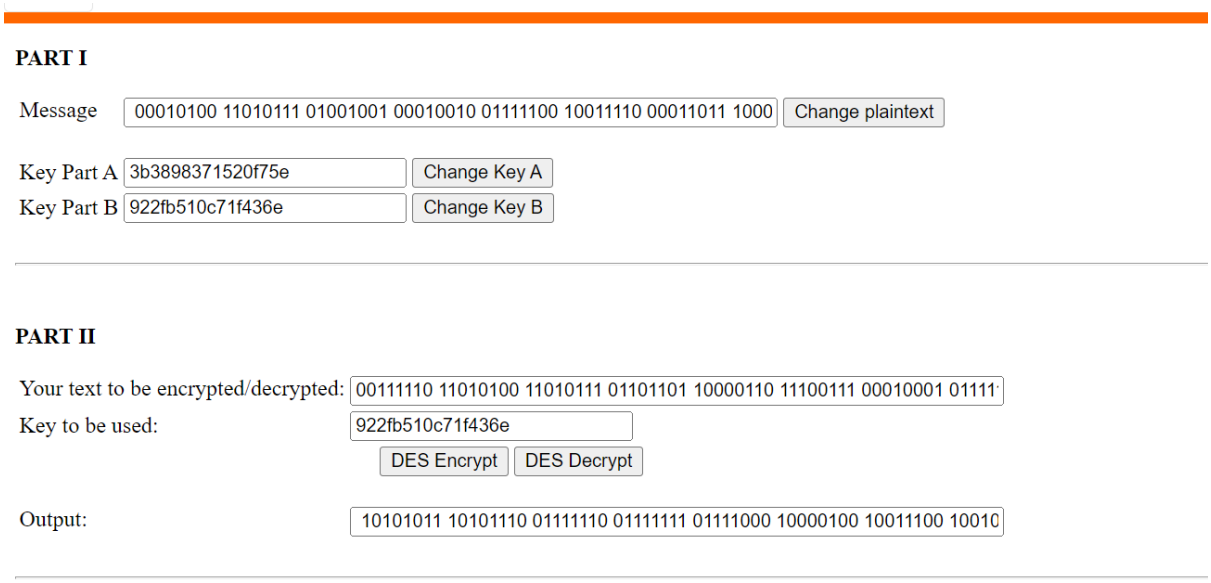
Basically, we need to do this:

Plaintext + KeyA -> C1 Enc

C1 + KeyB -> C2 Dec

C2 + KeyA -> C3 Enc

C3 is the 3 DES Cipher



PT + KA -> C1 Enc

00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000001 + 3b3898371520f75e => 00111110 11010100 11010111 01101101 10000110 11100111 00010001 01111101

PART I

Message 00010100 11010111 01001001 00010010 01111100 10011110 00011011 1000 Change plaintext

Key Part A 3b3898371520f75e Change Key A

Key Part B 922fb510c71f436e Change Key B

PART II

Your text to be encrypted/decrypted: 10101011 10101110 01111110 01111111 01111000 10000100 10011100 10010

Key to be used: 3b3898371520f75e

DES Encrypt DES Decrypt

Output: 00011101 11100100 10001000 01101111 11010001 00011011 00110000 1100

C1 + KB -> C2 Dec

00111110 11010100 11010111 01101101 10000110 11100111 00010001 01111101 + 922fb510c71f436e => 10101011 10101110 01111110 01111111 01111000 10000100 10011100 10010110

PART III

Enter your answer here:

00011101 11100100 10001000 01101111 11010001 00011011 00110000 1100

Check Answer!

CORRECT!

C2 + KA -> C3 Enc

10101011 10101110 01111110 01111111 01111000 10000100 10011100 10010110 + 3b3898371520f75e => 00011101 11100100 10001000 01101111 11010001 00011011 00110000 11000000

Code:

```
# The plaintext and ciphertext would be hexadecimal
def hex2bin(s):
    mp = {'0': "0000",
          '1': "0001",
          '2': "0010",
          '3': "0011",
          '4': "0100",
          '5': "0101",
          '6': "0110",
          '7': "0111",
          '8': "1000",
          '9': "1001",
          'A': "1010",
          'B': "1011",
          'C': "1100",
          'D': "1101",
          'E': "1110",
          'F': "1111"}
    bin = ""
    for i in range(len(s)):
        bin = bin + mp[s[i]]
    return bin

def bin2hex(s):
    mp = {"0000": '0',
          "0001": '1',
          "0010": '2',
          "0011": '3',
          "0100": '4',
          "0101": '5',
          "0110": '6',
          "0111": '7',
          "1000": '8',
          "1001": '9',
          "1010": 'A',
          "1011": 'B',
          "1100": 'C',
          "1101": 'D',
          "1110": 'E',
          "1111": 'F'}
    hex = ""
    for i in range(0, len(s), 4):
        ch = ""
        ch = ch + s[i]
```

```
        ch = ch + s[i + 1]
        ch = ch + s[i + 2]
        ch = ch + s[i + 3]
        hex = hex + mp[ch]
    return hex


# Binary to decimal conversion
def bin2dec(binary):
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal


# Decimal to binary conversion
def dec2bin(num):
    res = bin(num).replace("0b", "")
    if(len(res) % 4 != 0):
        div = len(res) / 4
        div = int(div)
        counter = (4 * (div + 1)) - len(res)
        for i in range(0, counter):
            res = '0' + res
    return res


# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation


# shifting the bits towards left by nth shifts
def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s = s + k[j]
        s = s + k[0]
        k = s
        s = ""
    return k
```

```python
# calculating xow of two strings of binary number a and b
def xor(a, b):
    ans = ""
    for i in range(len(a)):
        if a[i] == b[i]:
            ans = ans + "0"
        else:
            ans = ans + "1"
    return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
         6, 7, 8, 9, 8, 9, 10, 11,
         12, 13, 12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21, 20, 21,
         22, 23, 24, 25, 24, 25, 26, 27,
         28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16,  7, 20, 21,
       29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2,  8, 24, 14,
       32, 27,  3,  9,
       19, 13, 30,  6,
       22, 11,  4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
         [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
         [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
         [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

        [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
         [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
         [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
         [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],
```

```
        [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
         [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
         [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
         [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

        [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
         [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
         [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
         [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

        [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
         [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
         [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
         [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

        [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
         [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
         [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
         [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

        [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
         [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
         [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
         [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

        [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
         [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
         [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
         [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
              39, 7, 47, 15, 55, 23, 63, 31,
              38, 6, 46, 14, 54, 22, 62, 30,
              37, 5, 45, 13, 53, 21, 61, 29,
              36, 4, 44, 12, 52, 20, 60, 28,
              35, 3, 43, 11, 51, 19, 59, 27,
              34, 2, 42, 10, 50, 18, 58, 26,
              33, 1, 41, 9, 49, 17, 57, 25]

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
```

```
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)
    # Initial Permutation
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))
    # Splitting
    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        #  Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)
        # XOR RoundKey[i] and right_expanded
        xor_x = xor(right_expanded, rkb[i])
        # S-boxex: substituting the value from s-box table by calculating row
and column
        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(
                int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] +
xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str = sbox_str + dec2bin(val)
        # Straight D-box: After substituting rearranging the bits
        sbox_str = permute(sbox_str, per, 32)
        # XOR left and sbox_str
        result = xor(left, sbox_str)
        left = result
        # Swapper
```

```
        if(i != 15):
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
              " ", bin2hex(right), " ", rk[i])
    # Combination
    combine = left + right
    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text

# 64bit PT and 64bit Key
print("DES Algorithm")

print("What do you wish to do?")
print("1. Encrypt")
print("2. Decrypt\n")

n = int(input())
if n == 1:
    print("Enter Plaintext: ", end='')
    plaintext = input()
    print("Enter Key: ", end='')
    key = input()
    key = hex2bin(key)
    # Splitting
    left = key[0:28]    # rkb for RoundKeys in binary
    right = key[28:56]  # rk for RoundKeys in hexadecimal
    rkb = []
    rk = []
    for i in range(0, 16):
        # Shifting the bits by nth shifts by checking from shift table
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])

        # Combination of left and right string
        combine_str = left + right

        # Compression of key from 56 to 48 bits
        round_key = permute(combine_str, key_comp, 48)

        rkb.append(round_key)
        rk.append(bin2hex(round_key))
    cipher_text = bin2hex(encrypt(plaintext, rkb, rk))
    print("Cipher Text : ", cipher_text)
else:
    print("Enter Ciphertext: ", end='')
    ciphertext = input()
    print("Enter Key: ", end='')
```

```
    key = input()
    key = hex2bin(key)
    # Splitting
    left = key[0:28]    # rkb for RoundKeys in binary
    right = key[28:56]  # rk for RoundKeys in hexadecimal
    rkb = []
    rk = []
    for i in range(0, 16):
        # Shifting the bits by nth shifts by checking from shift table
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])

        # Combination of left and right string
        combine_str = left + right

        # Compression of key from 56 to 48 bits
        round_key = permute(combine_str, key_comp, 48)

        rkb.append(round_key)
        rk.append(bin2hex(round_key))
    rkb_rev = rkb[::-1]
    rk_rev = rk[::-1]
    text = bin2hex(encrypt(ciphertext, rkb_rev, rk_rev))
    print("Plaintext: ", text)
```

We now solve some examples with the code.

Say we wish to encrypt: ‘123456ABCD132536’
and we take our key as ‘AABB09182736CCDD’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES> py .\script.py

DES Algorithm
What do you wish to do?
1. Encrypt
2. Decrypt

1
Enter Plaintext: 123456ABCD132536
Enter Key: AABB09182736CCDD
```

```
After initial permutation 14A7D67818CA18AD
Round  1   18CA18AD   9DAF94C4   A1DB4D5057F0
Round  2   9DAF94C4   908A3267   AE149ADCF814
Round  3   908A3267   D19BF56B   7E025C2146FC
Round  4   D19BF56B   A12C1D36   0ED81899B883
Round  5   A12C1D36   E03FDA4D   0E297EA64635
Round  6   E03FDA4D   DFD06779   AE6C091B2BC6
Round  7   DFD06779   CB3473C9   4B2F28B4C191
Round  8   CB3473C9   2CDB0C31   C8BC99432647
Round  9   2CDB0C31   428FF863   9940A66093D3
Round  10   428FF863   8C2A99C3   B00BBC17A42F
Round  11   8C2A99C3   FAC20EAE   9432256E1DC0
Round  12   FAC20EAE   23E02501   831E7408E17F
Round  13   23E02501   3F786CF1   CC72E467DC80
Round  14   3F786CF1   8520A7C2   92D768C8057B
Round  15   8520A7C2   543378E7   C853638FDA0C
Round  16   C3B1E73B   543378E7   3FA232090E6A
Cipher Text :  77678609B93FCE56
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES>
```

We now decipher:

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES> py .\script.py

DES Algorithm
What do you wish to do?
1. Encrypt
2. Decrypt

2
Enter Ciphertext: 77678609B93FCE56
Enter Key: AABB09182736CCDD
```

```
After initial permutation C3B1E73B543378E7
Round  1   543378E7   8520A7C2   3FA232090E6A
Round  2   8520A7C2   3F786CF1   C853638FDA0C
Round  3   3F786CF1   23E02501   92D768C8057B
Round  4   23E02501   FAC20EAE   CC72E467DC80
Round  5   FAC20EAE   8C2A99C3   831E7408E17F
Round  6   8C2A99C3   428FF863   9432256E1DC0
Round  7   428FF863   2CDB0C31   B00BBC17A42F
Round  8   2CDB0C31   CB3473C9   9940A66093D3
Round  9   CB3473C9   DFD06779   C8BC99432647
Round  10   DFD06779   E03FDA4D   4B2F28B4C191
Round  11   E03FDA4D   A12C1D36   AE6C091B2BC6
Round  12   A12C1D36   D19BF56B   0E297EA64635
Round  13   D19BF56B   908A3267   0ED81899B883
Round  14   908A3267   9DAF94C4   7E025C2146FC
Round  15   9DAF94C4   18CA18AD   AE149ADCF814
Round  16   14A7D678   18CA18AD   A1DB4D5057F0
Plaintext:  123456ABCD132536
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES>
```

Therefore, we get our plaintext back.

Let’s take another example:

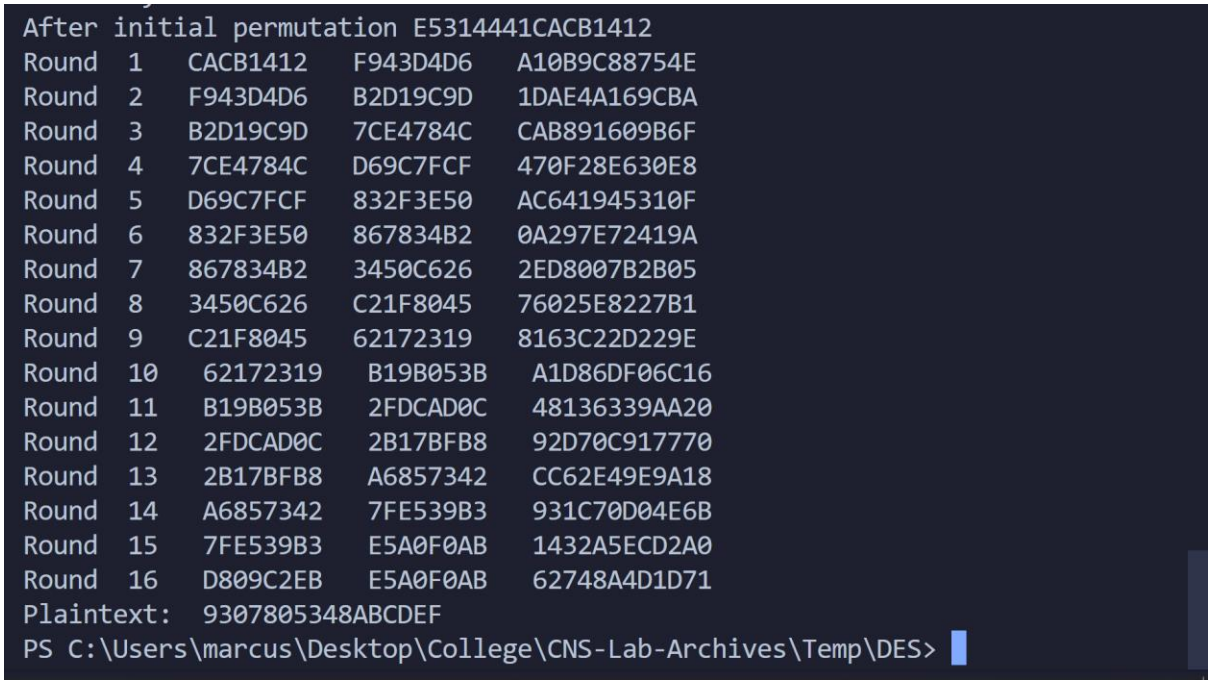
Say we wish to encrypt: ‘9307805348ABCDEF’
and we take our key as ‘1234567890ABCDEF’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES> py .\script.py

DES Algorithm
What do you wish to do?
1. Encrypt
2. Decrypt

1
Enter Plaintext: 9307805348ABCDEF
Enter Key: 1234567890ABCDEF
```

```
After initial permutation D809C2EBE5A0F0AB
Round  1   E5A0F0AB   7FE539B3   62748A4D1D71
Round  2   7FE539B3   A6857342   1432A5ECD2A0
Round  3   A6857342   2B17BFB8   931C70D04E6B
Round  4   2B17BFB8   2FDCAD0C   CC62E49E9A18
Round  5   2FDCAD0C   B19B053B   92D70C917770
Round  6   B19B053B   62172319   48136339AA20
Round  7   62172319   C21F8045   A1D86DF06C16
Round  8   C21F8045   3450C626   8163C22D229E
Round  9   3450C626   867834B2   76025E8227B1
Round  10   867834B2   832F3E50   2ED8007B2B05
Round  11   832F3E50   D69C7FCF   0A297E72419A
Round  12   D69C7FCF   7CE4784C   AC641945310F
Round  13   7CE4784C   B2D19C9D   470F28E630E8
Round  14   B2D19C9D   F943D4D6   CAB891609B6F
Round  15   F943D4D6   CACB1412   1DAE4A169CBA
Round  16   E5314441   CACB1412   A10B9C88754E
Cipher Text :  71A24CA01A50E5E0
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES>
```

We now decipher:

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Temp\DES> py .\script.py

DES Algorithm
What do you wish to do?
1. Encrypt
2. Decrypt

2
Enter Ciphertext: 71A24CA01A50E5E0
Enter Key: 1234567890ABCDEF
```



We get the plaintext back.

Thus, we demonstrated the working of the code with examples.

Conclusion:

Thus, the Data Encryption Standard (DES) algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 7
Implementation and Understanding of Advanced Encryption Standard (AES) Cipher

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Advanced Encryption Standard (AES)

Aim: To Study, Implement and Demonstrate the Advanced Encryption Standard (AES)

- Part A- Implementation of AES using Virtual Lab
- Part B- Implementation of AES using C/C++/Java/Python or any other programming language

Theory:

The Advanced Encryption Standard (AES), also known by its original name Rijndael, as a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES is a variant of the Rijndael block cipher developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits.

AES has been adopted by the U.S. government. It supersedes the Data Encryption Standard (DES),[7] which was published in 1977. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely used today as it is a much stronger than DES and triple DES despite being harder to implement.

Points to remember

- AES is a block cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each.

That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text as output. AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

The number of rounds depends on the key length as follows :

- 128 bit key – 10 rounds
- 192 bit key – 12 rounds
- 256 bit key – 14 rounds

A Key Schedule algorithm is used to calculate all the round keys from the key. So, the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.

Each round comprises of 4 steps :

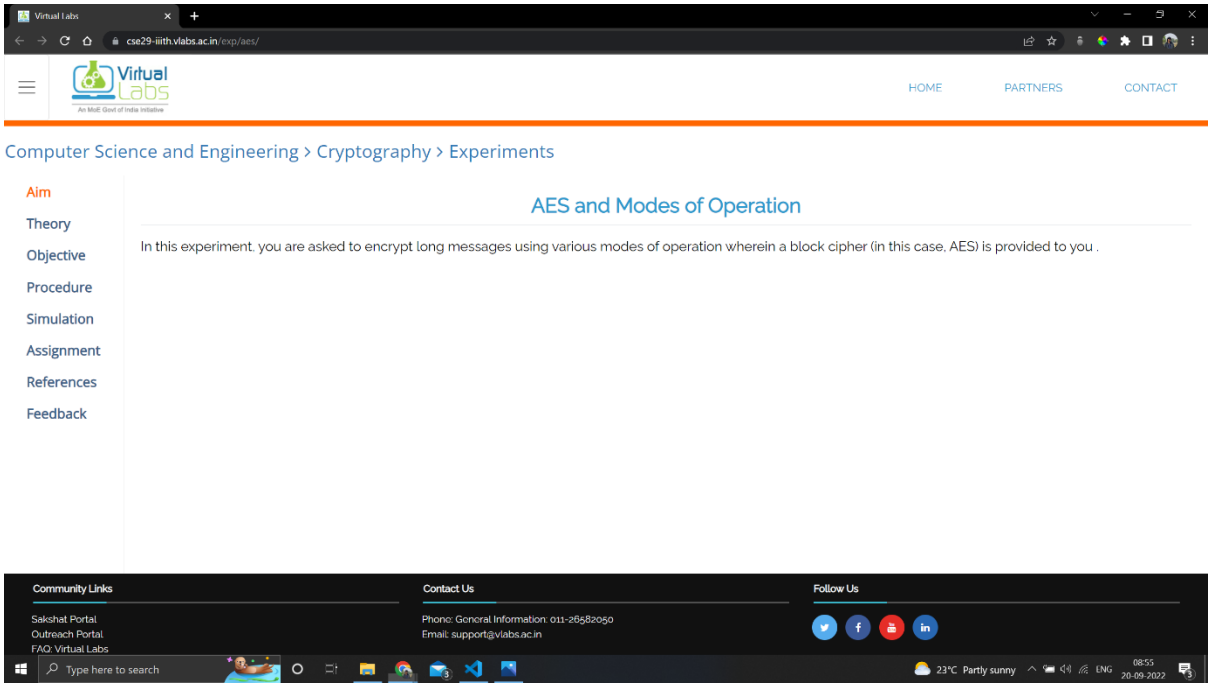
- SubBytes
- ShiftRows
- MixColumns

- Add Round Key

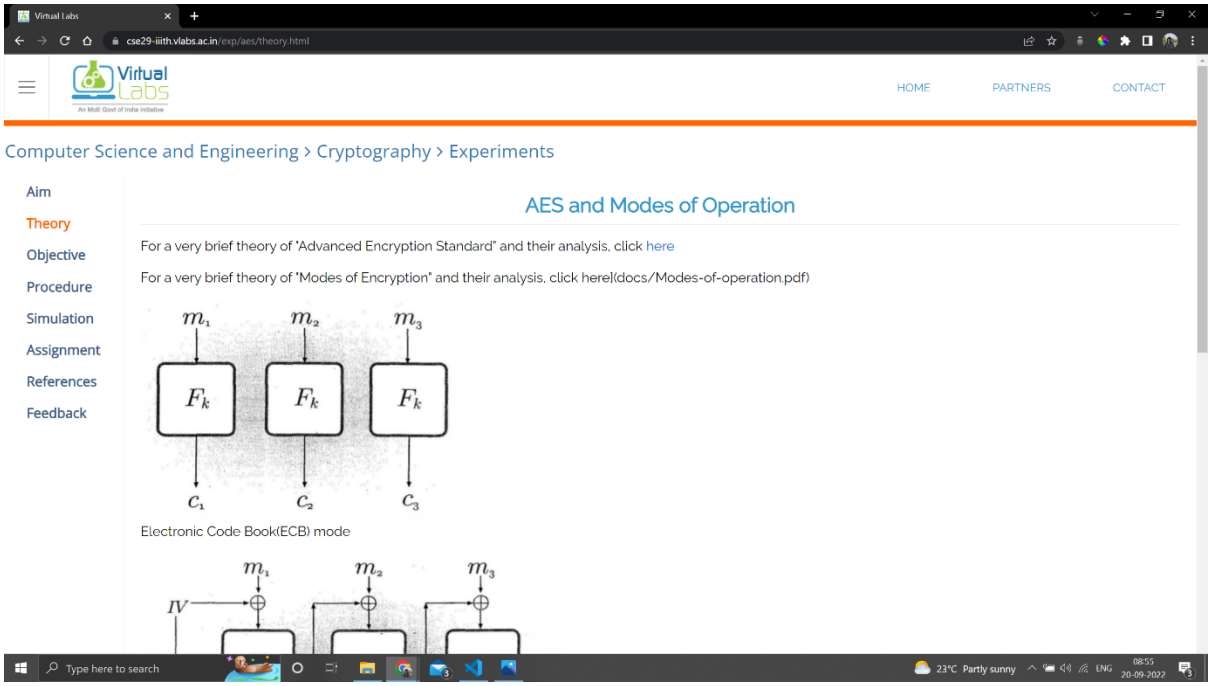
The last round doesn't have the MixColumns round.

AES instruction set is now integrated into the CPU (offers throughput of several GB/s)to improve the speed and security of applications that use AES for encryption and decryption. Even though its been 20 years since its introduction we have failed to break the AES algorithm as it is infeasible even with the current technology. Till date the only vulnerability remains in the implementation of the algorithm.

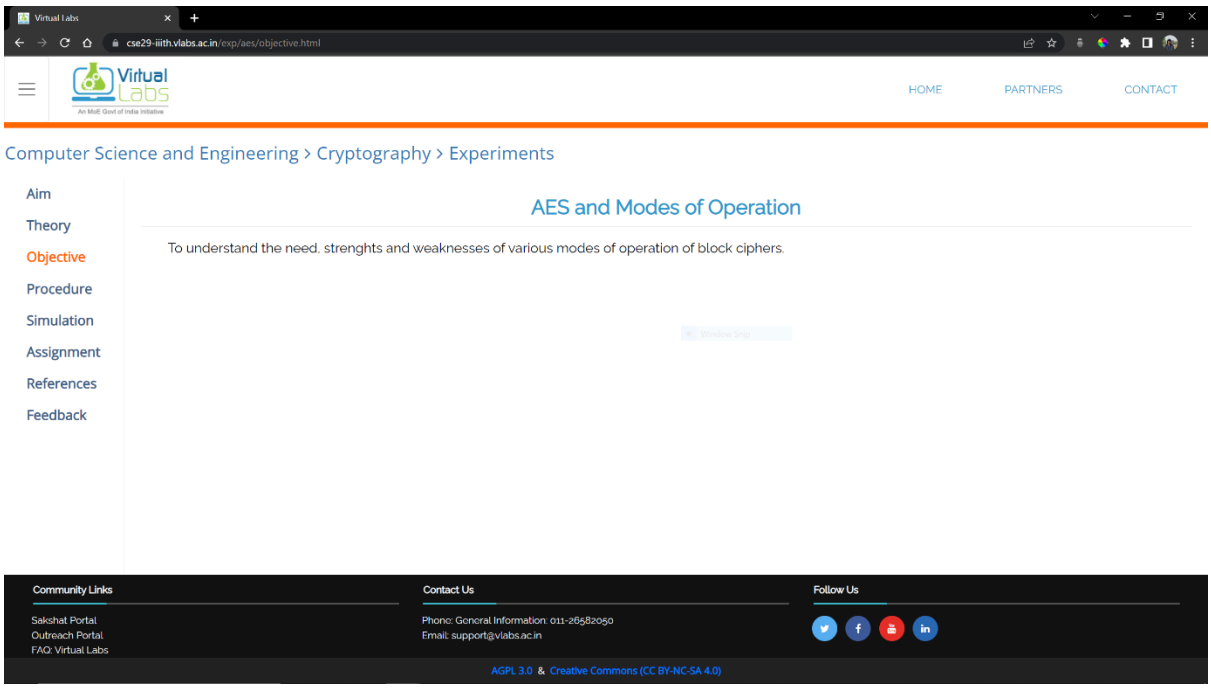
V-Lab Implementation:



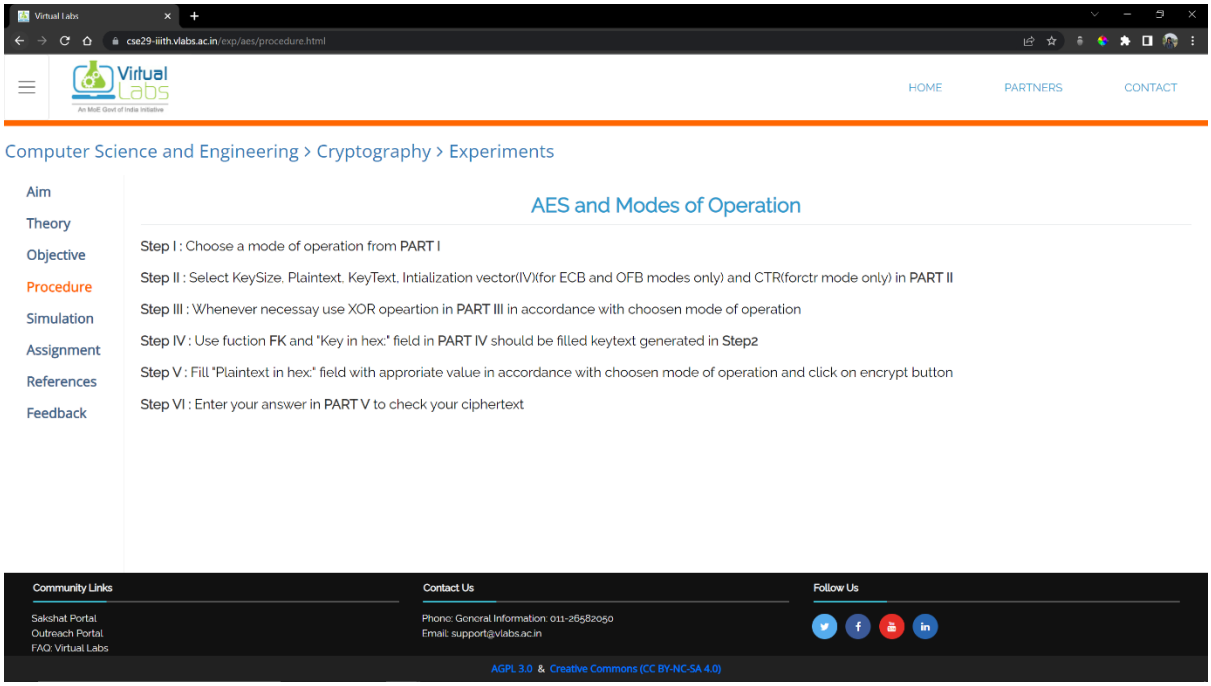
Aim



Theory



Objective

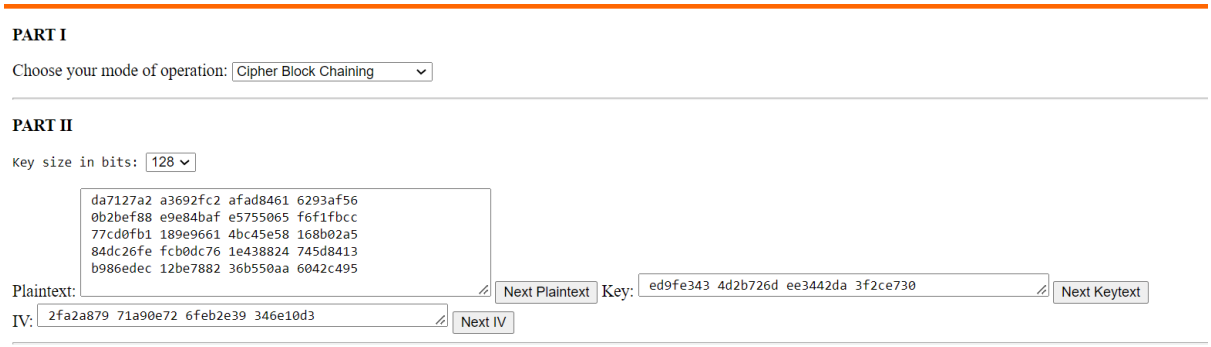


Procedure

We would have to perform in all the 4 modes

Cipher Block Chaining

Generate the values



Plaintext:

da7127a2 a3692fc2 afad8461 6293af56

0b2bef88 e9e84baf e5755065 f6f1fbcc

77cd0fb1 189e9661 4bc45e58 168b02a5

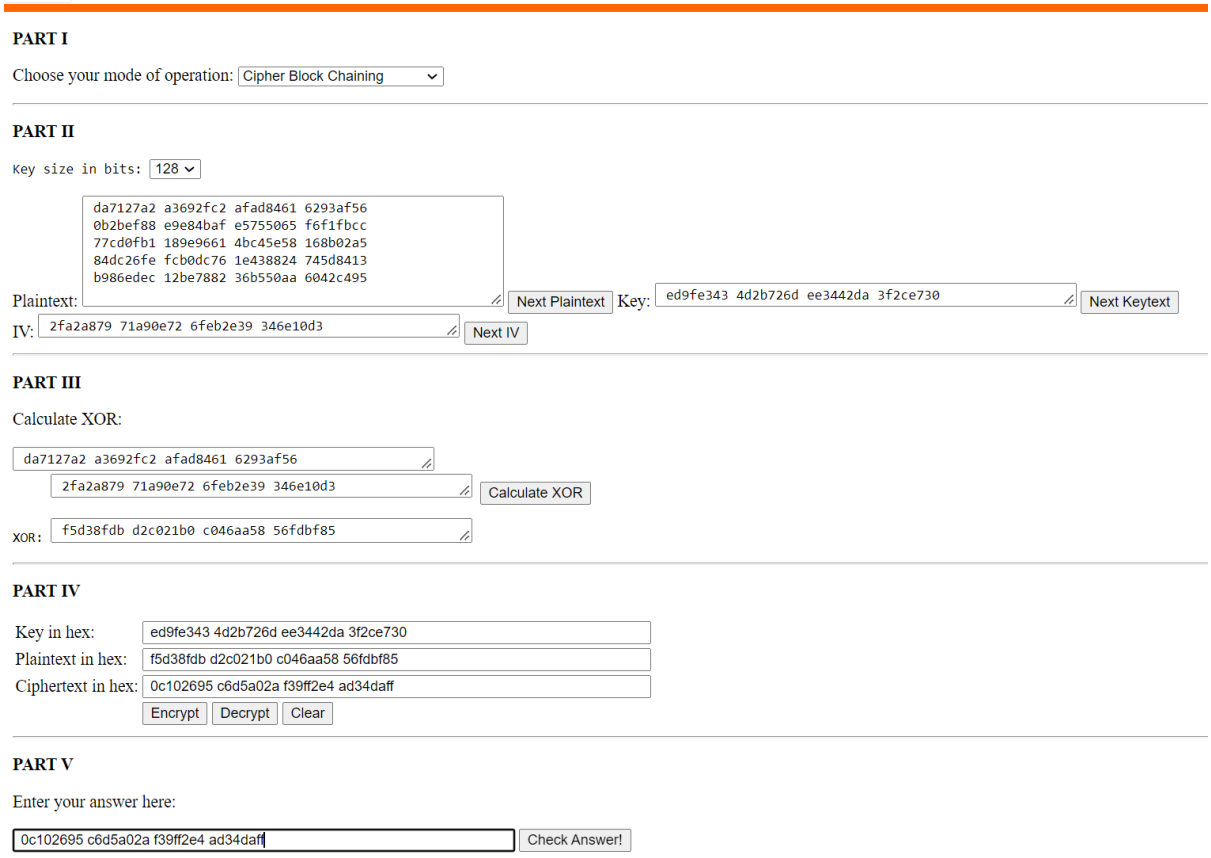
84dc26fe fcb0dc76 1e438824 745d8413

b986edec 12be7882 36b550aa 6042c495

Key: ed9fe343 4d2b726d ee3442da 3f2ce730

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3

First, take message block 1 and XOR with IV then take that Cipher and Encrypt with the key. And add that result to the final output.



0c102695 c6d5a02a f39ff2e4 ad34daff

Now, we take the previously generated Cipher and XOR with Plaintext of 2nd part. Then Encrypt with Key and Append to the final output.

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

0b2bef88 e9e84baf e5755065 f6f1fbcc

0c102695 c6d5a02a f39ff2e4 ad34daff Calculate XOR

XOR: 073bc91d 2f3deb85 16eaa281 5bc52133

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 073bc91d 2f3deb85 16eaa281 5bc52133

Ciphertext in hex: b456d318 268261b7 9abd2009 c8240b4f

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240l Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

77cd0fb1 189e9661 4bc45e58 168b02a5

b456d318 268261b7 9abd2009 c8240b4f Calculate XOR

XOR: c39bdca9 3e1cf7d6 d1797e51 deaf09ea

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: c39bdca9 3e1cf7d6 d1797e51 deaf09ea

Ciphertext in hex: 89801e77 e52dc587 59bd4788 365fa2c1

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240l Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

84dc26fe fcb0dc76 1e438824 745d8413

89801e77 e52dc587 59bd4788 365fa2c1 Calculate XOR

XOR: 0d5c3889 199d19f1 47fecfac 420226d2

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 0d5c3889 199d19f1 47fecfac 420226d2

Ciphertext in hex: f02d4d27 38b7552b 374d15e1 a8d4abf8

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240| Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

b986edec 12be7882 36b550aa 6042c495

f02d4d27 38b7552b 374d15e1 a8d4abf8 Calculate XOR

XOR: 49aba0cb 2a092da9 01f8454b c8966f6d

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 49aba0cb 2a092da9 01f8454b c8966f6d

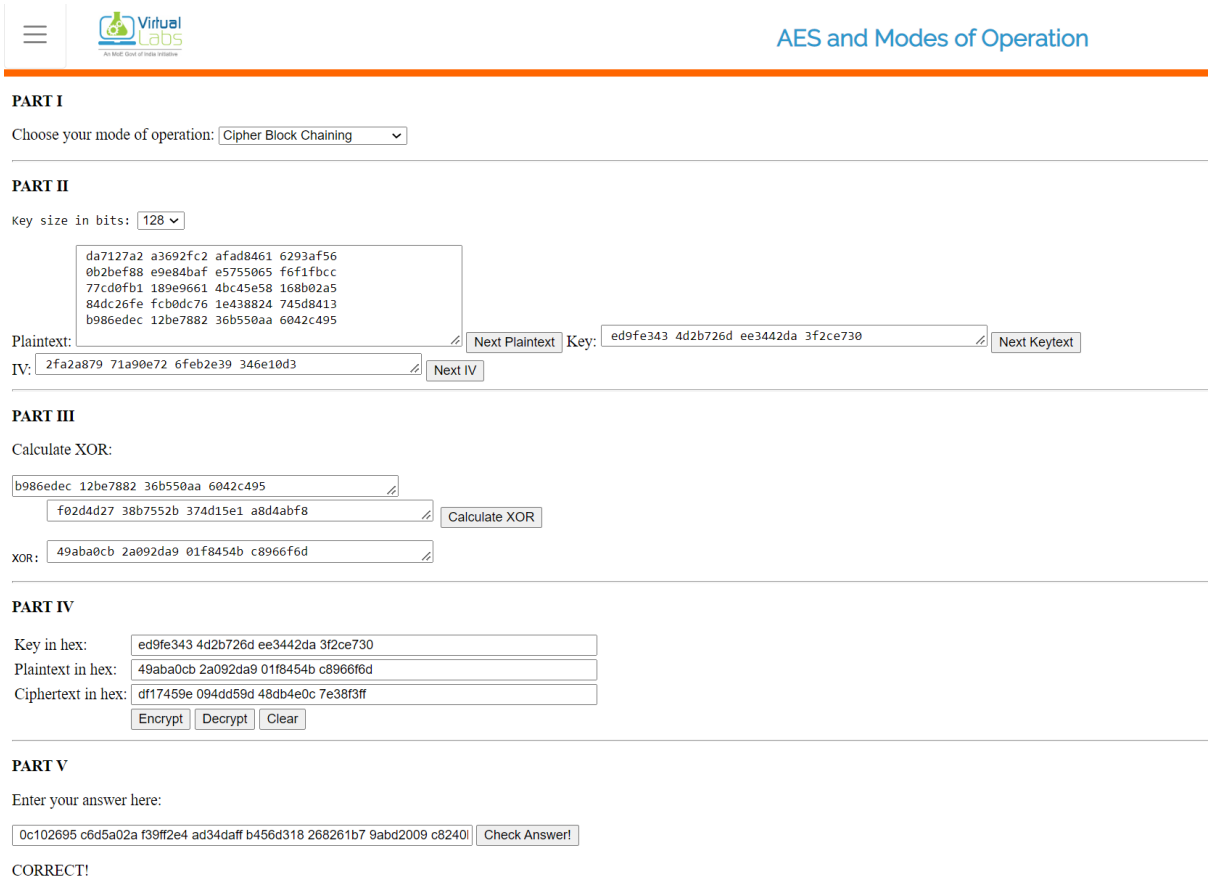
Ciphertext in hex: df17459e 094dd59d 48db4e0c 7e38f3ff

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240| Check Answer!



We repeat this procedure for all Plaintext

89801e77 e52dc587 59bd4788 365fa2c1
f02d4d27 38b7552b 374d15e1 a8d4abf8
df17459e 094dd59d 48db4e0c 7e38f3ff

Encryption: 0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240b4f 89801e77 e52dc587 59bd4788 365fa2c1 f02d4d27 38b7552b 374d15e1 a8d4abf8 df17459e 094dd59d 48db4e0c 7e38f3ff

Code:

```
import os
import sys
import math

class AES(object):
    '''AES funtions for a single block
    '''
```

```
    # Very annoying code:  all is for an object, but no state is kept!
    # Should just be plain functions in a AES modlule.

    # valid key sizes
    keySize = dict(SIZE_128=16, SIZE_192=24, SIZE_256=32)

    # Rijndael S-box
    sbox =  [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]

    # Rijndael Inverted S-box
    rsbox = [0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3,
            0x9e, 0x81, 0xf3, 0xd7, 0xfb , 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f,
            0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb , 0x54,
            0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b,
            0x42, 0xfa, 0xc3, 0x4e , 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24,
            0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25 , 0x72, 0xf8,
            0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
            0x65, 0xb6, 0x92 , 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
            0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84 , 0x90, 0xd8, 0xab,
            0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3,
            0x45, 0x06 , 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1,
            0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b , 0x3a, 0x91, 0x11, 0x41,
            0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,
            0x73 , 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9,
            0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e , 0x47, 0xf1, 0x1a, 0x71, 0x1d,
```

```
            0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b ,
            0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0,
            0xfe, 0x78, 0xcd, 0x5a, 0xf4 , 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07,
            0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f , 0x60,
            0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f,
            0x93, 0xc9, 0x9c, 0xef , 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5,
            0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61 , 0x17, 0x2b,
            0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
            0x21, 0x0c, 0x7d]

    def getSBoxValue(self,num):
        """Retrieves a given S-Box Value"""
        return self.sbox[num]

    def getSBoxInvert(self,num):
        """Retrieves a given Inverted S-Box Value"""
        return self.rsbox[num]

    def rotate(self, word):
        """ Rijndael's key schedule rotate operation.

        Rotate a word eight bits to the left: eg, rotate(1d2c3a4f) == 2c3a4f1d
        Word is an char list of size 4 (32 bits overall).
        """
        return word[1:] + word[:1]

    # Rijndael Rcon
    Rcon = [0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36,
            0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97,
            0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
            0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,
            0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
            0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
            0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
            0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
            0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
            0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
            0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc,
            0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
            0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
            0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
            0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
            0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
            0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
            0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
            0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
            0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
            0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
```

```
            0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2,
            0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
            0xe8, 0xcb ]

    def getRconValue(self, num):
        """Retrieves a given Rcon Value"""
        return self.Rcon[num]

    def core(self, word, iteration):
        """Key schedule core."""
        # rotate the 32-bit word 8 bits to the left
        word = self.rotate(word)
        # apply S-Box substitution on all 4 parts of the 32-bit word
        for i in range(4):
            word[i] = self.getSBoxValue(word[i])
        # XOR the output of the rcon operation with i to the first part
        # (leftmost) only
        word[0] = word[0] ^ self.getRconValue(iteration)
        return word

    def expandKey(self, key, size, expandedKeySize):
        """Rijndael's key expansion.

        Expands an 128,192,256 key into an 176,208,240 bytes key

        expandedKey is a char list of large enough size,
        key is the non-expanded key.
        """
        # current expanded keySize, in bytes
        currentSize = 0
        rconIteration = 1
        expandedKey = [0] * expandedKeySize

        # set the 16, 24, 32 bytes of the expanded key to the input key
        for j in range(size):
            expandedKey[j] = key[j]
        currentSize += size

        while currentSize < expandedKeySize:
            # assign the previous 4 bytes to the temporary value t
            t = expandedKey[currentSize-4:currentSize]

            # every 16,24,32 bytes we apply the core schedule to t
            # and increment rconIteration afterwards
            if currentSize % size == 0:
                t = self.core(t, rconIteration)
                rconIteration += 1
            # For 256-bit keys, we add an extra sbox to the calculation
```

```
            if size == self.keySize["SIZE_256"] and ((currentSize % size) ==
16):
                for l in range(4): t[l] = self.getSBoxValue(t[l])

            # We XOR t with the four-byte block 16,24,32 bytes before the new
            # expanded key.  This becomes the next four bytes in the expanded
            # key.
            for m in range(4):
                expandedKey[currentSize] = expandedKey[currentSize - size] ^ \
                        t[m]
                currentSize += 1

        return expandedKey

    def addRoundKey(self, state, roundKey):
        """Adds (XORs) the round key to the state."""
        for i in range(16):
            state[i] ^= roundKey[i]
        return state

    def createRoundKey(self, expandedKey, roundKeyPointer):
        """Create a round key.
        Creates a round key from the given expanded key and the
        position within the expanded key.
        """
        roundKey = [0] * 16
        for i in range(4):
            for j in range(4):
                roundKey[j*4+i] = expandedKey[roundKeyPointer + i*4 + j]
        return roundKey

    def galois_multiplication(self, a, b):
        """Galois multiplication of 8 bit characters a and b."""
        p = 0
        for counter in range(8):
            if b & 1: p ^= a
            hi_bit_set = a & 0x80
            a <<= 1
            # keep a 8 bit
            a &= 0xFF
            if hi_bit_set:
                a ^= 0x1b
            b >>= 1
        return p

    #
    # substitute all the values from the state with the value in the SBox
    # using the state value as index for the SBox
```

```
    #
    def subBytes(self, state, isInv):
        if isInv: getter = self.getSBoxInvert
        else: getter = self.getSBoxValue
        for i in range(16): state[i] = getter(state[i])
        return state

    # iterate over the 4 rows and call shiftRow() with that row
    def shiftRows(self, state, isInv):
        for i in range(4):
            state = self.shiftRow(state, i*4, i, isInv)
        return state

    # each iteration shifts the row to the left by 1
    def shiftRow(self, state, statePointer, nbr, isInv):
        for i in range(nbr):
            if isInv:
                state[statePointer:statePointer+4] = \
                        state[statePointer+3:statePointer+4] + \
                        state[statePointer:statePointer+3]
            else:
                state[statePointer:statePointer+4] = \
                        state[statePointer+1:statePointer+4] + \
                        state[statePointer:statePointer+1]
        return state

    # galois multiplication of the 4x4 matrix
    def mixColumns(self, state, isInv):
        # iterate over the 4 columns
        for i in range(4):
            # construct one column by slicing over the 4 rows
            column = state[i:i+16:4]
            # apply the mixColumn on one column
            column = self.mixColumn(column, isInv)
            # put the values back into the state
            state[i:i+16:4] = column

        return state

    # galois multiplication of 1 column of the 4x4 matrix
    def mixColumn(self, column, isInv):
        if isInv: mult = [14, 9, 13, 11]
        else: mult = [2, 1, 1, 3]
        cpy = list(column)
        g = self.galois_multiplication

        column[0] = g(cpy[0], mult[0]) ^ g(cpy[3], mult[1]) ^ \
                    g(cpy[2], mult[2]) ^ g(cpy[1], mult[3])
```

```
        column[1] = g(cpy[1], mult[0]) ^ g(cpy[0], mult[1]) ^ \
                    g(cpy[3], mult[2]) ^ g(cpy[2], mult[3])
        column[2] = g(cpy[2], mult[0]) ^ g(cpy[1], mult[1]) ^ \
                    g(cpy[0], mult[2]) ^ g(cpy[3], mult[3])
        column[3] = g(cpy[3], mult[0]) ^ g(cpy[2], mult[1]) ^ \
                    g(cpy[1], mult[2]) ^ g(cpy[0], mult[3])
        return column

    # applies the 4 operations of the forward round in sequence
    def aes_round(self, state, roundKey):
        state = self.subBytes(state, False)
        state = self.shiftRows(state, False)
        state = self.mixColumns(state, False)
        state = self.addRoundKey(state, roundKey)
        return state

    # applies the 4 operations of the inverse round in sequence
    def aes_invRound(self, state, roundKey):
        state = self.shiftRows(state, True)
        state = self.subBytes(state, True)
        state = self.addRoundKey(state, roundKey)
        state = self.mixColumns(state, True)
        return state

    # Perform the initial operations, the standard round, and the final
    # operations of the forward aes, creating a round key for each round
    def aes_main(self, state, expandedKey, nbrRounds):
        state = self.addRoundKey(state, self.createRoundKey(expandedKey, 0))
        i = 1
        while i < nbrRounds:
            state = self.aes_round(state,
                                   self.createRoundKey(expandedKey, 16*i))
            i += 1
        state = self.subBytes(state, False)
        state = self.shiftRows(state, False)
        state = self.addRoundKey(state,
                                 self.createRoundKey(expandedKey,
16*nbrRounds))
        return state

    # Perform the initial operations, the standard round, and the final
    # operations of the inverse aes, creating a round key for each round
    def aes_invMain(self, state, expandedKey, nbrRounds):
        state = self.addRoundKey(state,
                                 self.createRoundKey(expandedKey,
16*nbrRounds))
        i = nbrRounds - 1
        while i > 0:
```

```
            state = self.aes_invRound(state,
                                      self.createRoundKey(expandedKey, 16*i))
            i -= 1
        state = self.shiftRows(state, True)
        state = self.subBytes(state, True)
        state = self.addRoundKey(state, self.createRoundKey(expandedKey, 0))
        return state

    # encrypts a 128 bit input block against the given key of size specified
    def encrypt(self, iput, key, size):
        output = [0] * 16
        # the number of rounds
        nbrRounds = 0
        # the 128 bit block to encode
        block = [0] * 16
        # set the number of rounds
        if size == self.keySize["SIZE_128"]: nbrRounds = 10
        elif size == self.keySize["SIZE_192"]: nbrRounds = 12
        elif size == self.keySize["SIZE_256"]: nbrRounds = 14
        else: return None

        # the expanded keySize
        expandedKeySize = 16*(nbrRounds+1)

        # Set the block values, for the block:
        # a0,0 a0,1 a0,2 a0,3
        # a1,0 a1,1 a1,2 a1,3
        # a2,0 a2,1 a2,2 a2,3
        # a3,0 a3,1 a3,2 a3,3
        # the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3
        #
        # iterate over the columns
        for i in range(4):
            # iterate over the rows
            for j in range(4):
                block[(i+(j*4))] = iput[(i*4)+j]

        # expand the key into an 176, 208, 240 bytes key
        # the expanded key
        expandedKey = self.expandKey(key, size, expandedKeySize)

        # encrypt the block using the expandedKey
        block = self.aes_main(block, expandedKey, nbrRounds)

        # unmap the block again into the output
        for k in range(4):
            # iterate over the rows
            for l in range(4):
```

```
                output[(k*4)+l] = block[(k+(l*4))]
        return output

    # decrypts a 128 bit input block against the given key of size specified
    def decrypt(self, iput, key, size):
        output = [0] * 16
        # the number of rounds
        nbrRounds = 0
        # the 128 bit block to decode
        block = [0] * 16
        # set the number of rounds
        if size == self.keySize["SIZE_128"]: nbrRounds = 10
        elif size == self.keySize["SIZE_192"]: nbrRounds = 12
        elif size == self.keySize["SIZE_256"]: nbrRounds = 14
        else: return None

        # the expanded keySize
        expandedKeySize = 16*(nbrRounds+1)

        # Set the block values, for the block:
        # a0,0 a0,1 a0,2 a0,3
        # a1,0 a1,1 a1,2 a1,3
        # a2,0 a2,1 a2,2 a2,3
        # a3,0 a3,1 a3,2 a3,3
        # the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3

        # iterate over the columns
        for i in range(4):
            # iterate over the rows
            for j in range(4):
                block[(i+(j*4))] = iput[(i*4)+j]
        # expand the key into an 176, 208, 240 bytes key
        expandedKey = self.expandKey(key, size, expandedKeySize)
        # decrypt the block using the expandedKey
        block = self.aes_invMain(block, expandedKey, nbrRounds)
        # unmap the block again into the output
        for k in range(4):
            # iterate over the rows
            for l in range(4):
                output[(k*4)+l] = block[(k+(l*4))]
        return output


class AESModeOfOperation(object):
    '''Handles AES with plaintext consistingof multiple blocks.
    Choice of block encoding modes:  OFT, CFB, CBC
    '''
    # Very annoying code:  all is for an object, but no state is kept!
    # Should just be plain functions in an AES_BlockMode module.
```

```
    aes = AES()

    # structure of supported modes of operation
    modeOfOperation = dict(OFB=0, CFB=1, CBC=2)

    # converts a 16 character string into a number array
    def convertString(self, string, start, end, mode):
        if end - start > 16: end = start + 16
        if mode == self.modeOfOperation["CBC"]: ar = [0] * 16
        else: ar = []

        i = start
        j = 0
        while len(ar) < end - start:
            ar.append(0)
        while i < end:
            ar[j] = ord(string[i])
            j += 1
            i += 1
        return ar

    # Mode of Operation Encryption
    # stringIn - Input String
    # mode - mode of type modeOfOperation
    # hexKey - a hex key of the bit length size
    # size - the bit length of the key
    # hexIV - the 128 bit hex Initilization Vector
    def encrypt(self, stringIn, mode, key, size, IV):
        if len(key) % size:
            return None
        if len(IV) % 16:
            return None
        # the AES input/output
        plaintext = []
        iput = [0] * 16
        output = []
        ciphertext = [0] * 16
        # the output cipher string
        cipherOut = []
        # char firstRound
        firstRound = True
        if stringIn != None:
            for j in range(int(math.ceil(float(len(stringIn))/16))):
                start = j*16
                end = j*16+16
                if  end > len(stringIn):
                    end = len(stringIn)
                plaintext = self.convertString(stringIn, start, end, mode)
```

```
                # print 'PT@%s:%s' % (j, plaintext)
                if mode == self.modeOfOperation["CFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(plaintext)-1 < i:
                            ciphertext[i] = 0 ^ output[i]
                        elif len(output)-1 < i:
                            ciphertext[i] = plaintext[i] ^ 0
                        elif len(plaintext)-1 < i and len(output) < i:
                            ciphertext[i] = 0 ^ 0
                        else:
                            ciphertext[i] = plaintext[i] ^ output[i]
                    for k in range(end-start):
                        cipherOut.append(ciphertext[k])
                    iput = ciphertext
                elif mode == self.modeOfOperation["OFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(plaintext)-1 < i:
                            ciphertext[i] = 0 ^ output[i]
                        elif len(output)-1 < i:
                            ciphertext[i] = plaintext[i] ^ 0
                        elif len(plaintext)-1 < i and len(output) < i:
                            ciphertext[i] = 0 ^ 0
                        else:
                            ciphertext[i] = plaintext[i] ^ output[i]
                    for k in range(end-start):
                        cipherOut.append(ciphertext[k])
                    iput = output
                elif mode == self.modeOfOperation["CBC"]:
                    for i in range(16):
                        if firstRound:
                            iput[i] =  plaintext[i] ^ IV[i]
                        else:
                            iput[i] =  plaintext[i] ^ ciphertext[i]
                    # print 'IP@%s:%s' % (j, iput)
                    firstRound = False
                    ciphertext = self.aes.encrypt(iput, key, size)
                    # always 16 bytes because of the padding for CBC
                    for k in range(16):
```

```
                    cipherOut.append(ciphertext[k])
        return mode, len(stringIn), cipherOut

    # Mode of Operation Decryption
    # cipherIn - Encrypted String
    # originalsize - The unencrypted string length - required for CBC
    # mode - mode of type modeOfOperation
    # key - a number array of the bit length size
    # size - the bit length of the key
    # IV - the 128 bit number array Initilization Vector
    def decrypt(self, cipherIn, originalsize, mode, key, size, IV):
        # cipherIn = unescCtrlChars(cipherIn)
        if len(key) % size:
            return None
        if len(IV) % 16:
            return None
        # the AES input/output
        ciphertext = []
        iput = []
        output = []
        plaintext = [0] * 16
        # the output plain text character list
        chrOut = []
        # char firstRound
        firstRound = True
        if cipherIn != None:
            for j in range(int(math.ceil(float(len(cipherIn))/16))):
                start = j*16
                end = j*16+16
                if j*16+16 > len(cipherIn):
                    end = len(cipherIn)
                ciphertext = cipherIn[start:end]
                if mode == self.modeOfOperation["CFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(output)-1 < i:
                            plaintext[i] = 0 ^ ciphertext[i]
                        elif len(ciphertext)-1 < i:
                            plaintext[i] = output[i] ^ 0
                        elif len(output)-1 < i and len(ciphertext) < i:
                            plaintext[i] = 0 ^ 0
                        else:
                            plaintext[i] = output[i] ^ ciphertext[i]
                    for k in range(end-start):
```

```
                        chrOut.append(chr(plaintext[k]))
                    iput = ciphertext
                elif mode == self.modeOfOperation["OFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(output)-1 < i:
                            plaintext[i] = 0 ^ ciphertext[i]
                        elif len(ciphertext)-1 < i:
                            plaintext[i] = output[i] ^ 0
                        elif len(output)-1 < i and len(ciphertext) < i:
                            plaintext[i] = 0 ^ 0
                        else:
                            plaintext[i] = output[i] ^ ciphertext[i]
                    for k in range(end-start):
                        chrOut.append(chr(plaintext[k]))
                    iput = output
                elif mode == self.modeOfOperation["CBC"]:
                    output = self.aes.decrypt(ciphertext, key, size)
                    for i in range(16):
                        if firstRound:
                            plaintext[i] = IV[i] ^ output[i]
                        else:
                            plaintext[i] = iput[i] ^ output[i]
                    firstRound = False
                    if originalsize is not None and originalsize < end:
                        for k in range(originalsize-start):
                            chrOut.append(chr(plaintext[k]))
                    else:
                        for k in range(end-start):
                            chrOut.append(chr(plaintext[k]))
                    iput = ciphertext
        return "".join(chrOut)


def append_PKCS7_padding(s):
    """return s padded to a multiple of 16-bytes by PKCS7 padding"""
    numpads = 16 - (len(s)%16)
    return s + numpads*chr(numpads)

def strip_PKCS7_padding(s):
    """return s stripped of PKCS7 padding"""
    if len(s)%16 or not s:
        raise ValueError("String of len %d can't be PCKS7-padded" % len(s))
    numpads = ord(s[-1])
    if numpads > 16:
```

```
        raise ValueError("String ending with %r can't be PCKS7-padded" % s[-
1])
    return s[:-numpads]

def encryptData(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """encrypt `data` using `key`

    `key` should be a string of bytes.

    returned cipher is a string of bytes prepended with the initialization
    vector.

    """
    key = list(key)
    if mode == AESModeOfOperation.modeOfOperation["CBC"]:
        data = append_PKCS7_padding(data)
    keysize = len(key)
    assert keysize in list(AES.keySize.values()), 'invalid key size: %s' %
keysize
    # create a new iv using random data
    iv = [i for i in os.urandom(16)]
    moo = AESModeOfOperation()
    (mode, length, ciph) = moo.encrypt(data, mode, key, keysize, iv)
    # With padding, the original length does not need to be known. It's a bad
    # idea to store the original message length.
    # prepend the iv.
    return ''.join(map(chr, iv)) + ''.join(map(chr, ciph))

def decryptData(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """decrypt `data` using `key`

    `key` should be a string of bytes.

    `data` should have the initialization vector prepended as a string of
    ordinal values.
    """

    key = list(key)
    keysize = len(key)
    assert keysize in list(AES.keySize.values()), 'invalid key size: %s' %
keysize
    # iv is first 16 bytes
    iv = list(map(ord, data[:16]))
    data = list(map(ord, data[16:]))
    moo = AESModeOfOperation()
    decr = moo.decrypt(data, None, mode, key, keysize, iv)
    if mode == AESModeOfOperation.modeOfOperation["CBC"]:
        decr = strip_PKCS7_padding(decr)
```

```
    return decr

def generateRandomKey(keysize):
    """Generates a key from random data of length `keysize`.
    The returned key is a string of bytes.
    """
    if keysize not in (16, 24, 32):
        emsg = 'Invalid keysize, %s. Should be one of (16, 24, 32).'
        raise ValueError(emsg % keysize)
    return os.urandom(keysize)

def testStr(cleartext, keysize=16, modeName = "CBC"):
    '''Test with random key, choice of mode.'''
    print('Random key test', 'Mode:', modeName)
    print('cleartext:', cleartext)
    key =  generateRandomKey(keysize)
    print('Key:', [x for x in key])
    mode = AESModeOfOperation.modeOfOperation[modeName]
    cipher = encryptData(key, cleartext, mode)
    print('Cipher:', [ord(x) for x in cipher])
    decr = decryptData(key, cipher, mode)
    print('Decrypted:', decr)

def convertToList(keyString):
    lst = []
    for k in keyString:
        lst.append(ord(k))
    return lst

if __name__ == "__main__":
    moo = AESModeOfOperation()
    print("-- AES Encryption and Decryption --")
    print("Enter Plaintext: ", end='')
    cleartext = input()
    print("Enter Cipher Key (16 characters): ", end='')
    cypherkey = input()
    cypherkey = convertToList(cypherkey)
    iv = [103,35,148,239,76,213,47,118,255,222,123,176,106,134,98,92]
    mode, orig_len, ciph = moo.encrypt(cleartext, moo.modeOfOperation["CBC"],
            cypherkey, moo.aes.keySize["SIZE_128"], iv)
    print('m=%s, ol=%s (%s), ciph=%s' % (mode, orig_len, len(cleartext),
ciph))
    decr = moo.decrypt(ciph, orig_len, mode, cypherkey,
            moo.aes.keySize["SIZE_128"], iv)
    print(decr)
    testStr(cleartext, 16, "CBC")
```

We now illustrate with examples:

The code is for CBC chaining mode of the AES.

We encrypt – ‘Devang is a Developer’

And let the key be – ‘WalchandCollege1’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives> cd .\AESCipher\
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher> py .\script.py
-- AES Encryption and Decryption --
Enter Plaintext: Devang is a Developer
Enter Cipher Key (16 characters): WalchandCollege1
m=2, ol=21 (21), ciph=[48, 191, 121, 70, 86, 183, 52, 78, 217, 201, 181, 233, 84, 84, 197, 210, 229
, 216, 244, 136, 91, 87, 100, 187, 76, 164, 203, 69, 36, 228, 110, 154]
Devang is a Developer
Random key test Mode: CBC
cleartext: Devang is a Developer
Key: [187, 38, 90, 176, 243, 85, 253, 148, 42, 57, 81, 154, 139, 232, 75, 115]
Cipher: [174, 116, 198, 99, 147, 99, 55, 113, 32, 15, 27, 195, 192, 175, 244, 2, 82, 175, 242, 239,
 234, 226, 185, 238, 240, 106, 227, 177, 132, 29, 193, 39, 60, 84, 62, 37, 252, 7, 98, 248, 35, 89,
 121, 210, 228, 87, 252, 189]
Decrypted: Devang is a Developer
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher>
```

As shown, the encryption generated cipher-text as numbers (we don’t convert them to ASCII to avoid vernacular characters).

Supplying Key and Ciphertext to the AES decryption algorithm, returns our our plaintext again.

Let’s look at another example, a larger one.

‘The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God wills it!’

Let the key be, ‘QueenElizabethII’

Result of the encryption algorithm:

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher> py .\script.py
-- AES Encryption and Decryption --
Enter Plaintext: The King has left the tower. Prepare all soldiers to attack in the valley by sunse
t. God wills it!
Enter Cipher Key (16 characters): QueenElizabethII
m=2, ol=98 (98), ciph=[135, 97, 3, 91, 71, 194, 170, 51, 116, 26, 6, 236, 143, 36, 255, 170, 47, 97
, 197, 57, 63, 248, 155, 134, 242, 98, 113, 4, 111, 207, 68, 133, 115, 240, 222, 213, 53, 14, 3, 10
3, 151, 238, 174, 14, 106, 112, 152, 204, 128, 208, 239, 123, 18, 8, 190, 11, 150, 54, 215, 119, 57
, 170, 242, 177, 185, 134, 216, 251, 207, 111, 15, 55, 0, 96, 47, 12, 118, 39, 18, 27, 0, 221, 161,
 216, 145, 148, 233, 148, 88, 123, 162, 210, 240, 50, 188, 67, 122, 215, 14, 35, 208, 147, 18, 217,
 185, 59, 18, 64, 48, 175, 152, 89]
The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God wills it!
Random key test Mode: CBC
```

Decryption Algorithm:

```
Key: [128, 63, 131, 5, 143, 187, 34, 236, 38, 55, 219, 55, 193, 144, 139, 49]
Cipher: [171, 144, 32, 102, 25, 143, 239, 254, 249, 255, 133, 241, 218, 107, 247, 59, 127, 183, 2,
36, 123, 167, 65, 49, 195, 177, 222, 178, 89, 76, 104, 155, 181, 144, 99, 64, 190, 217, 130, 142, 5
8, 189, 126, 55, 136, 45, 134, 42, 160, 187, 171, 59, 142, 23, 51, 35, 97, 61, 233, 101, 220, 20, 1
43, 163, 254, 220, 234, 210, 235, 175, 145, 221, 229, 172, 123, 1, 110, 241, 165, 223, 123, 9, 102,
 79, 121, 195, 138, 213, 212, 240, 126, 255, 74, 133, 137, 215, 159, 145, 253, 251, 207, 95, 212, 5
6, 177, 106, 201, 6, 225, 253, 33, 195, 241, 165, 88, 225, 25, 15, 7, 78, 207, 35, 66, 146, 37, 175
, 197, 231]
Decrypted: The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God
 wills it!
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher>
```

We get our plaintext back!

Thus, we demonstrated the working of the code with examples.

Conclusion:

Thus, the Advanced Encryption Standard (AES) algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 8
Implementation and Understanding of Euclid and Extended Euclid's Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Euclid and Extended Euclid's Algorithm

Aim: To Study, Implement and Demonstrate the:

- Euclid's Algorithm
- Extended Euclid's Algorithm

Euclid's Algorithm

Theory:

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

Basic Euclidean Algorithm for GCD:
The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.

- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.

Let’s illustrate with an example:

We wish to calculate the GCD of 1220 and 516.

1220 and 516

We first compute – 1220 % 516 = 188

Then, 516 % 188 = 140

Then 188 % 140 = 48

Then 140 % 48 = 44

Then 48 % 44 = 4

Then 44 % 4 = 0

Therefore, GCD is 4

This is computed in a tabular form. The operation of GCD gives other useful results, which we shall see in the Extended Euclid Theorem.

Code:

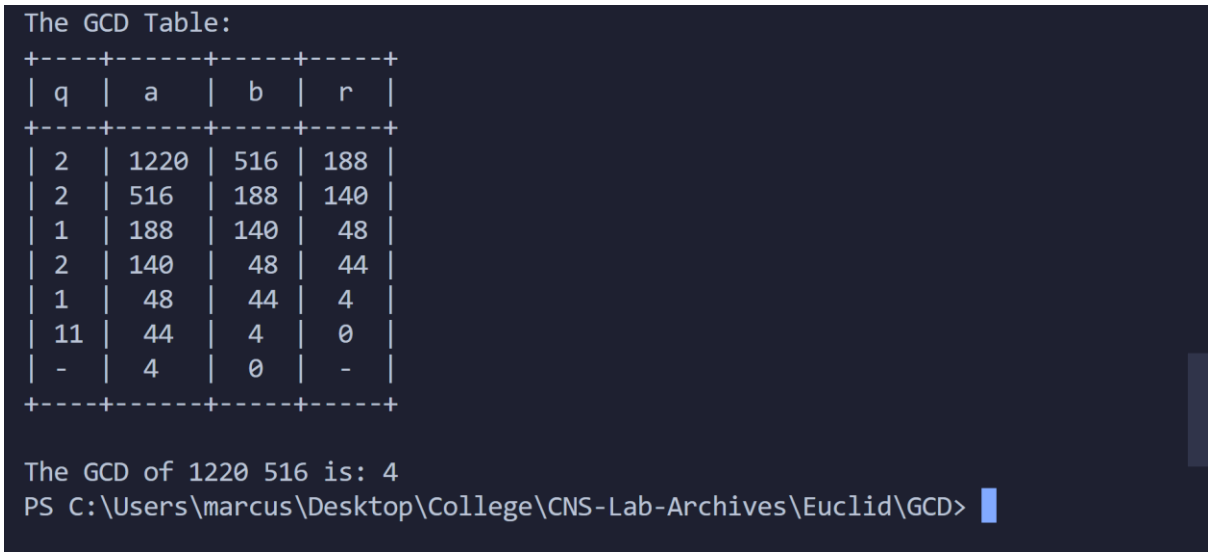
```
from prettytable import PrettyTable

def gcd(a, b):
    # ensure that a is always the larger number
    if b > a:
        temp = a
        a = b
        b = temp
    theGCDData = []
    q = a//b
    r =  a%b
    theGCDData.append([q, a, b, r])
    while(b>0):
        a = b
```

```
        b = r
        if b == 0:
            theGCDData.append(["-", a, b, "-"])
            break
        q = a//b
        r = a%b
        theGCDData.append([q, a, b, r])
    return theGCDData


print("Enter the 2 numbers:\n")
print("First Number: ", end='')
a = int(input())
print("Second Number: ", end='')
b = int(input())

# Initialise the table
x = PrettyTable()
x.field_names = ["q", "a", "b", "r"]

theGCDData = gcd(a,b)
for i in theGCDData:
    x.add_row(i)

print("\nThe GCD Table:")
print(x)
print("\nThe GCD of "+str(a)+" "+str(b)+" is:
"+str(theGCDData[len(theGCDData)-1][1]))
```

We now demonstrate with an example

Let’s compute GCD of 1220 and 516 as in the illustration.

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Euclid\GCD> py .\script.
py
Enter the 2 numbers:

First Number: 1220
Second Number: 516

The GCD Table:
+----+------+-----+-----+
| q  |  a   |  b  |  r  |
+----+------+-----+-----+
| 2  | 1220 | 516 | 188 |
| 2  | 516  | 188 | 140 |
| 1  | 188  | 140 |  48 |
| 2  | 140  |  48 |  44 |
| 1  |  48  |  44 |  4  |
| 11 |  44  |  4  |  0  |
| -  |  4   |  0  |  -  |
+----+------+-----+-----+
```



Therefore, the GCD is 4.

Let’s take another example:

We compute GCD of 9973 and 1009

```
First Number: 9973
Second Number: 1009

The GCD Table:
+----+------+------+-----+
| q  |  a   |  b   |  r  |
+----+------+------+-----+
| 9  | 9973 | 1009 | 892 |
| 1  | 1009 | 892  | 117 |
| 7  | 892  | 117  |  73 |
| 1  | 117  |  73  |  44 |
| 1  |  73  |  44  |  29 |
| 1  |  44  |  29  |  15 |
| 1  |  29  |  15  |  14 |
| 1  |  15  |  14  |  1  |
| 14 |  14  |  1   |  0  |
| -  |  1   |  0   |  -  |
+----+------+------+-----+

The GCD of 9973 1009 is: 1
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Euclid\GCD>
```

The GCD is 1, thus, the numbers are co-prime.

Hence, we demonstrated Euclid’s Algorithm.

Extended Eulid’s Algorithm

Theory

In arithmetic and computer programming, the extended Euclidean algorithm is an extension to the Euclidean algorithm, and computes, in addition to the greatest common divisor (gcd) of integers a and b, also the coefficients of Bézout's identity, which are integers x and y such that:

$$ax + by = gcd(a,b)$$

This is a certifying algorithm, because the gcd is the only number that can simultaneously satisfy this equation and divide the inputs. It allows one to compute also, with almost no extra cost, the quotients of a and b by their greatest common divisor.

Extended Euclidean algorithm also refers to a very similar algorithm for computing the polynomial greatest common divisor and the coefficients of Bézout's identity of two univariate polynomials.

The extended Euclidean algorithm is particularly useful when a and b are coprime. With that provision, x is the modular multiplicative inverse of a modulo b, and y is the modular multiplicative inverse of b modulo a. Similarly, the polynomial extended Euclidean algorithm allows one to compute the multiplicative inverse in algebraic field extensions and, in particular in finite fields of non prime order. It follows that both extended Euclidean algorithms are widely used in cryptography. In particular, the computation of the modular multiplicative inverse is an essential step in the derivation of key-pairs in the RSA public-key encryption method.

Following our Euclid's algorithm, we introduce 3 columns to our table – t1, t2 and t, t1 and t2 initialized as 0 and 1. Then we perform computation to obtain t. Following this, we get the values to obtain the multiplicative inverse of a number.

Code:

```
from prettytable import PrettyTable

def gcd(a, b):
    # ensure that a is always the larger number
    if b > a:
        temp = a
        a = b
        b = temp
    theGCDData = []
    q = a//b
    r =  a%b
    t1 = 0
    t2 = 1
    t = t1 - t2*q
    theGCDData.append([q, a, b, r, t1, t2, t])
    while(b>0):
        a = b
        b = r
        t1 = t2
        t2 = t
        if b == 0:
            theGCDData.append(["-", a, b, "-", t1, t2, "-"])
```

```
            break
        q = a//b
        r = a%b
        t = t1 - q*t2
        theGCDData.append([q, a, b, r, t1, t2, t])
    return theGCDData

print("\n-- Multiplicative Inverse using Extended Euclid Algo --\n")
print("Enter Number a: ", end='')
a = int(input())
print("Enter Number b: ", end='')
b = int(input())

# Initialise the table
x = PrettyTable()
x.field_names = ["q", "a", "b", "r", "t1", "t2", "t"]

theGCDData = gcd(a,b)
for i in theGCDData:
    x.add_row(i)

print("\nThe Extended Euclid Table:")
print(x)
print("\nThe M.I. of "+str(a)+"%"+str(b)+" is:
"+str(theGCDData[len(theGCDData)-1][4]))
```

Now, let’s see demonstrate with example.

We wish to compute multiplicative inverse of 37 mod 50

Then,

```
-- Multiplicative Inverse using Extended Euclid Algo --

Enter Number a: 37
Enter Number b: 50

The Extended Euclid Table:
+---+----+----+----+-----+-----+-----+
| q | a  | b  | r  | t1  |  t2 |  t  |
+---+----+----+----+-----+-----+-----+
| 1 | 50 | 37 | 13 | 0   |  1  |  -1 |
| 2 | 37 | 13 | 11 | 1   |  -1 |  3  |
| 1 | 13 | 11 | 2  | -1  |  3  |  -4 |
| 5 | 11 | 2  | 1  | 3   |  -4 |  23 |
| 2 | 2  | 1  | 0  | -4  |  23 | -50 |
| - | 1  | 0  | -  | 23  | -50 |  -  |
+---+----+----+----+-----+-----+-----+

The M.I. of 37%50 is: 23
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Euclid\Extended>
```

We find that the Modular Inverse is 23.

We can verify it as – a * a(-1) mod b must equal 1

37 * 23 = 851

851 mod 50 = 1

Hence verified.

Now, we take another example:

Say we wish to compute 34 mod 67

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Euclid\Extended> ^C
ript.py

-- Multiplicative Inverse using Extended Euclid Algo --

Enter Number a: 34
Enter Number b: 67

The Extended Euclid Table:
+----+----+----+----+----+-----+-----+
| q  | a  | b  | r  | t1 |  t2 |  t  |
+----+----+----+----+----+-----+-----+
| 1  | 67 | 34 | 33 | 0  |  1  |  -1 |
| 1  | 34 | 33 | 1  | 1  |  -1 |  2  |
| 33 | 33 | 1  | 0  | -1 |  2  | -67 |
| -  | 1  | 0  | -  | 2  | -67 |  -  |
+----+----+----+----+----+-----+-----+

The M.I. of 34%67 is: 2
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\Euclid\Extended>
```

We get modular inverse as 2.

We can verify as:

34 * 2 = 68

68 mod 67 = 1

We thus illustrated Extended Euclids Algorithm with example.

Conclusion:

Thus, the Euclid's algorithm and Extended Euclid's algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 9
Implementation and Understanding of Chinese Remainder Theorem

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Chinese Remainder Theorem (CRT)

Aim: To Study, Implement and Demonstrate the Chinese Remainder Theorem (CRT)

Theory:

In mathematics, the Chinese remainder theorem states that if one knows the remainders of the Euclidean division of an integer n by several integers, then one can determine uniquely the remainder of the division of n by the product of these integers, under the condition that the divisors are pairwise coprime (no two divisors share a common factor other than 1).

For example, if we know that the remainder of n divided by 3 is 2, the remainder of n divided by 5 is 3, and the remainder of n divided by 7 is 2, then without knowing the value of n, we can determine that the remainder of n divided by 105 (the product of 3, 5, and 7) is 23. Importantly, this tells us that if n is a natural number less than 105, then 23 is the only possible value of n.

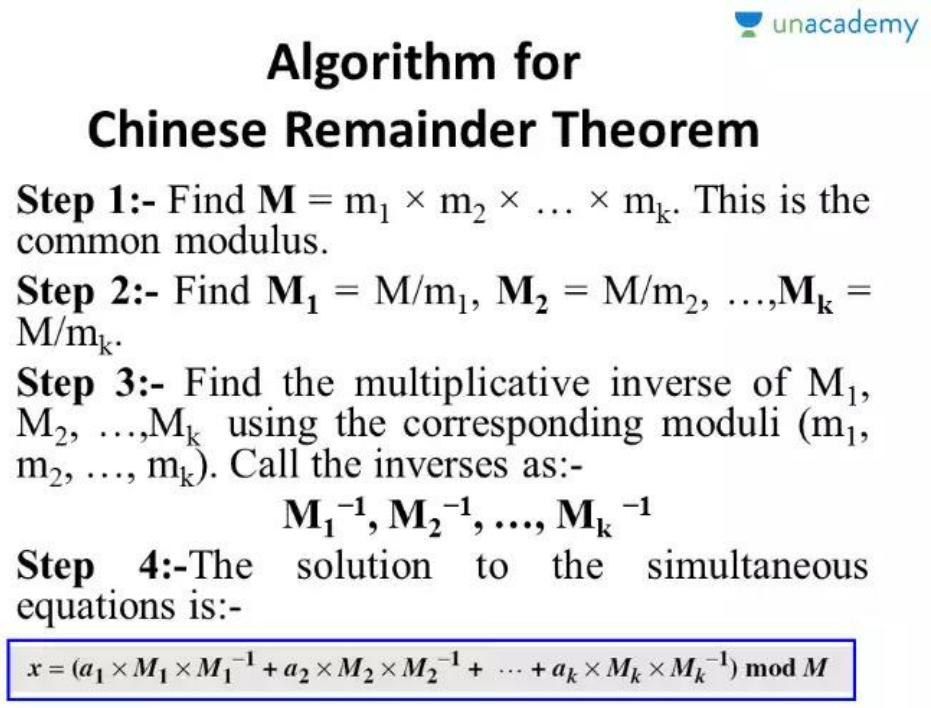
The earliest known statement of the theorem is by the Chinese mathematician Sun-tzu in the Sun-tzu Suan-ching in the 3rd century CE.

The Chinese remainder theorem is widely used for computing with large integers, as it allows replacing a computation for which one knows a bound on the size of the result by several similar computations on small integers.

CRT typically gives the smallest possible solution, but it's not the only solution. There are infinite number of solutions possible for CRT equations.

We make use of the Extended Euclid's algorithm of previous practical for the computation of CRT.

The algorithm to compute CRT is as follows:



We now therefore, code for the algorithm.

Code:

```
# We use the Extended Euclidean Function from last assignment
def computeInverse(a, b):
    # ensure that a is always the larger number
    if b > a:
        temp = a
        a = b
        b = temp
    amt = 1
    while(True):
        if((a*amt) % b == 1):
            return amt
        amt += 1
```

```
def computeCRT(lst):
    M = 1
    for i in range(len(lst)):
        M *= lst[i][1]
    valuesOfMs = []
    for i in range(len(lst)):
        print(lst[i][1])
        print(M//lst[i][1])
        print(computeInverse(lst[i][1], M//lst[i][1]))
        print()
        valuesOfMs.append([M//lst[i][1], computeInverse(lst[i][1],
M//lst[i][1])])
    print("\nValue of M: "+str(M)+"\n")
    for i in range(len(lst)):
        print("a"+str(i+1)+" = "+str(lst[i][0])+" M"+str(i+1)+" =
"+str(valuesOfMs[i][0])+", M"+str(i+1)+" -1 = "+str(valuesOfMs[i][1]))
    theModularValue = 0
    for i in range(len(lst)):
        theModularValue += (lst[i][0] * valuesOfMs[i][0] * valuesOfMs[i][1])
        theModularValue %= M
    return theModularValue, M

print("-- Chinese Remainder Theorem --\n")

print("Enter number of equations: ", end='')
n = int(input())

equations = []
for i in range(n):
    print("Enter number: ", end='')
    num = int(input())
    print("Enter modulo: ", end='')
    mod = int(input())
    print()
    equations.append([num, mod])
ans, M = computeCRT(equations)
print("\nOur M is "+str(M))
print("\nThe first solution is: "+str(ans))

print("\nTherefore our solutions are "+str(ans)+", "+str(ans+M)+",
"+str(ans+(2*M))+" and so on...")

# 2 3
# 3 5
# 2 7
```

We now illustrate with examples:

We illustrate first for 3 equations:

2 3

3 5

2 7

We get:

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\CRT> py .\script.py
-- Chinese Remainder Theorem --

Enter number of equations: 3
Enter number: 2
Enter modulo: 3

Enter number: 3
Enter modulo: 5

Enter number: 2
Enter modulo: 7
```

```
Value of M: 105

a1 = 2 M1 = 35, M1 -1 = 2
a2 = 3 M2 = 21, M2 -1 = 1
a3 = 2 M3 = 15, M3 -1 = 1

Our M is 105

The first solution is: 23

Therefore our solutions are 23, 128, 233 and so on...
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\CRT>
```

All solutions of the form 23 + n*105 are valid.

Let’s take another example:

We take 2 equations:

3 5

6 8

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\CRT> py .\script.py
-- Chinese Remainder Theorem --

Enter number of equations: 2
Enter number: 3
Enter modulo: 5

Enter number: 6
Enter modulo: 8
```

Solution:

```
 Value of M: 40

 a1 = 3 M1 = 8, M1 -1 = 2
 a2 = 6 M2 = 5, M2 -1 = 2

 Our M is 40

 The first solution is: 28

 Therefore our solutions are 28, 68, 108 and so on...
 PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\CRT>
```

We get solutions 28, 68, 108, 148 and so on.

Thus, we demonstrated the working of the code with examples.

Conclusion:

Thus, the Chinese Remainder Theorem algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 10
Implementation and Understanding of RSA Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of RSA Algorithm

Aim: To Study, Implement and Demonstrate the RSA Algorithm

Theory:

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission. It is also one of the oldest. The acronym "RSA" comes from the surnames of Ron Rivest, Adi Shamir and Leonard Adleman, who publicly described the algorithm in 1977. An equivalent system was developed secretly in 1973 at GCHQ (the British signals intelligence agency) by the English mathematician Clifford Cocks. That system was declassified in 1997.

In a public-key cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private). An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers.

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used.

RSA is a relatively slow algorithm. Because of this, it is not commonly used to directly encrypt user data. More often, RSA is used to transmit shared keys for symmetric-key cryptography, which are then used for bulk encryption–decryption.

Operation:

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

A basic principle behind RSA is the observation that it is practical to find three very large positive integers e, d, and n, such that with modular exponentiation for all integers m (with $0 \leq m < n$):

$$(m^e)^d \equiv m \pmod{n}$$

and that knowing e and n, or even m, it can be extremely difficult to find d. The triple bar (≡) here denotes modular congruence (which is to say that when you divide (me)d by n and m by n, they both have the same remainder).

In addition, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies:

$$(m^d)^e \equiv m \pmod{n}.$$

RSA involves a public key and a private key. The public key can be known by everyone and is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time by using the private key. The public key is represented by the integers n and e, and the private key by the integer d (although n is also used during the decryption process, so it might be considered to be a part of the private key too). m represents the message (previously prepared with a certain technique explained below).

This is the crux of the RSA algorithm.

Essentially, we first convert the string to numbers, then encrypt the large number. Decryption is similar – we get our original number which can be converted to the plaintext.

Code:

```
from os import system, name
import msvcrt
import rsa
import sys

def ConvertToInt(message):
    theNumber = "1"
    for m in message:
        theASCII = str(ord(m))
        if len(theASCII) == 1:
            theASCII = "00"+theASCII
        elif len(theASCII) == 2:
            theASCII = "0"+theASCII
        theNumber += theASCII
    return int(theNumber)

def clear():
    if name == 'nt':
        _ = system('cls')
    else:
        _ = system('clear')

def Generate_Keys():
    clear()
    print("\n\n**********Key
Generation**********\n")
    (pubkey, privkey) = rsa.newkeys(512)
    print("Your Public Keys 'e' and 'n' respectively are:")
    print(pubkey.e)
    print()
    print(pubkey.n)
    print("\nYour Private Keys 'd' and 'n' respectively are:")
    print(privkey.d)
    print()
    print(privkey.n)
    print("Warning: Don't share your Private Keys with Anyone!")
    print("Press Any Key to Go Back")
    msvcrt.getch()
    home()

def Encrypt_Dat():
    clear()
    print("\n\n**********Encryption**********\n"
)
    print("Enter the Message to Encrypt:")
    contents = []
    while True:
```

```
        try:
            line = input()
            line.lstrip()
            line.rstrip()
            if len(line) == 0:
                break
        except EOFError:
            break
        contents.append(line)
    print("Enter 'n' of Receiver:")
    n = input()
    print("Enter 'e' of Receiver:")
    e = input()
    encry = []
    print("\nEncoded Message is:")
    for line in contents:
        mess = ConvertToInt(line)
        mess = pow(mess, int(e), int(n))
        encry.append(mess)
    for line in encry:
        print(line)

    print("\nPress Any Key to Continue.")
    msvcrt.getch()
    home()

def PowMod(a, n, mod):
    if n == 0:
        return 1 % mod
    elif n == 1:
        return a % mod
    else:
        b = PowMod(a, n // 2, mod)
        b = b * b % mod
        if n % 2 == 0:
          return b
        else:
          return b * a % mod


def ExtendedEuclid(a, b):
    if b == 0:
        return (1, 0)
    (x, y) = ExtendedEuclid(b, a % b)
    k = a // b
    return (y, x - k * y)

def InvertModulo(a, n):
    (b, x) = ExtendedEuclid(a, n)
```

```
    if b < 0:
        b = (b % n + n) % n
    return b

def Decrypt(ciphertext, d, n):
  return ConvertToStr(PowMod(ciphertext, d, n))

def ConvertToStr(numbers):
    numbers = str(numbers)
    numbers = numbers[1:]
    theMessageList = [numbers[i:i+3] for i in range(0, len(numbers), 3)]
    theMessage = ""
    for tml in theMessageList:
        theMessage += chr(int(tml))
    return theMessage

def Decrypt_Dat():
    clear()
    print("\n\n************************Decryption************************\n"
)
    print("Enter Message to Decrypt:")
    obey = []
    while True:
        try:
            line = input()
            line.lstrip()
            line.rstrip()
            if len(line) == 0:
                break
        except EOFError:
            break
        obey.append(int(line))

    # print("\nEnter Private Key 'p':")
    # p = int(input())
    # print("\nEnter Private Key 'q':")
    # q = int(input())
    print("\nEnter Private Key 'd':")
    d = int(input())
    print("\nEnter Your Public Key 'n':")
    n = int(input())

    print("\n\nMessage as Decrypted:")
    for line in obey:
        print(Decrypt(line, d, n))

    print("\nPress Any Key to Continue.")
    msvcrt.getch()
```

```
    home()

def home():
    clear()
    print("\n\n**********************DeSipher**********************\n")
    print("Choose your Action:")
    print("\t1. Encrypt Data.")
    print("\t2. Decrypt Data.")
    print("\t3. Generate Public and Private Keys.")
    print("\t4. Exit.")
    print("\nYour Choice: ", end='')
    inp = input()
    if inp == '1':
        Encrypt_Dat()
    elif inp == '2':
        Decrypt_Dat()
    elif inp == '3':
        Generate_Keys()
    elif inp == '4':
        sys.exit()
    else:
        print("Invalid Choice.")
        print("Try Again.")
        print("Press Any Key To Continue.")
        msvcrt.getch()
        home()

home()
```

We now illustrate with examples:

We need to first generate 2 public and private keys:

For our 2 users – Alice and Bob.

Your Public Keys 'e' and 'n' respectively are:

65537

87043814027724142339292066801941057591567348120899725468519830946784610152732575304465688320643283854369546021351510922996618745735634343468366461432006
19

Your Private Keys 'd' and 'n' respectively are:

62158025184208765452780394682863748650157191999299741396870898703167977659179841807715160369446897354632364937297852227168991886385967185257248613249278
73

87043814027724142339292066801941057591567348120899725468519830946784610152732575304465688320643283854369546021351510922996618745735634343468366461432006
19

Then for Bob:

Your Public Keys 'e' and 'n' respectively are:

65537

949277258945385888471004808509376889966489381481800598256953180213712205905128352828928419373759817660066631824943949731213580368623959226927671738649333 9

Your Private Keys 'd' and 'n' respectively are:

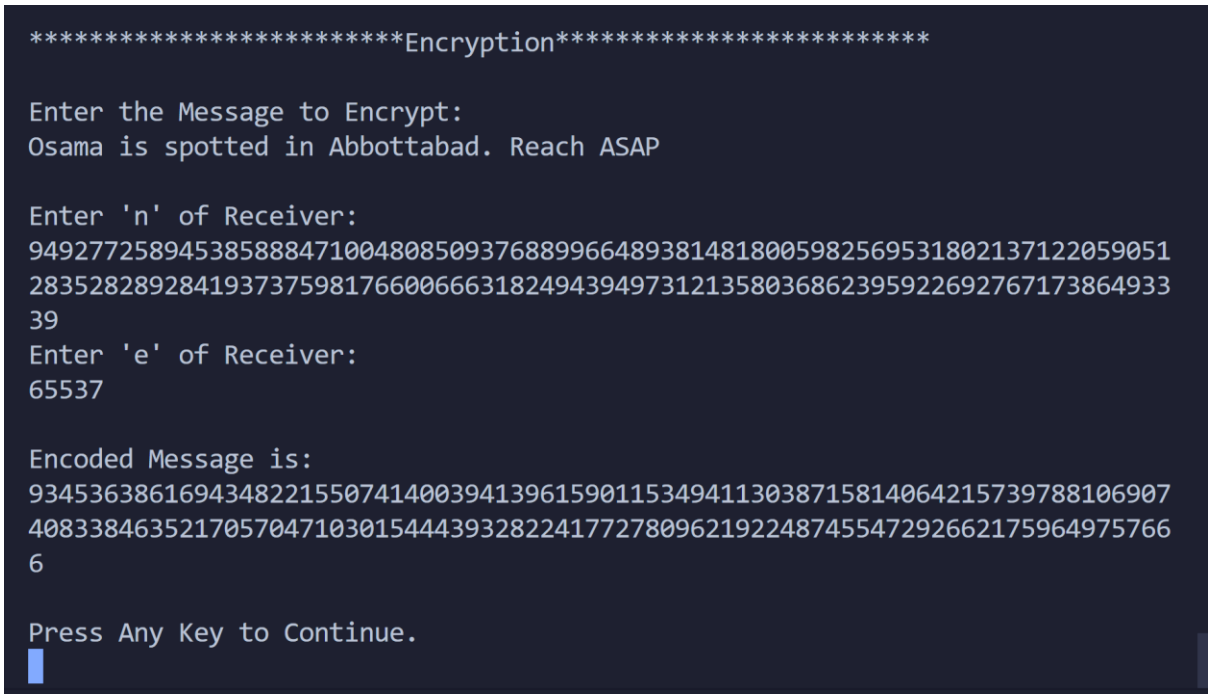
505338740102333382318025172944674625587849330014145586035195119570563062387872968578277608145415231141982570785939044527490635516717329183711721278650041 7

949277258945385888471004808509376889966489381481800598256953180213712205905128352828928419373759817660066631824943949731213580368623959226927671738649333 9

Now, say Alice wishes to share a message to Bob. She would send the message using Bob’s public key.

She wishes to send – ‘Osama is spotted in Abbottabad. Reach ASAP’.

This would work as follows:



Encoded Message is:

93453638616943482215507414003941396159011534941130387158140642157397881069074083384635217057047103015444393282241772780962192248745547292662175964975766

When Bob would receive this message, he would decrypt using his private key.

```
************************Decryption************************

Enter Message to Decrypt:
9345363861694348221550741400394139615901153494113038715814064215739788106907408338463521705704710301544439328224177278096219224874554729266217596497576666


Enter Private Key 'd':
5053387401023333823180251729446746255878493300141455860351951195705630623878729685782776081454152311419825707859390445274906355167173291837117212786500417

Enter Your Public Key 'n':
9492772589453858884710048085093768899664893814818005982569531802137122059051283528289284193737598176600666318249439497312135803686239592269276717386493339


Message as Decrypted:
Osama is spotted in Abbottabad. Reach ASAP

Press Any Key to Continue.
```

We receive our plaintext back.

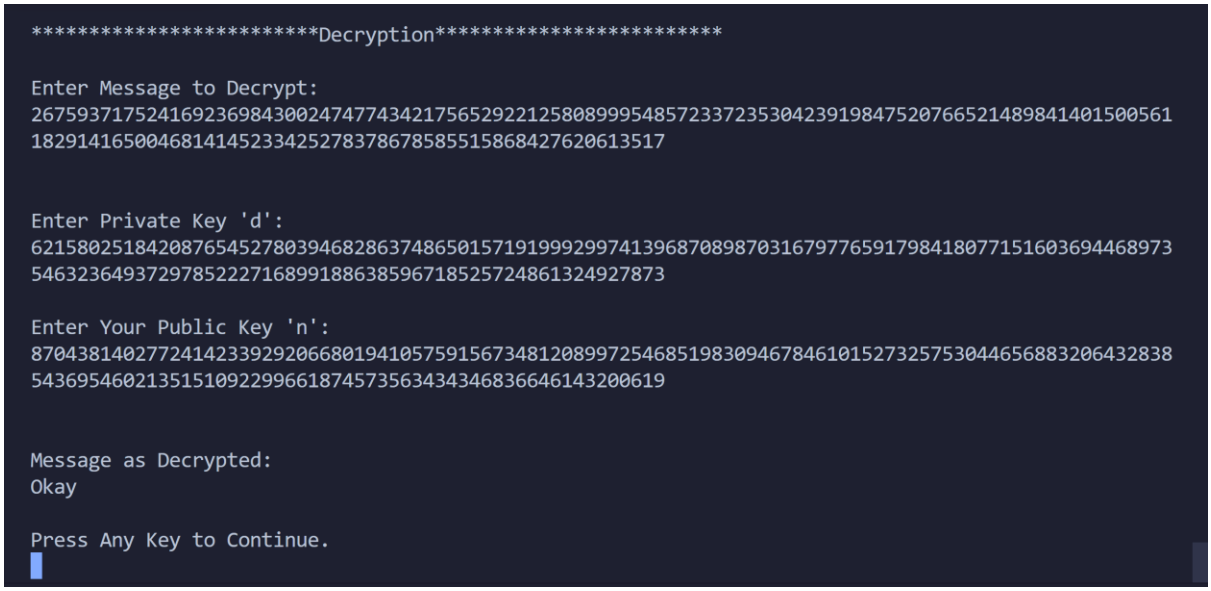
Now say Bob wishes to message ‘Okay’ to Alice.

```
************************Encryption************************

Enter the Message to Encrypt:
Okay

Enter 'n' of Receiver:
8704381402772414233929206680194105759156734812089972546851983094678461015273257530446568832064328385436954602135151092299661874573563434346836646143200619
Enter 'e' of Receiver:
65537

Encoded Message is:
2675937175241692369843002474774342175652922125808999548572337235304239198475207665214898414015005611829141650046814145233425278378678585515868427620613517

Press Any Key to Continue.
```

Encoded Message is:

2675937175241692369843002474774342175652922125808999548572337235304239198475207665214898414015005611829141650046814145233425278378678585515868427620613517

Then Alice would decrypt using her private key:



Thus, she received the message from Bob.

Thu, we illustrated working of RSA in the code.

Conclusion:

Thus, the RSA algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 11
Implementation and Understanding of Diffie-Hellman Key Exchange Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of Diffie-Hellman Key Exchange Algorithm

Aim: To Study, Implement and Demonstrate the Diffie-Hellman Key Exchange Algorithm

Theory:

Diffie–Hellman key exchange is a mathematical method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key.

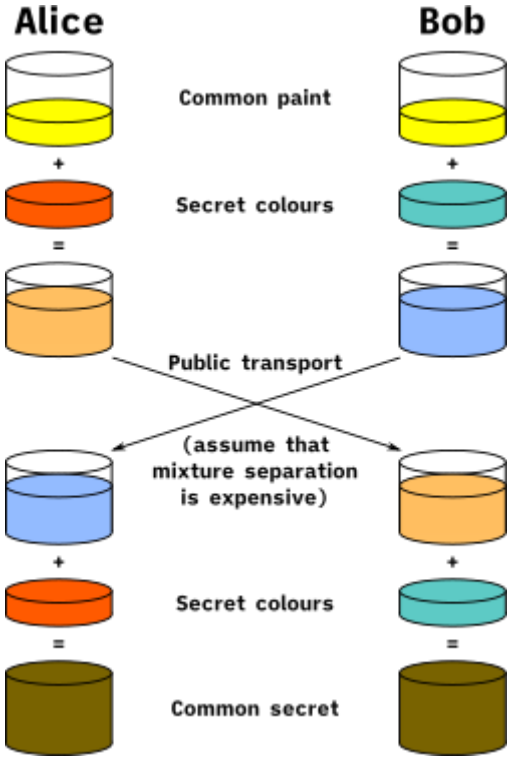
Traditionally, secure encrypted communication between two parties required that they first exchange keys by some secure physical means, such as paper key lists transported by a trusted courier. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric-key cipher.

Diffie–Hellman is used to secure a variety of Internet services. However, research published in October 2015 suggests that the parameters in use for many DH Internet applications at that time are not strong enough to prevent compromise by very well-funded attackers, such as the security services of some countries.

We look at an illustration:

Diffie–Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:

The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that does not need to be kept secret. In this example, the color is yellow. Each person also selects a secret color that they keep to themselves – in this case, red and cyan. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to their partner's final color mixture.



We look at Cryptographic Explanation:

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime, and g is a primitive root modulo p. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to p–1. Here is an example of the protocol, with non-secret values in blue, and secret values in red.

1. Alice and Bob publicly agree to use a modulus p = 23 and base g = 5 (which is a primitive root modulo 23).
2. Alice chooses a secret integer a = 4, then sends Bob A = g^a mod p
    a. A = 5^4 mod 23 = 4 (in this example both A and a have the same value 4, but this is usually not the case)
3. Bob chooses a secret integer b = 3, then sends Alice B = g^b mod p
    a. B = 5^3 mod 23 = 10
4. Alice computes s = B^a mod p
    a. s = 10^4 mod 23 = 18
5. Bob computes s = A^b mod p
    a. s = 4^3 mod 23 = 18
6. Alice and Bob now share a secret (the number 18).

Both Alice and Bob have arrived at the same values because under mod p,

$$A^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p = B^a \bmod p$$

More specifically,

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

Only a and b are kept secret. All the other values – p, g, g^a mod p, and g^b mod p – are sent in the clear. The strength of the scheme comes from the fact that g^{ab} mod p = g^{ba} mod p take extremely long times to compute by any known algorithm just from the knowledge of p, g, g^a mod p, and g^b mod p. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

Secrecy chart [edit]

The chart below depicts who knows what, again with non-secret values in blue, and secret values in **red**. Here Eve is an eavesdropper – she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.

- g = public (primitive root) base, known to Alice, Bob, and Eve. $g = 5$
- p = public (prime) modulus, known to Alice, Bob, and Eve. $p = 23$
- a = Alice's private key, known only to Alice. $a = 6$
- b = Bob's private key known only to Bob. $b = 15$
- A = Alice's public key, known to Alice, Bob, and Eve. $A = g^a \bmod p = 8$
- B = Bob's public key, known to Alice, Bob, and Eve. $B = g^b \bmod p = 19$

Alice		Bob		Eve	
Known	Unknown	Known	Unknown	Known	Unknown
$p = 23$		$p = 23$		$p = 23$	
$g = 5$		$g = 5$		$g = 5$	
$a = 6$	b	$b = 15$	a		a, b
$A = 5^a \bmod 23$		$B = 5^b \bmod 23$			
$A = 5^6 \bmod 23 = 8$		$B = 5^{15} \bmod 23 = 19$			
$B = 19$		$A = 8$		$A = 8$, $B = 19$	
$s = B^a \bmod 23$		$s = A^b \bmod 23$			
$s = 19^6 \bmod 23 = 2$		$s = 8^{15} \bmod 23 = 2$			s

Let’s now look at the program implementation.

Code:

We have implemented the program in React + Node using Socket.io

There’s a server-side Node.js server which acts as a relay for the data passing. In node, we write the script to open the portal for the socket, then broadcast the incoming data:

```
const express = require('express');
const app = express();
const cors = require("cors");
const http = require('http').createServer(app);
const PORT = 8080;
const io = require('socket.io')(http,{cors: {origin: "*"}});

app.use(cors());
app.use(require('express').json());

let p = 0;
let g = 0;

let sharedKeyOfAlice = 0
let sharedKeyOfBob = 0;


io.on("connection", (socket) => {
  console.log("Someone joined!");
```

```
  socket.on("hello", (message)=>{
    console.log(message)
  });

  socket.on("publicKeyValues", (keys)=>{
    p = keys['p'];
    g = keys['g'];
    console.log(p)
    console.log(g)
    socket.broadcast.emit("publicKeyValues", keys);
  });

  socket.on("privKeyValueAlice", (value)=>{
    sharedKeyOfAlice = value
    socket.broadcast.emit("privKeyValueAlice", sharedKeyOfAlice);
  });

  socket.on("privKeyValueBob", (value)=>{
    sharedKeyOfBob = value
    socket.broadcast.emit("privKeyValueBob", sharedKeyOfBob);
  });

});

io.listen(PORT);
```

In the React side, we use socket-io-client and build 3 pages for the 3 users – Alice, Bob and Eve.

For the project, we assume that Alice enters the Public Keys (P and G). Then both Alice and Bob would create the private keys and receive their final key. Eve, meanwhile is able to eavesdrop on the data shared by Alice and Bob.

Alice:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

function gcd(x, y) {
    if ((typeof x !== 'number') || (typeof y !== 'number'))
      return false;
    x = Math.abs(x);
```

```
    y = Math.abs(y);
    while(y) {
      var t = y;
      y = x % y;
      x = t;
    }
    return x;
}

const checkIfPrimitiveRoot = (p, g) => {
    if(g<0 || gcd(p,g)!=1 || g>=p){
        alert("g<0 || gcd(p,g)!=1 || g>=p")
        return false;
    }
    let solutionsList = [];
    for(let i=1; i<p; i++){
        let value = Math.pow(g, i);
        value %= p
        if(solutionsList.includes(value)){
            alert(solutionsList)
            alert(value)
            return false;
        }
        solutionsList.push(value);
    }
    return true;
}

export default function Alice() {
    var socket = socketClient (SERVER, {
        rejectUnauthorized: false // WARN: please do not do this in production
    });

    socket.on("connect", () => {
        console.log(socket.id); // x8WIv7-mJelg7on_ALbx
        socket.emit('hello', "Hello from Alice!")
    });

    const [arePublicKeysDeclared, setArePublicKeysDeclared] = useState(false);
    const [areKeysExchanged, setAreKeysExchanged] = useState(false);
    const [hasSharedKey, setHasSharedKey] = useState(false);
    const [hasIncomingKeyReceived, setHasIncomingKeyReceived] =
useState(false)

    const [p, setP] = useState(0);
    const [g, setG] = useState(0);
    const [privKey, setPrivKey] = useState(0);
    const [keyToExchange, setKeyToExchange] = useState(0);
```

```
    const [incomingSharedKey, setIncomingSharedKey] = useState(0);
    const [finalKey, setFinalKey] = useState(-1)

    const publicKeysSubmitHandler = (e) => {
        e.preventDefault();
        if(!checkIfPrimitiveRoot(p, g)){
            alert('Invalid Value of Primitive Root.');
            return;
        }
        socket.emit("publicKeyValues", {
            p, g
        });
        setArePublicKeysDeclared(true);
    }

    const keysExchangeSubmitHandler = (e) => {
        e.preventDefault();
        let theValueToExchange = Math.pow(g, privKey);
        theValueToExchange %= p;
        setKeyToExchange(theValueToExchange)
        socket.emit("privKeyValueAlice", theValueToExchange);
        setAreKeysExchanged(true);
    }

    socket.on("privKeyValueBob", (value)=>{
        setIncomingSharedKey(value);
        setHasSharedKey(true);
        setHasIncomingKeyReceived(true)
    });

    const computeFinalKey = () => {
        let finalKeyValue = Math.pow(incomingSharedKey, privKey);
        finalKeyValue %= p;
        setFinalKey(finalKeyValue)
    }

    if(finalKey === -1){
        if(areKeysExchanged&&hasIncomingKeyReceived){
            computeFinalKey()
        }
    }

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '1rem'}}>Alice 🙍‍♀️</h2>
        {!arePublicKeysDeclared ? <></> : <div className='container container-
fluid'>
          <h6>Public Keys:</h6>
```

```
          <h6>P: {p}</h6>
          <h6>G: {g}</h6>
          </div>}
        {!arePublicKeysDeclared ? <form className='PandG'
onSubmit={publicKeysSubmitHandler} >
            <div className="form-group">
                <label htmlFor="p">Enter Public Key P</label>
                <input onChange={(e)=>setP(parseInt(e.target.value))}
type="number" className="form-control" id="p" placeholder="P" required />
            </div>
            <div className="form-group">
                <label htmlFor="g">Enter Public Key G</label>
                <input onChange={(e)=>setG(parseInt(e.target.value))}
type="number" className="form-control" id="g" placeholder="G - Primitive Root
of P" required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form> : <></>}
        {arePublicKeysDeclared && !areKeysExchanged ? <>
        <div>
          <h6>Selection of Private Key and Exchange</h6>
          <form className='privKey' onSubmit={keysExchangeSubmitHandler} >
            <div className="form-group">
              <label htmlFor="privKey">Enter Private Key</label>
              <input onChange={(e)=>setPrivKey(parseInt(e.target.value))}
type="number" className="form-control" id="privKey" placeholder="Private Key"
required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form>
        </div>
        </> : <></>}
        {areKeysExchanged && !hasSharedKey ? <>
        <div>
          <h6>Key Shared on Public Channel - {keyToExchange}</h6>
          <h6>Waiting for Key to be shared from Bob...</h6>
        </div>
        </> : <></>}
        {areKeysExchanged&&hasIncomingKeyReceived ? <>
        <div className='container container-fluid'>
            <h6>Key Shared on Public Channel - {keyToExchange}</h6>
            <h6>Incoming Key - {incomingSharedKey}</h6>
        </div>
        <div className='container container-fluid'>
            <h6>We get final common key as: {finalKey}</h6>
        </div>
```

```
        </> : <></>}
    </div>
  )
}
```

Bob:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

export default function Bob() {

  var socket = socketClient (SERVER, {
    rejectUnauthorized: false // WARN: please do not do this in production
  });

  socket.on("connect", () => {
      console.log(socket.id); // x8WIv7-mJelg7on_ALbx
      socket.emit('hello', "Hello from Bob!")
  });

  const [arePublicKeysDeclared, setArePublicKeysDeclared] = useState(false);
  const [areKeysExchanged, setAreKeysExchanged] = useState(false);
  const [hasSharedKey, setHasSharedKey] = useState(false)
  const [hasIncomingKeyReceived, setHasIncomingKeyReceived] = useState(false)

  const [p, setP] = useState(0);
  const [g, setG] = useState(0);
  const [privKey, setPrivKey] = useState(0);
  const [keyToExchange, setKeyToExchange] = useState(0);
  const [incomingSharedKey, setIncomingSharedKey] = useState(0)
  const [finalKey, setFinalKey] = useState(-1)

  socket.on("publicKeyValues", (keys)=>{
    setP(keys['p'])
    setG(keys['g'])
    setArePublicKeysDeclared(true);
  });

  const keysExchangeSubmitHandler = (e) => {
    e.preventDefault();
    let theValueToExchange = Math.pow(g, privKey);
    console.log(theValueToExchange);
    console.log("Aaaaaaa")
    theValueToExchange %= p;
    setKeyToExchange(theValueToExchange)
```

```
    socket.emit("privKeyValueBob", theValueToExchange);
    setAreKeysExchanged(true);
  }

  socket.on("privKeyValueAlice", (value)=>{
    setIncomingSharedKey(value);
    setHasSharedKey(true);
    setHasIncomingKeyReceived(true)
  })

  const computeFinalKey = () => {
    let finalKeyValue = Math.pow(incomingSharedKey, privKey);
    finalKeyValue %= p;
    setFinalKey(finalKeyValue)
  }

  if(finalKey === -1){
    if(areKeysExchanged&&hasIncomingKeyReceived){
        computeFinalKey()
    }
  }

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '1rem'}}>Bob 🧒</h2>
        {!arePublicKeysDeclared ? <p>Waiting for declaration of Public
Keys...</p> : <div className='container container-fluid'>
          <h6>Public Keys:</h6>
          <h6>P: {p}</h6>
          <h6>G: {g}</h6>
          </div>}
        {arePublicKeysDeclared && !areKeysExchanged ? <>
        <div>
          <h6>Selection of Private Key and Exchange</h6>
          <form className='privKey' onSubmit={keysExchangeSubmitHandler} >
            <div className="form-group">
              <label htmlFor="privKey">Enter Private Key</label>
              <input onChange={(e)=>setPrivKey(parseInt(e.target.value))}
type="number" className="form-control" id="privKey" placeholder="Private Key"
required />
            </div>
            <button style={{margin: '1rem auto'}} type="submit" class="btn
btn-primary">Submit</button>
        </form>
        </div>
        </> : <></>}
        {areKeysExchanged && !hasSharedKey ? <>
        <div>
```

```
          <h6>Key Shared on Public Channel - {keyToExchange}</h6>
          <h6>Waiting for Key to be shared from Alice...</h6>
        </div>
        </> : <></>}
        {areKeysExchanged&&hasIncomingKeyReceived ? <>
        <div className='container container-fluid'>
            <h6>Key Shared on Public Channel - {keyToExchange}</h6>
            <h6>Incoming Key - {incomingSharedKey}</h6>
        </div>
        <div className='container container-fluid'>
            <h6>We get final common key as: {finalKey}</h6>
        </div>
        </> : <></>}
    </div>
  )
}
```

Eve:

```
import React, {useState, useEffect} from 'react'
import socketClient  from "socket.io-client";
const SERVER = "http://localhost:8080";

export default function Eve() {

  var socket = socketClient (SERVER, {
    rejectUnauthorized: false // WARN: please do not do this in production
  });

  const [arePublicKeysReceived, setArePublicKeysReceived] = useState(false);
  const [p, setP] = useState(0)
  const [g, setG] = useState(0)
  const [isKeySharedByAlice, setIsKeySharedByAlice] = useState(false)
  const [keySharedByAlice, setKeySharedByAlice] = useState(0)
  const [isKeySharedByBob, setIsKeySharedByBob] = useState(false)
  const [keySharedByBob, setKeySharedByBob] = useState(0)


  socket.on("connect", () => {
      console.log(socket.id); // x8WIv7-mJelg7on_ALbx
      socket.emit('hello', "Hello from Eve!")
  });

  socket.on("publicKeyValues", (keys)=>{
    setP(keys['p']);
    setG(keys['g']);
    setArePublicKeysReceived(true);
```

```
  });

  socket.on("privKeyValueAlice", (value)=>{
    setKeySharedByAlice(value);
    setIsKeySharedByAlice(true);
  });

  socket.on("privKeyValueBob", (value)=>{
    setKeySharedByBob(value);
    setIsKeySharedByBob(true);
  });

  return (
    <div className='container container-fluid'>
        <h2 style={{textAlign: 'center', marginBottom: '2rem'}}>Eve 😈</h2>
        <div className='container container-fluid'>
          <h6>Public Key P: {arePublicKeysReceived ? p : 'Waiting for
it...'}</h6>
          <h6>Public Key G: {arePublicKeysReceived ? g : 'Waiting for
it...'}</h6>
          <h6>Key Shared By Alice: {isKeySharedByAlice ? keySharedByAlice :
'Waiting for it...'}</h6>
          <h6>Key Shared By Bob: {isKeySharedByBob ? keySharedByBob : 'Waiting
for it...'}</h6>
        </div>
    </div>
  )
}
```

We now illustrate with examples:

Say we wish to demonstrate the example above.

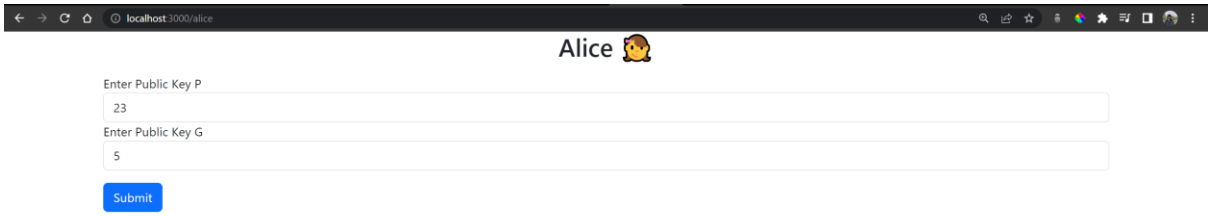
P – 23

G – 5

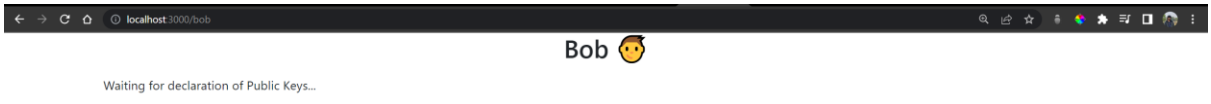
a – 6 (Private key chosen by Alice)

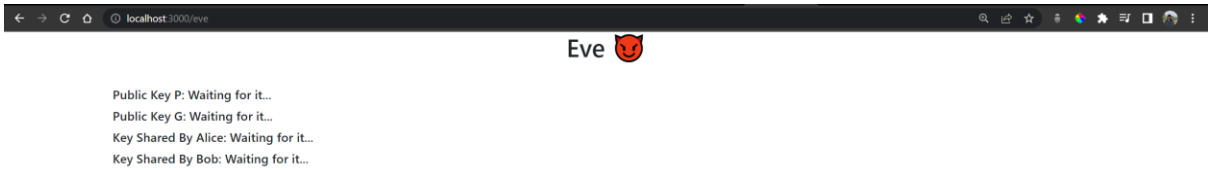
b – 15 (Private key chosen by Bob)

Now, we start with Alice:

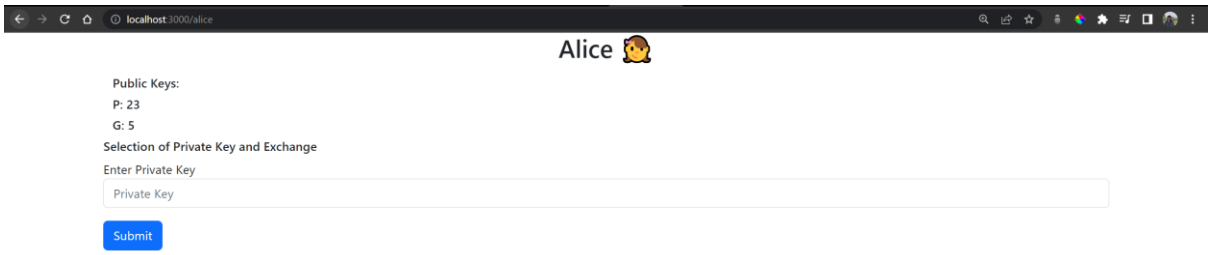


Meanwhile, Bob and Eve are idle and waiting.

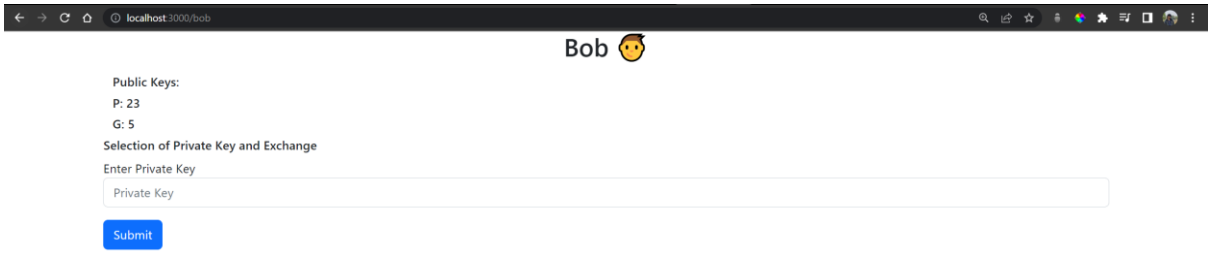




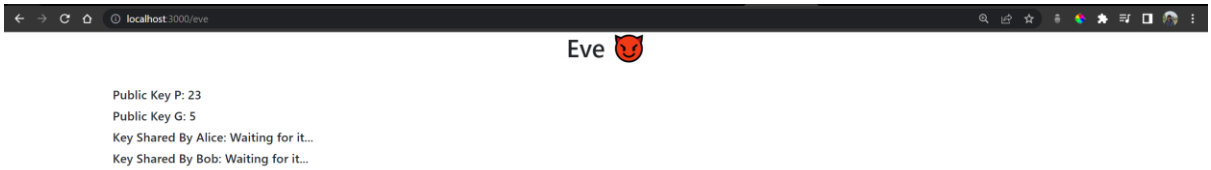
After Alice submits the key:



Same with Bob:

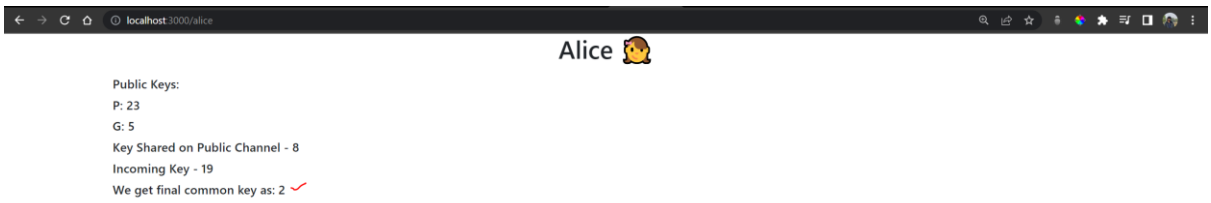


And with Eve:

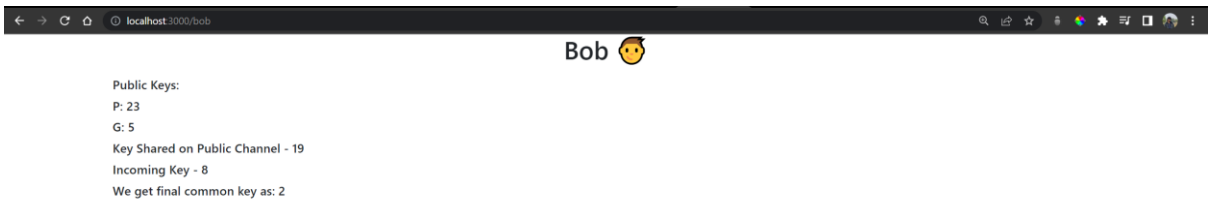


When Alice and Bob enter 6 and 15 respectively:

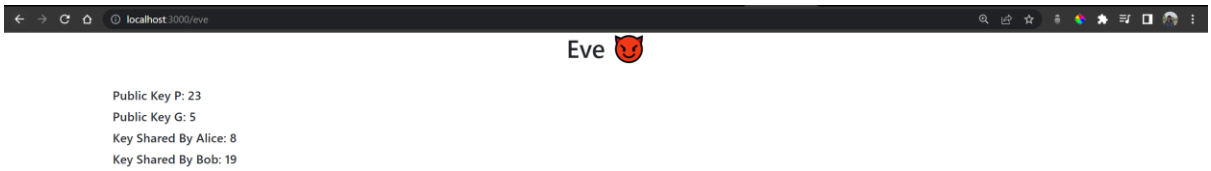
Alice:



Bob:



Eve:



Thus, we see that Alice and Bob compute the required common final key on their end, meanwhile Eve does not know the a and b to do so.

Discrete logarithm would be used to hack Diffie Hellman. Therefore, for very large numbers, this algorithm would be secure.

Conclusion:

Thus, the Diffie-Hellman Key Exchange algorithm was studied and demonstrated with the code.

Cryptography and Network Security Lab

Assignment 12
Implementation and Understanding of SHA-512 Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of SHA-512 Algorithm

Aim: To Study, Implement and Demonstrate the SHA-512 Algorithm

Theory:

SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA) and first published in 2001. They are built using the Merkle–Damgård construction, from a one-way compression function itself built using the Davies–Meyer structure from a specialized block cipher.

SHA-2 includes significant changes from its predecessor, SHA-1. The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. SHA-256 and SHA-512 are novel hash functions computed with eight 32-bit and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds. SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512 respectively, computed with different initial values. SHA-512/224 and SHA-512/256 are also truncated versions of SHA-512, but the initial values are generated using the method described in Federal Information Processing Standards (FIPS) PUB 180-4.

SHA-2 was first published by the National Institute of Standards and Technology (NIST) as a U.S. federal standard (FIPS). The SHA-2 family of

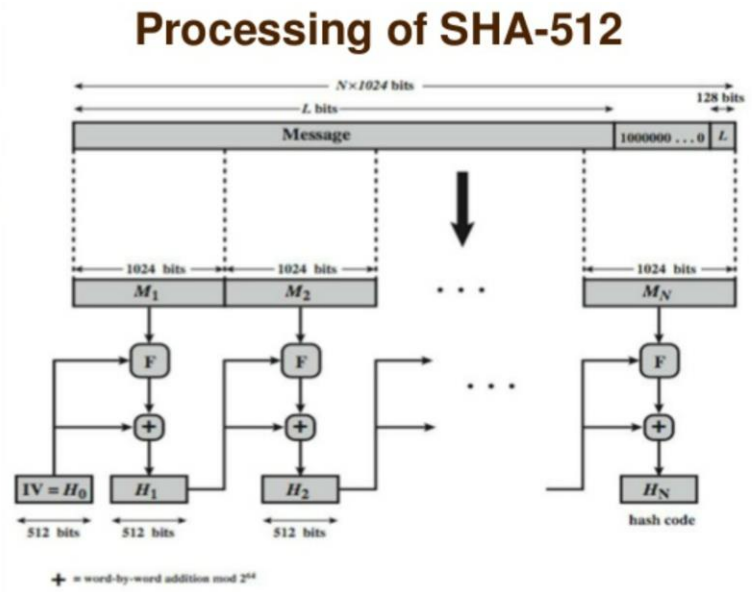
algorithms are patented in US. The United States has released the patent under a royalty-free license.

SHA-512 is a function of cryptographic algorithm SHA-2, which is an evolution of famous SHA-1.

SHA-512 is very close to Sha-256 except that it used 1024 bits "blocks", and accept as input a 2^128 bits maximum length string. SHA-512 also has others algorithmic modifications in comparison with Sha-256.

Pseudocode is as follows:

- the message is broken into 1024-bit chunks
- the initial hash values and round constants are extended to 64 bits
- there are 80 rounds instead of 64
- the message schedule array has 80 – 64-bit words instead of 64 – 32-bit words
- to extend the message schedule array w, the loop is from 16 to 79 instead of from 16 to 63
- the round constants are based on the first 80 primes 2, ..., 409
- the word size used for calculations is 64 bits long
- the appended length of the message (before pre-processing), in bits, is a 128-bit big-endian integer
- the shift and rotate amounts used are different

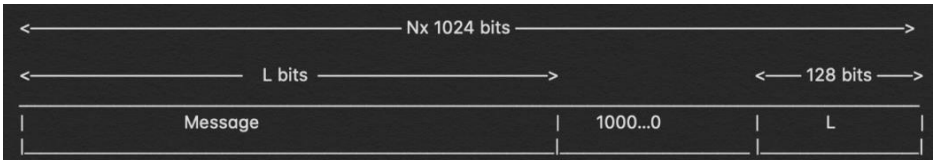


Following is the step-wise working of SHA-512:

1. Appending Bits

The first step is to carry out the padding function in which we append a certain number of bits to the plaintext message to increase its length, which should be exactly 128 bits less than an exact multiple of 1024.

When we are appending these bits at the end of the message we start with a '1' and then keep on adding '0' till the moment we reach the last bit that needs to be appended under padding and leave the 128 bits after that.



2. Append: Length bits

Now, we add the remaining 128 bits left to this entire block to make it an exact multiple of 1024, so that the whole thing can be broken down in 'n' number of 1024 blocks of message that we will apply our operation on. The way to calculate the rest of the 128 bits is by calculating the modulo with 2^{64}.

Once done, we append it to the padded bit and the original message to make the entire length of the block to be "n x 1024" in length.

3. Initialize the buffers

Now, that we have "n x 1024" length bit message that we need to hash, let us focus on the parts of the hashing function itself. To carry on the hash and the computations required, we need to have some default values initialized.

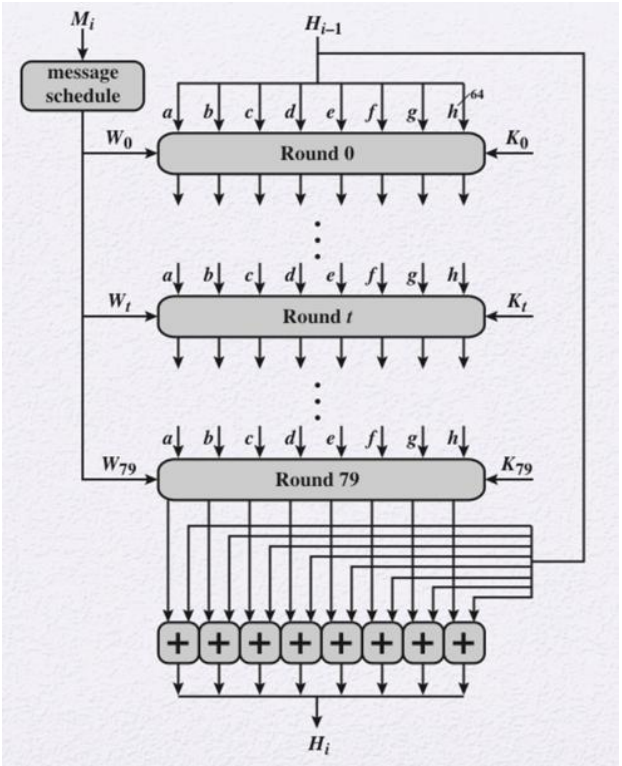
```
a = 0x6a09e667f3bcc908
b = 0xbb67ae8584caa73b
c = 0x3c6ef372fe94f82b
d = 0xa54ff53a5f1d36f1
e = 0x510e527fade682d1
f = 0x9b05688c2b3e6c1f
g = 0x1f83d9abfb41bd6b
h = 0x5be0cd19137e2179
```

These are the values of the buffer that we will need, there are other default values that we need to initialize as well. These are the value for the ‘k’ variable which we will be using.

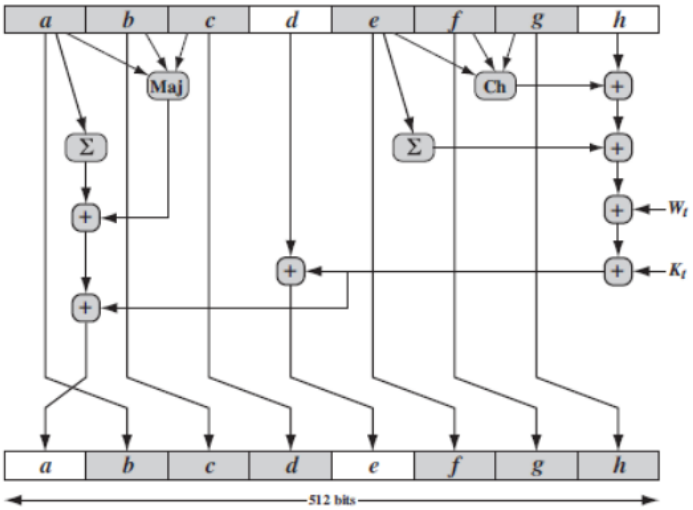
```
k[0..79] := [ 0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc, 0x3956c25bf348b538,
              0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242, 0x12835b0145706fbe,
              0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2, 0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
              0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3, 0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
              0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5, 0x983e5152ee66dfab,
              0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
              0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,
              0x53380d139d95b3df, 0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6, 0x92722c851482353b,
              0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
              0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8, 0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
              0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,
              0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc, 0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
              0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915, 0xc67178f2e372532b, 0xca273eceea26619c,
              0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba, 0x0a637dc5a2c898a6,
              0x113f9804bef90dae, 0x1b710b35131c471b, 0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
              0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817]
```

4. Compression Function

   Now, we need to have a wider look at the hash function so that we can understand what is happening. First of all, we take 1024 bits of messages and divide the complete message in ‘n’ bits.



Now that we know the methodology to obtain the values of W for 0 to 79 and already have the values for K as well for all the rounds from 0 to 79 we can proceed ahead and see where and how we do put in these values for the computation of the hash.



In the image above we can see exactly what happens in each round and now that we have the values and formulas for each of the functions carried out we can perform the entire hashing process.

This, gives us the hashed output in SHA-512.

Code:

```
import binascii
import struct

initial_hash = (
    0x6a09e667f3bcc908,
    0xbb67ae8584caa73b,
    0x3c6ef372fe94f82b,
    0xa54ff53a5f1d36f1,
    0x510e527fade682d1,
    0x9b05688c2b3e6c1f,
    0x1f83d9abfb41bd6b,
    0x5be0cd19137e2179,
)

round_constants = (
    0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f,
    0xe9b5dba58189dbbc, 0x3956c25bf348b538, 0x59f111f1b605d019,
    0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242,
    0x12835b0145706fbe, 0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
    0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
    0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
    0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65, 0x2de92c6f592b0275,
    0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
```

```
    0x983e5152ee66dfab, 0xa831c66d2db43210, 0xb00327c898fb213f,
    0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
    0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc,
    0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
    0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6,
    0x92722c851482353b, 0xa2bfe8a14cf10364, 0xa81a664bbc423001,
    0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
    0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
    0x19a4c116b8d2d0c8, 0x1e376c085141ab53, 0x2748774cdf8eeb99,
    0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb,
    0x5b9cca4f7763e373, 0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc,
    0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
    0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915,
    0xc67178f2e372532b, 0xca273eceea26619c, 0xd186b8c721c0c207,
    0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba,
    0x0a637dc5a2c898a6, 0x113f9804bef90dae, 0x1b710b35131c471b,
    0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
    0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
    0x5fcb6fab3ad6faec, 0x6c44198c4a475817,
)


def _right_rotate(n: int, bits: int) -> int:
    return (n >> bits) | (n << (64 - bits)) & 0xFFFFFFFFFFFFFFFF


def sha512(message: str) -> str:
    if type(message) is not str:
        raise TypeError('Given message should be a string.')
    message_array = bytearray(message, encoding='utf-8')

    mdi = len(message_array) % 128
    padding_len = 119 - mdi if mdi < 112 else 247 - mdi
    ending = struct.pack('!Q', len(message_array) << 3)
    message_array.append(0x80)
    message_array.extend([0] * padding_len)
    message_array.extend(bytearray(ending))

    sha512_hash = list(initial_hash)
    for chunk_start in range(0, len(message_array), 128):
        chunk = message_array[chunk_start:chunk_start + 128]

        w = [0] * 80
        w[0:16] = struct.unpack('!16Q', chunk)

        for i in range(16, 80):
            s0 = (
                _right_rotate(w[i - 15], 1) ^
                _right_rotate(w[i - 15], 8) ^
```

```
                (w[i - 15] >> 7)
            )
            s1 = (
                _right_rotate(w[i - 2], 19) ^
                _right_rotate(w[i - 2], 61) ^
                (w[i - 2] >> 6)
            )
            w[i] = (w[i - 16] + s0 + w[i - 7] + s1) & 0xFFFFFFFFFFFFFFFF

        a, b, c, d, e, f, g, h = sha512_hash

        for i in range(80):
            sum1 = (
                _right_rotate(e, 14) ^
                _right_rotate(e, 18) ^
                _right_rotate(e, 41)
            )
            ch = (e & f) ^ (~e & g)
            temp1 = h + sum1 + ch + round_constants[i] + w[i]
            sum0 = (
                _right_rotate(a, 28) ^
                _right_rotate(a, 34) ^
                _right_rotate(a, 39)
            )
            maj = (a & b) ^ (a & c) ^ (b & c)
            temp2 = sum0 + maj

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

        sha512_hash = [
            (x + y) & 0xFFFFFFFFFFFFFFFF
            for x, y in zip(sha512_hash, (a, b, c, d, e, f, g, h))
        ]

    return binascii.hexlify(
        b''.join(struct.pack('!Q', element) for element in sha512_hash),
    ).decode('utf-8')


if __name__ == "__main__":
    print("\n-- SHA512 --\n")
    print("Enter Message: ", end='')
```

```
    message = input()
    messageDigest = sha512(message)
    print("\nMessage Digest is:\n"+messageDigest)
```

We now illustrate with examples:

Say we wish to hash – ‘Password@1234’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512> py .\script.py

-- SHA512 --

Enter Message: Password@1234

Message Digest is:
1154e4485abba57f484c30a2ea543144178ff6a03da4503678a04ff8e808445709476e2e058a
f1c35afb1842eff5b7182d5d18fc3b98f185c47472520169af71
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512>
```

Message Digest is:

1154e4485abba57f484c30a2ea543144178ff6a03da4503678a04ff8e8084457094
76e2e058af1c35afb1842eff5b7182d5d18fc3b98f185c47472520169af71

We take another example:

We now wish to hash ‘Password@123’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512> py .\script.py

-- SHA512 --

Enter Message: Password@123

Message Digest is:
e8be9807c3ec0bd1ce2a58e9cddd4a89966820aedc509830b62f05a226dd29a3764e4962acf2
8b4b87377b347b9395e1dd5bc87fed1e44f17315b89d7501836c
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512>
```

Message Digest is:

e8be9807c3ec0bd1ce2a58e9cddd4a89966820aedc509830b62f05a226dd29a376
4e4962acf28b4b87377b347b9395e1dd5bc87fed1e44f17315b89d7501836c

Thus, we compute the hash. We also note how there is a significant difference in hash value even by just changing one character.

This property makes SHA-512 secure.

Thus, we demonstrated SHA-512 with examples.

Conclusion:

Thus, the SHA-512 algorithm was studied and demonstrated with the code.

The SHA-512 hashing algorithm is currently one of the best and secured hashing algorithms after hashes like MD5 and SHA-1 has been broken down. Due to their complicated nature it is not well accepted and SHA-256 is a general standard, but the industry is slowing moving towards this hashing algorithm.