НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра прикладної математики

Звіт

з лабораторної роботи № 3

із дисципліни «Криптографічні методи захисту інформації»

на тему

*Стандарт шифрування DES*

|  |  |
| --- | --- |
| Виконав: | Керівник: |
| студент групи КМ-01 | *ст. викладач Бай Ю. П.* |
| *Боженко А. О.* |  |

Київ — 2022

ЗМІСТ

[Постановка задачі 2](#_Toc101104223)

[Основні теоретичні відомості зі стандарту шифрування DES 3](#_Toc101104224)

[Контрольний приклад 5](#_Toc101104225)

[Шифрування тексту 10](#_Toc101104226)

[Розшифрування тексту 10](#_Toc101104227)

[Список літератури 11](#_Toc101104228)

[Додаток 1 12](#_Toc101104229)

[Додаток 2 17](#_Toc101104230)

[Відповіді на контрольні запитання 27](#_Toc101104231)

***Мета роботи:*** опрацювати загальну послідовність алгоритму DES, розробити криптосистему на основі стандарту шифрування DES, дослідити особливості алгоритму: нелінійні блоки, лавиновий ефект, режими роботи.

# Постановка задачі

1. Скласти програму для шифрування та розшифрування за алгоритмом DES 64-бітного блоку інформації, використовуючи 64-бітний ключ. Продемонструвати роботу програми на контрольному прикладі. Навести скріншоти детального покрокового виконання алгоритму.

Контрольний приклад:

<https://courses.cs.washington.edu/courses/cse467/99au/admin/Slides/Week6Lecture1/sld001.htm>. Виконати приклад в деталях. Зробити скріншоти до результатів слайдів 7, 9, 10, 11, 12, 13, 14, 15, 17, 19, 20, 21, 22, 23.

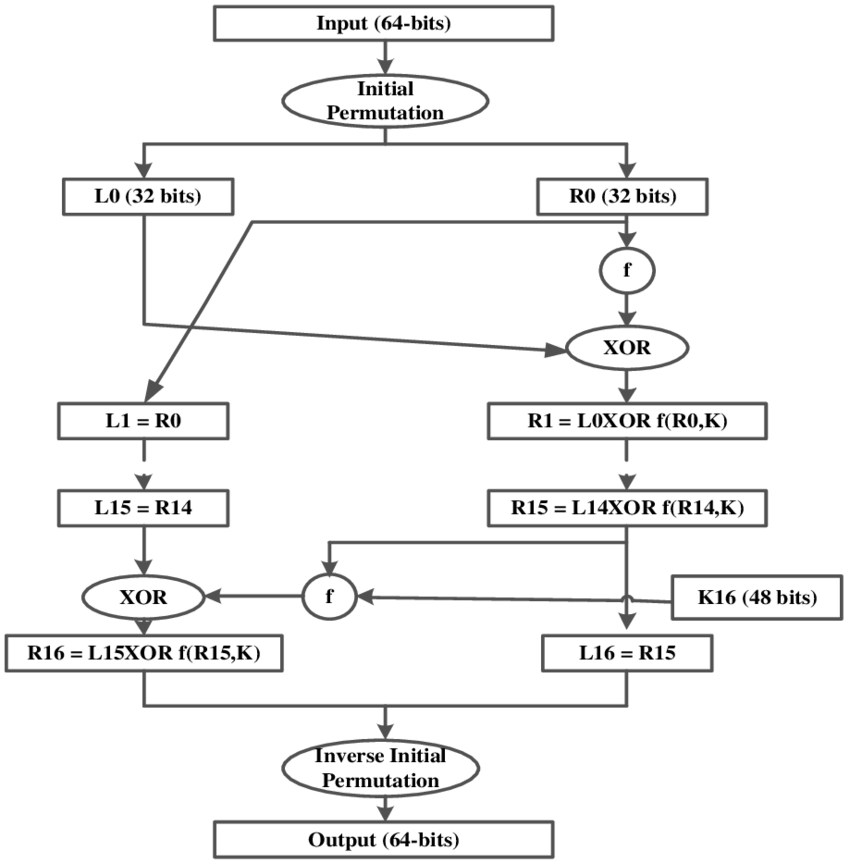
2. Розширити функціональність програми для випадку відкритого тексту довільної довжини та 64-бітного ключа.

3. За стандартом DES зашифрувати текст довжиною від 8 до 80 символів, користуючись 64-бітним ключем.

4. Дано зашифроване за стандартом DES повідомлення довжиною від 8 до 80 символів та відомий 64-бітний ключ. Розшифрувати задане повідомлення.

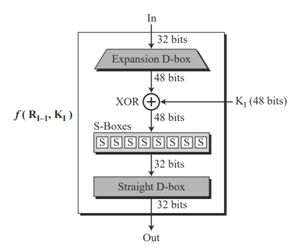
# Основні теоретичні відомості зі стандарту шифрування DES

**DES** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) Data Encryption Standard) — це симетричний алгоритм шифрування даних. DES є блочним [шифром](https://uk.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80) - дані шифруються [блоками](https://uk.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA) по 64 біти - 64 бітний блок явного тексту подається на вхід алгоритму, а 64-бітний блок шифрограми отримується в результаті роботи алгоритму. Принцип роботи алгоритму – зображений на мал.1

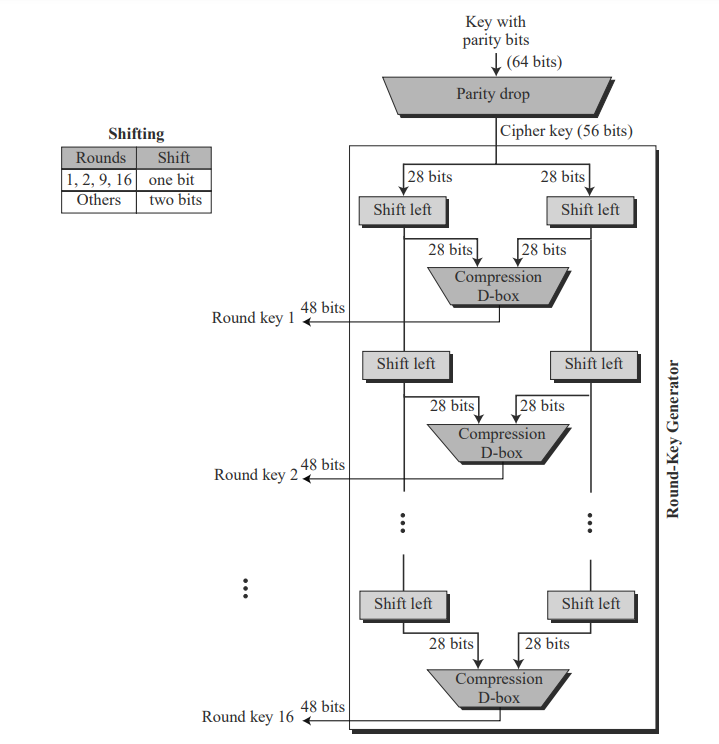


мал. 1.1 - блок-схема алгоритму DES шифрування

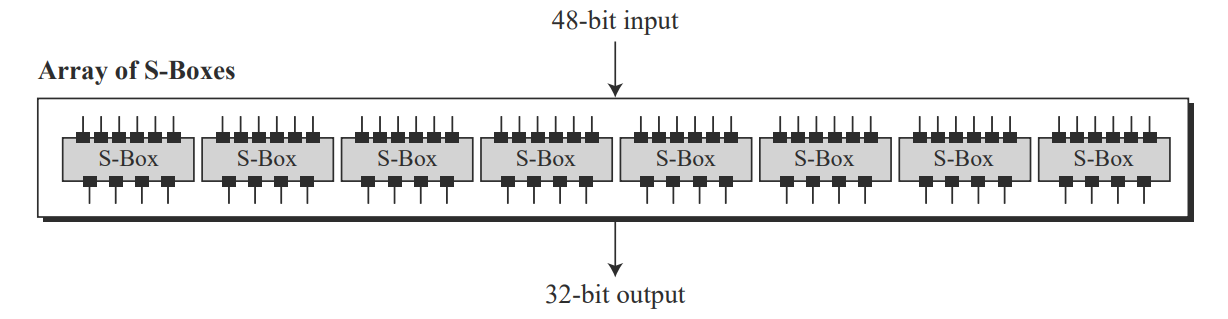
Ключі генеруються з даного 64 бітного ключа, 8 парних бітів якого відкидаються. Принцип генерації 16 ключів зображено на мал. 1.2



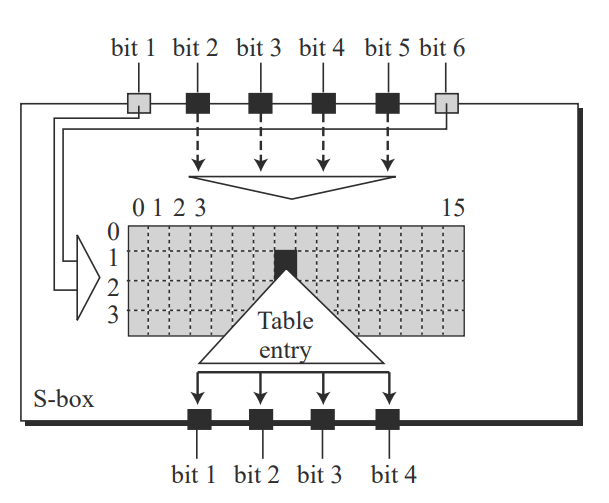
мал. 1.6 - Опис функції f (Ri-1, ki)



мал. 1.2 – алгоритм генерації 16ти ключів для DES-шифрування



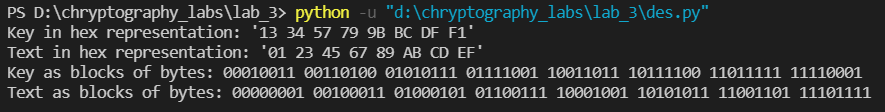
мал. 1.3 – прохождення через S-boxes

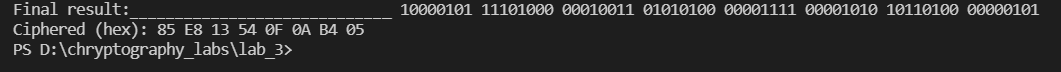


мал. 1.4 – принцип одиничного проходження через S-box, щоб отримати з 6 бітів 4

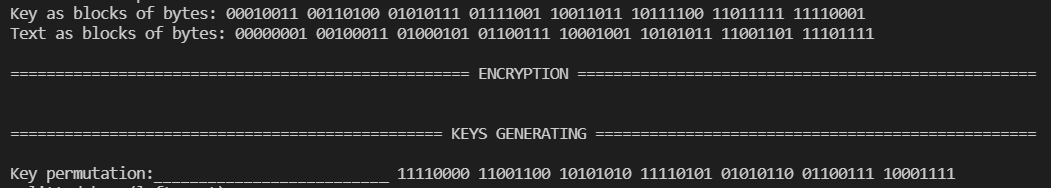
# Контрольний приклад

<https://courses.cs.washington.edu/courses/cse467/99au/admin/Slides/Week6Lecture1/sld001.htm>

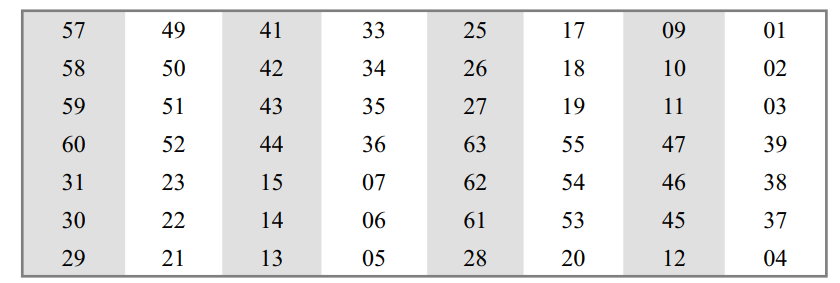




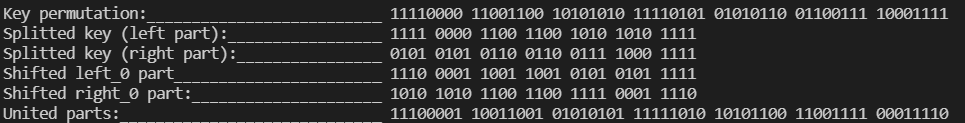
мал. 2.1 – скриншот початкових даних та результату (зашифрованих)



Мал. 2.2 – скриншот даного ключа (в байтах) та зменшеного (56 бітів), перестановка (Parity drop – мал. 2.3)



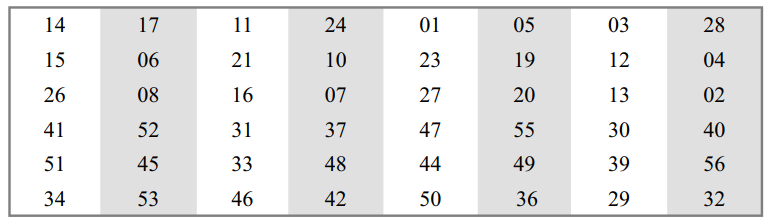
мал. 2.3 – таблиця Parity drop (генерація ключів), відкидання 8 бітів 64 бітного ключа, щоб отримати 56 бітів для генерації 16 ключів



мал. 2.4 – скриншот до зсуву лівої та правої половин ключа (перший ключ)



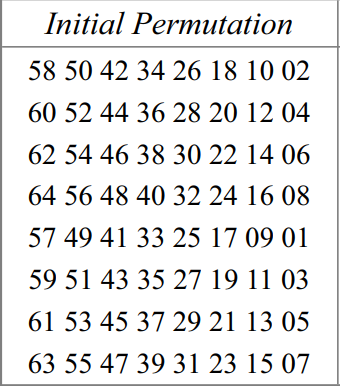
мал. 2.5 – скриншот до об’єднання лівої та правої половин та пройшовшого престановку (Compression D-box – мал 2.6) остаточно сформованого першого ключа



мал. 2.6 – таблиця зтиснення (Compression D-box) для утворення і-го ключа після зсуву та з’єднання лівої та правої частини(генерація ключа)



Мал. 2.7 – скриншот до першого блоку, що пройнов початкову перестановку (Initial permutation – мал. 2.8)



мал. 2.8 - таблиця Initial permutation

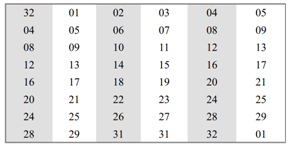


мал. 2.9 – скриншот до поділу першого блоку на дві половини

Здійснимо перестановку-збільшення правої половини:

Права (до перстановки):11110000 10101010 11110000 10101010

Права (після перестановки): 00010101 01010101 01111010 00010101 01010101



мал. 2.10 – таблиця перстановки-збільшення (Expansion box)

Після операції XOR ключа з правою половиною отримуємо:

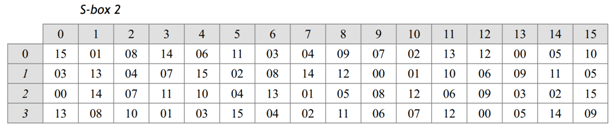
Ключ: 00011011 00000010 11101111 11111100 01110000 01110010

Права: 01111010 00010101 01010101 01111010 00010101 01010101

XOR: 01100001 00010111 10111010 10000110 01100101 00100111

зXORений масив, розбитий на 6-бітні блоки:

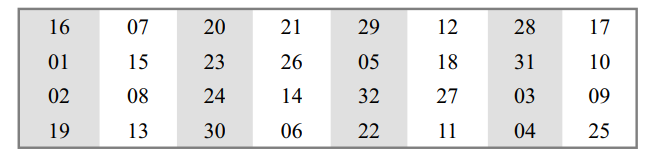
011000 010001 011110 111010 100001 100110 010100 100111



мал. 2.11 – другий S-box

Проведемо кожні 6 біт через S-boxes, для прикладу візьмімо другий 6 бітний блок: маємо рядок 01 – себто другий, стовпчик 1000 – себто дев’ятий, на перетині маємо число 12, або 1100.

Отриманий масив бітів: 0010 0111 0001 0000 1110 0001 0110 1111



мал. 2.12 – таблиця перстановки (Straight D-box) після застосування S-boxes

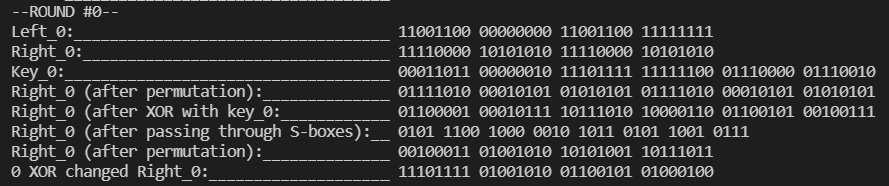
Переставимо(Straight D-box ): 00100011 01001010 10101001 10111011

Здійснимо операцію XOR лівої половини з зміненою правою:

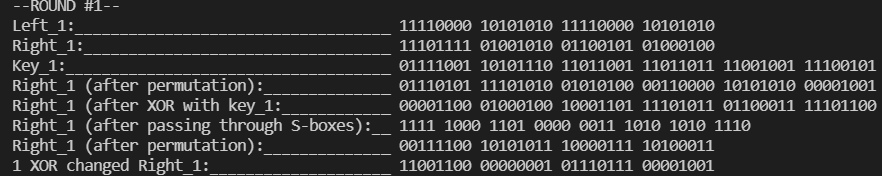
Ліва: 11001100 00000000 11001100 11111111

Права: 00100011 01001010 10101001 10111011

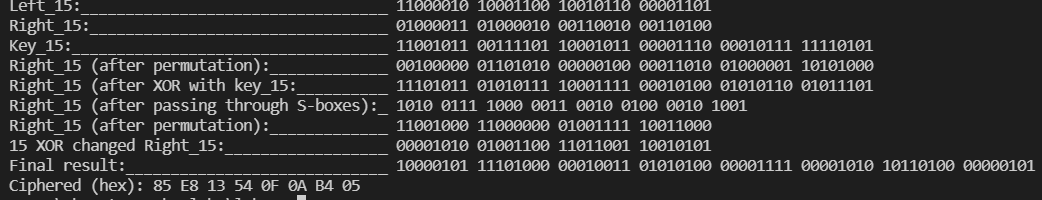
XOR: 11101111 01001010 01100101 01000100



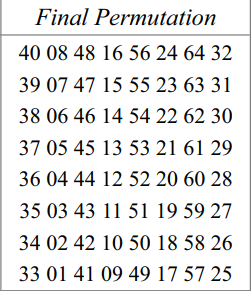
мал. 2.13 – скриншот до першого раунду



мал. 2.14 – скриншот до другого раунду



мал. 2.15 – скриншот до фінальної перестановки (Inverse Initial permutation) – мал. 2.16) до зXORеної правої половини та лівої половини.



мал. 2.16 – таблиця Inverse Initial permutation

# Шифрування тексту

Зашифруємо текст

"Morning coffee" ,

використовуючи ключ *key* = '13 34 57 79 9B BC DF F1' (hex).

Одержуємо зашифрований текст:

26 42 F8 47 1F 8D B6 6A 85 48 BD D9 F4 9F 6E 71 (hex)

# Розшифрування тексту

Дано зашифроване за стандартом DES повідомлення:

EC 0D 8C 83 00 78 14 63 25 CE B5 27 9F E4 73 69 (hex)

і відомо 64-бітний ключ *key* = '13 34 57 79 9B BC DF F1' (hex).

Розшифровуючи заданий криптотекст, одержимо:

47 6C 6F 72 79 20 74 6F 20 55 6B 72 61 69 6E 65 (hex)

Або в символах ascii: 'Glory to Ukraine'

# Список літератури

1. Тарнавський Ю.А. Технології захисту інформації [Електронний ресурс] / Ю. А. Тарнавський. – Київ: КПІ ім. Ігоря Сікорського, 2018. – 162 с.
2. Шнайер Б. Прикладная криптография: Протоколы, алгоритмы, исходные тексты на языке Си / Б. Шнайер. – М.: Диалектика, 2003. – 610 с.
3. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В. Основы криптографии. – М.: Гелиос АРВ, 2001. – 480 с.
4. Столлингс В. Криптография и защита сетей: принципы и практика, 2-е изд.: Пер. с англ. – М.: «Вильямс», 2001. – 672 с.
5. Гулак Г.М., Мухачов В.А., Хорошко В.О., Яремчук Ю.Є. Основи криптографічного захисту інформації. – Вінниця: ВНТУ, 2011. – 198 с.
6. <https://academic.csuohio.edu/yuc/security/Chapter_06_Data_Encription_Standard.pdf>

# Додаток 1

Текст програми, що реалізує стандарт шифрування DES

#-\*- coding: utf8 -\*-

ENCRYPT = 0

DECRYPT = 1

def string\_to\_bit\_array(text):#Convert a string into a list of bits

    array = list()

    for char in text:

        binval = binvalue(char, 8)#Get the char value on one byte

        array.extend([int(x) for x in list(binval)]) #Add the bits to the final list

    return array

def bit\_array\_to\_string(array): #Recreate the string from the bit array

    res = ''.join([chr(int(y,2)) for y in [''.join([str(x) for x in \_bytes]) for \_bytes in  nsplit(array,8)]])

    return res

def binvalue(val, bitsize):

    binval = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]

    if len(binval) > bitsize:

        raise "Binary value larger than the expected size."

    while len(binval) < bitsize:

        binval = "0" + binval

    return binval

def nsplit(s, n):

    """Split a list into sublists of size 'n'"""

    return [s[k:k+n] for k in range(0, len(s), n)]

class DES():

        #Initial permut matrix for the datas

    \_\_PI = [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]

    #Initial permut made on the key

    \_\_CP\_1 = [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]

    #Permut applied on shifted key to get Ki+1

    \_\_CP\_2 = [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]

    #Expand matrix to get a 48bits matrix of datas to apply the xor with Ki

    \_\_E = [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]

    #SBOX

    \_\_S\_BOX = [

    [[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],

    ]

    ]

    #Permut made after each SBox substitution for each round

    \_\_P = [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]

    #Final permut for datas after the 16 rounds

    \_\_PI\_1 = [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]

    #Matrix that determine the shift for each round of keys

    \_\_SHIFT = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

    def \_\_init\_\_(self):

        self.password = None

        self.text = None

        self.keys = list()

    def run(self, key, text, action, padding):

        if len(key) < 8:

            raise "Key should be 8 bytes long."

        elif len(key) > 8:

            key = key[:8]

        self.password = key

        self.text = text

        if action == ENCRYPT:

            # show log

            print(f"\n{' ENCRYPTION '.center(114, '=')}\n")

            if padding:

                self.addPadding()

            elif len(self.text) % 8 != 0:

                raise "Text length must be divisible by 8 if padding not supplied."

        self.generatekeys()

        if action == DECRYPT:

            # show log

            print(f"\n{' DECRYPTION '.center(114, '=')}\n")

            self.keys.reverse()

        text\_blocks = nsplit(self.text, 8)

        result = []

        for block in text\_blocks:

            block = string\_to\_bit\_array(block)

            # show log

            print("#"\*114)

            show\_log("Block:", show\_bit\_arr(block, 8), 42)

            block = self.permute(block, DES.\_\_PI)

            left, right = nsplit(block, 32)

            tmp = None

            for i in range(16):

                # show log

                print(f"--ROUND #{i}--")

                show\_log(f"Left\_{i}:", show\_bit\_arr(left, 8), 42)

                show\_log(f"Right\_{i}:", show\_bit\_arr(right, 8), 42)

                right\_e = self.permute(right, DES.\_\_E)

                # show log

                show\_log(f"Key\_{i}:", show\_bit\_arr(self.keys[i], 8), 42)

                show\_log(f"Right\_{i} (after permutation):", show\_bit\_arr(right\_e, 8), 42)

                tmp = self.xor(self.keys[i], right\_e)

                # show log

                show\_log(f"Right\_{i} (after XOR with key\_{i}:", show\_bit\_arr(tmp, 8), 42)

                tmp = self.substitute(tmp)

                # show log

                show\_log(f"Right\_{i} (after passing through S-boxes):", show\_bit\_arr(tmp, 4), 42)

                tmp = self.permute(tmp, DES.\_\_P)

                # show log

                show\_log(f"Right\_{i} (after permutation):", show\_bit\_arr(tmp, 8), 42)

                tmp = self.xor(left, tmp)

                # show log

                show\_log(f"{i} XOR changed Right\_{i}:", show\_bit\_arr(tmp, 8), 42)

                left = right

                right = tmp

            result += self.permute(right + left, DES.\_\_PI\_1)

            # show log

            show\_log(f"Final result:", show\_bit\_arr(result[-64:], 8), 42)

        final\_res = bit\_array\_to\_string(result)

        if action == DECRYPT and padding:

            return self.removePadding(final\_res)

        else:

            return final\_res

    def substitute(self, d\_e):

        """Pass block of 48 bits through operation with S-boxes to gain block of 32 bits"""

        subblocks = nsplit(d\_e, 6)

        result = []

        for idx, block in enumerate(subblocks):

            row = int(str(block[0]) + str(block[-1]), 2)

            column = int(''.join(str(x) for x in block[1:][:-1]), 2)

            val = DES.\_\_S\_BOX[idx][row][column]

            result += [int(x) for x in binvalue(val, 4)]

        return result

    def permute(self, block, table):

        """ Transform block (represented as list of bits) using bit position table """

        return [block[x - 1] for x in table]

    def xor(self, t1, t2):

        return [x^y for x, y in zip(t1, t2)]

    def generatekeys(self):

        """ Generate all 16 bit shifted versions of key at once"""

        self.keys = []

        print(f"\n{' KEYS GENERATING '.center(114, '=')}\n")

        key = string\_to\_bit\_array(self.password)

        key = self.permute(key, DES.\_\_CP\_1)

        show\_log("Key permutation:", show\_bit\_arr(key, 8), 42)

        left, right = nsplit(key, 28)

        show\_log("Splitted key (left part):", show\_bit\_arr(left, 4), 42)

        show\_log("Splitted key (right part):", show\_bit\_arr(left, 4), 42)

        for i in range(16):

            left, right = self.shift(left, right, DES.\_\_SHIFT[i])

            show\_log(f"Shifted left\_{i} part", show\_bit\_arr(left, 4), 42)

            show\_log(f"Shifted right\_{i} part:", show\_bit\_arr(right, 4), 42)

            tmp = left + right

            show\_log("United parts:", show\_bit\_arr(tmp, 8), 42)

            self.keys.append(self.permute(tmp, DES.\_\_CP\_2))

            show\_log(f"Permuted key\_{i}", show\_bit\_arr(self.keys[-1], 8), 42)

        print("#"\*114)

    def shift(self, left, right, n):

        """ Shift a list of the given value cyclically to the LEFT"""

        return left[n:] + left[:n], right[n:] + right[:n]

    def addPadding(self):

        """Add padding to the data using PKCS5 specification"""

        pad\_len = 8 - (len(self.text) % 8)

        self.text += pad\_len \* chr(pad\_len)

    def removePadding(self, data):

        """Remove the padding of the plain text (it is assumed that there is padding)"""

        pad\_len = ord(data[-1])

        return data[:-pad\_len]

    def encrypt(self, key, text, padding):

        return self.run(key, text, ENCRYPT, padding)

    def decrypt(self, key, text, padding):

        return self.run(key, text, DECRYPT, padding)

def hex\_string\_to\_text(hex\_string):

    return [chr(int(x, 16)) for x in hex\_string.split()]

def text\_to\_hex\_string(text):

    return ' '.join('{0:0>2}'.format(hex(ord(x))[2:].upper()) for x in text)

def show\_byte\_blocks(text):

    """text - string, function returns string in byte-blocks representation,

    where every byte corresponds to ascii number of char"""

    s = []

    for i in text:

        a = bin(ord(i))[2:]

        if len(a) > 8:

            raise "Binary value larger than the expected size."

        s.append((8 - len(a)) \* '0' + a)

    return ' '.join(s)

def show\_bit\_arr(arr, n):

    """arr - array of bits, function returns string representation of bits grouped in n-blocks"""

    return ' '.join(''.join(str(c) for c in i) for i in nsplit(arr, n))

def show\_log(first\_p, last\_p, n):

    """function prints log that contains first\_p aligned to left on n, and then prints last\_p"""

    print(f"{first\_p:\_<{n}} {last\_p}")

if \_\_name\_\_ == '\_\_main\_\_':

    t = "Morning coffee"

    # t = "EC 0D 8C 83 00 78 14 63 25 CE B5 27 9F E4 73 69"

    key\_hex =  '13 34 57 79 9B BC DF F1'

    # t = hex\_string\_to\_text(t)

    k = hex\_string\_to\_text(key\_hex)

    need\_to\_pad = not len(t) % 8 == 0

    print(f"Key as blocks of bytes: {show\_byte\_blocks(k)}")

    print(f"Text as blocks of bytes: {show\_byte\_blocks(t)}")

    d = DES()

    r = d.encrypt(k, t, need\_to\_pad)

    # r2 = d.decrypt(k, t, False)

    show\_log("Ciphered:", f"{r!r}", 42)

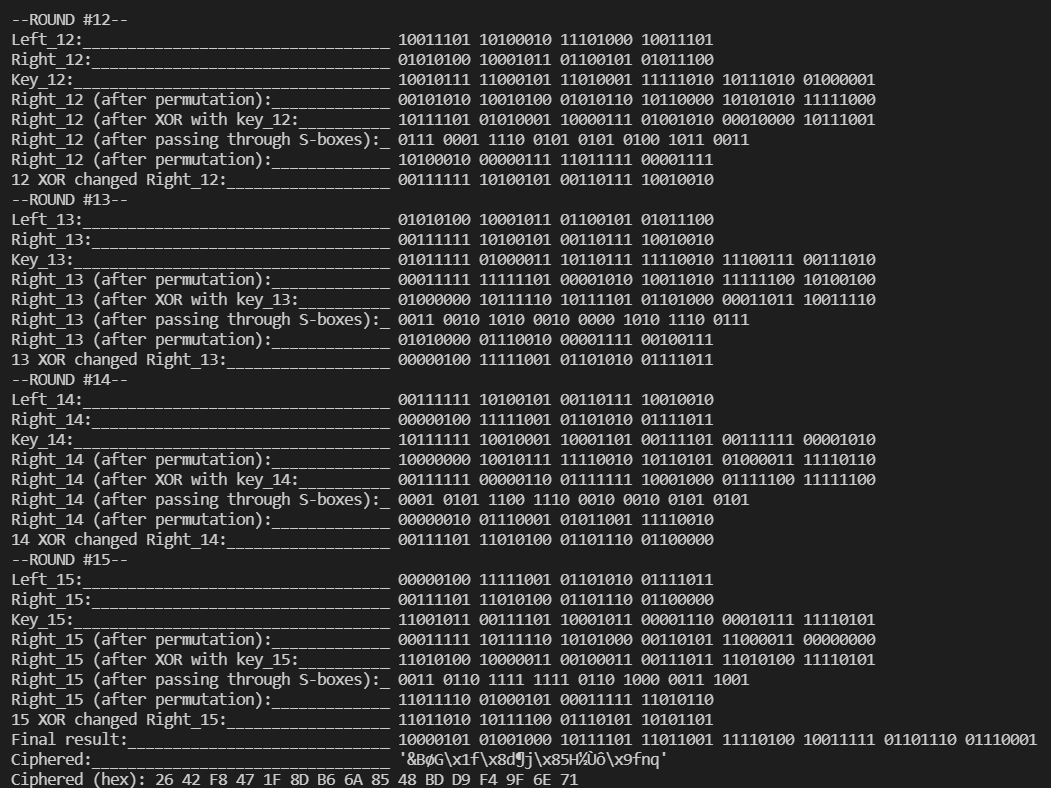
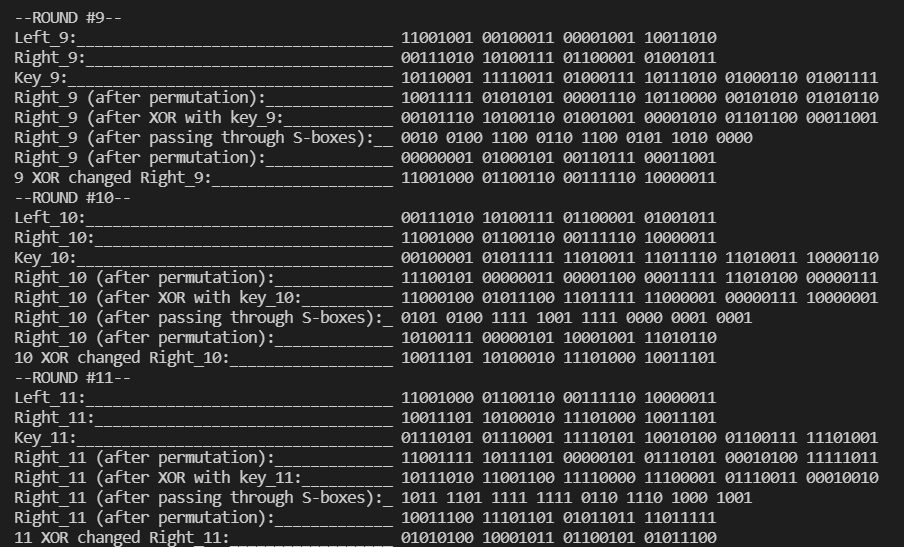
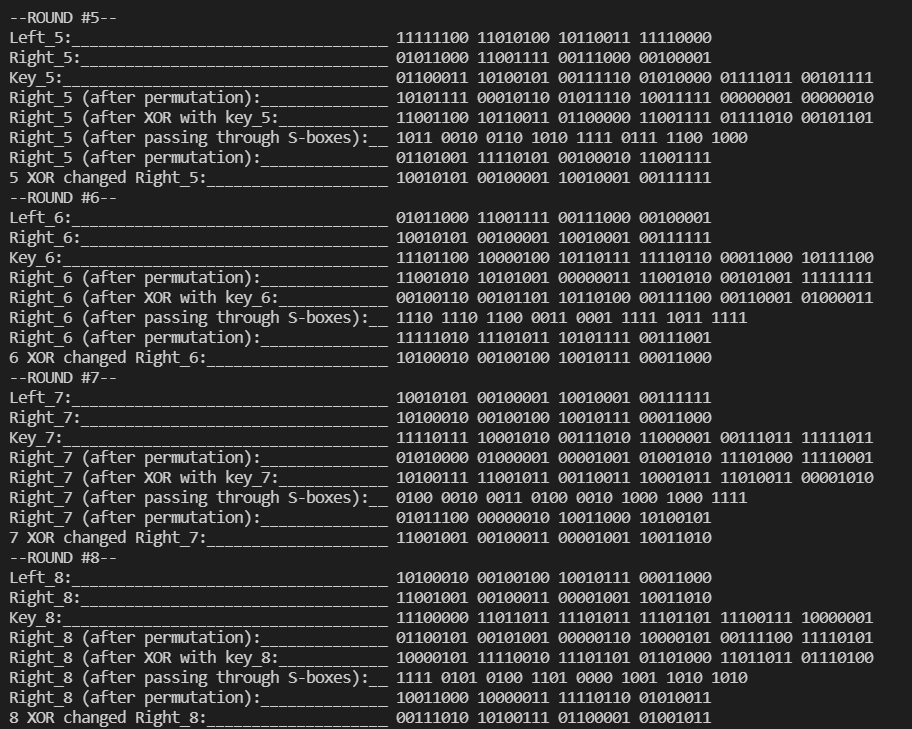
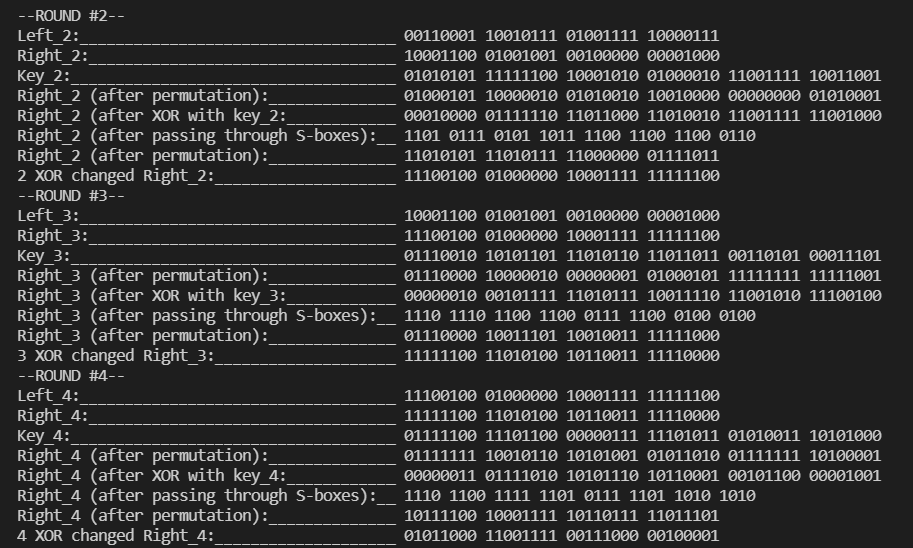
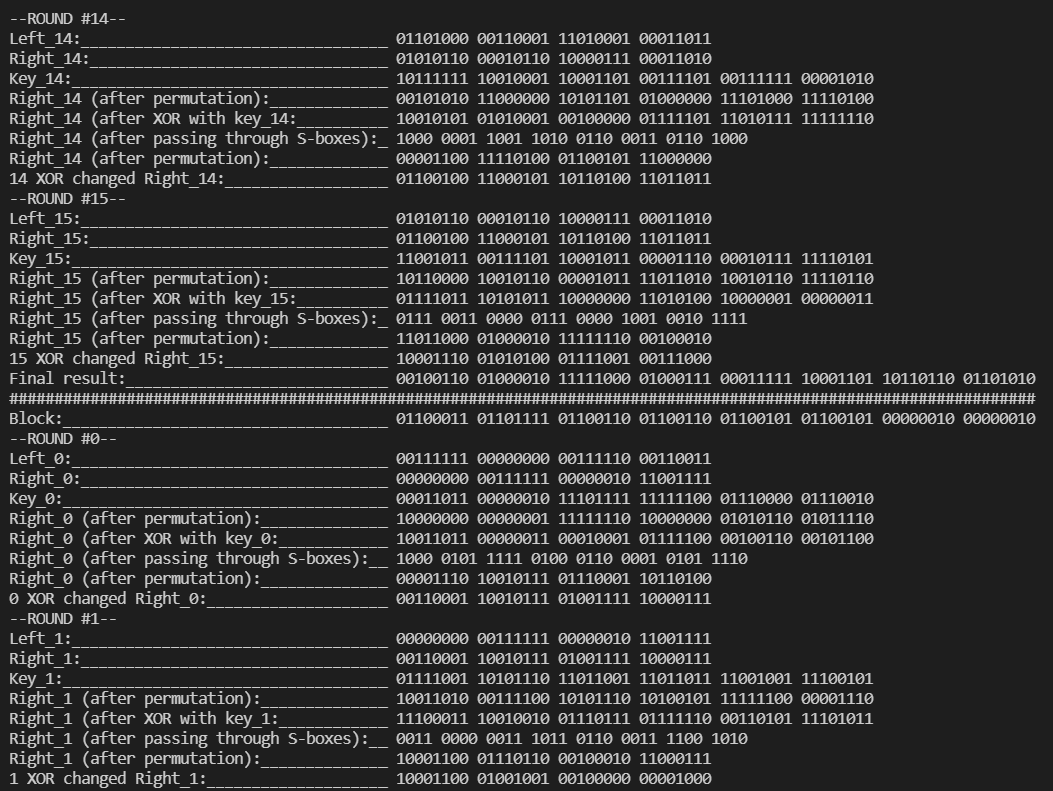
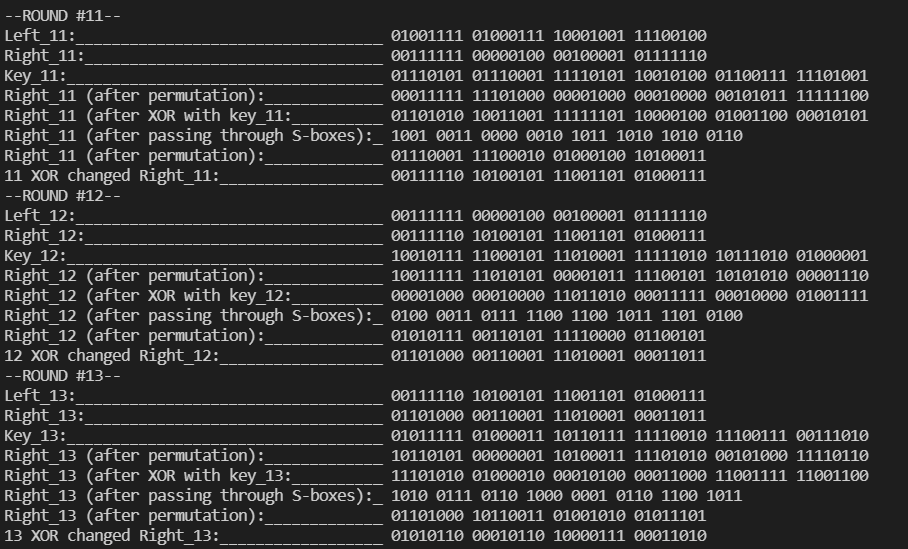
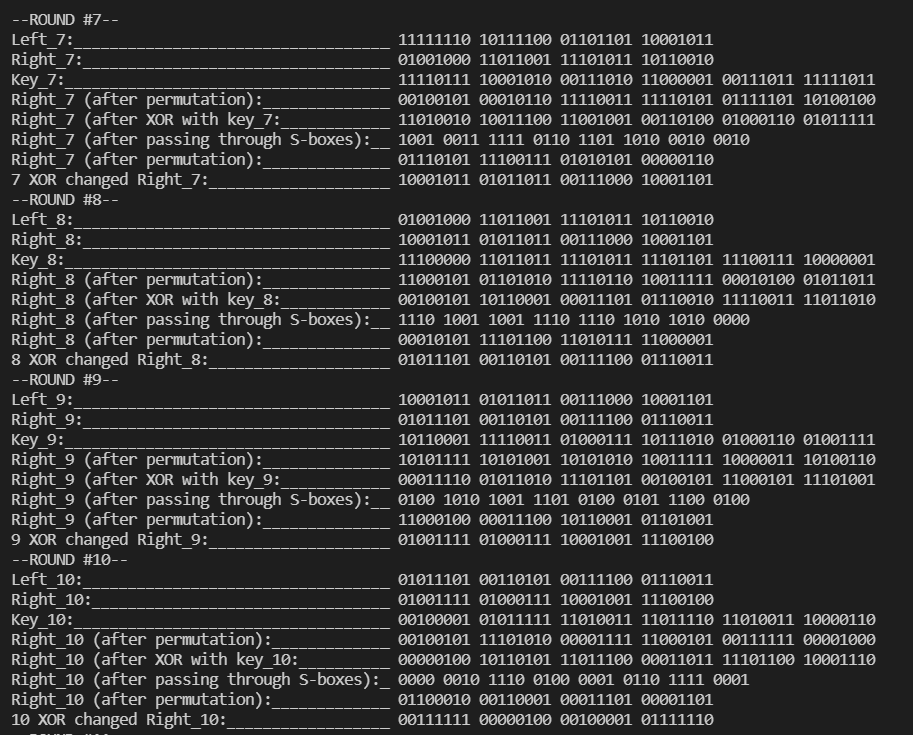
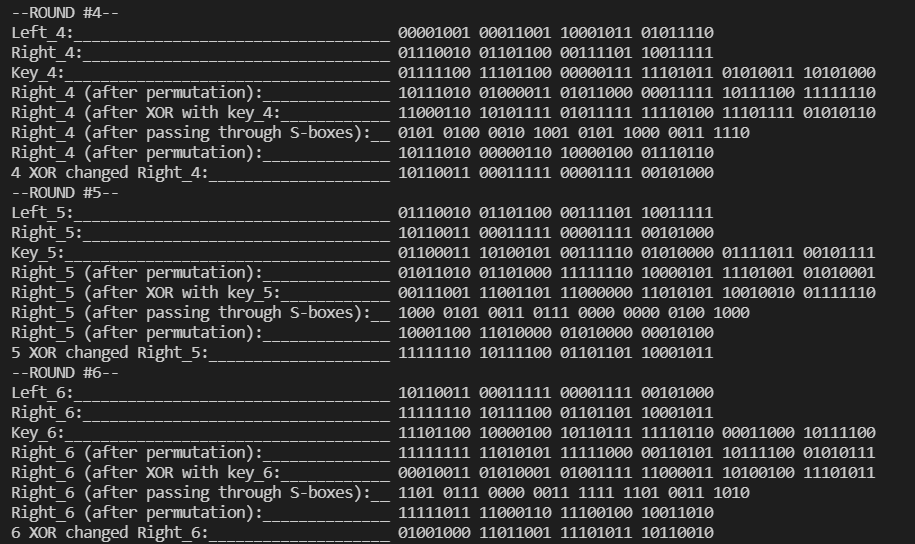
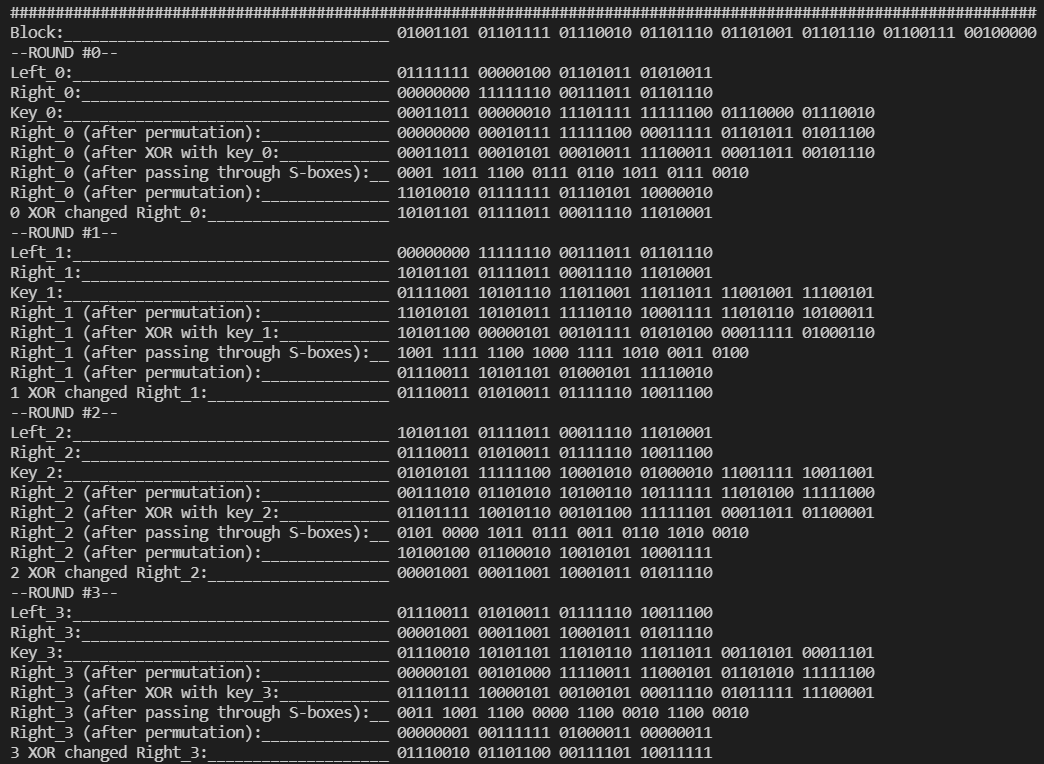
    # show\_log("Deciphered:", f"{r2!r}", 42)

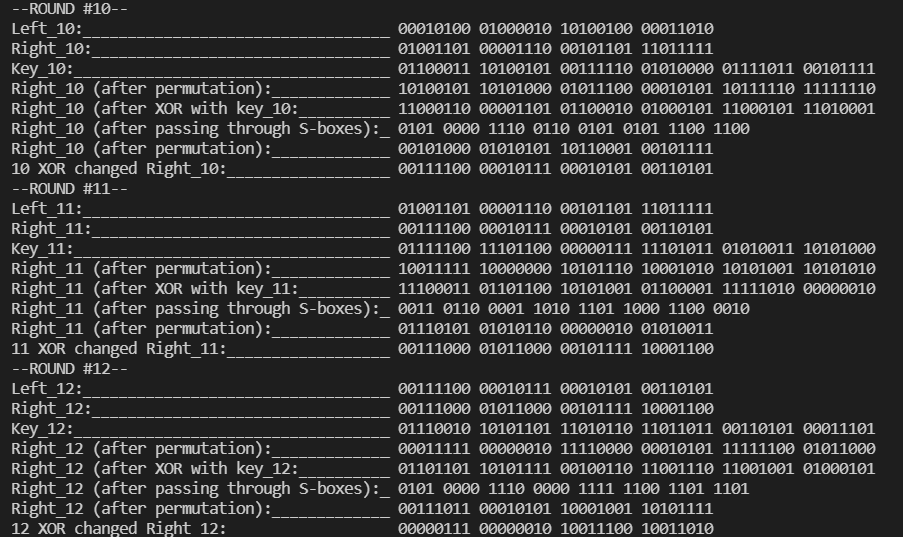
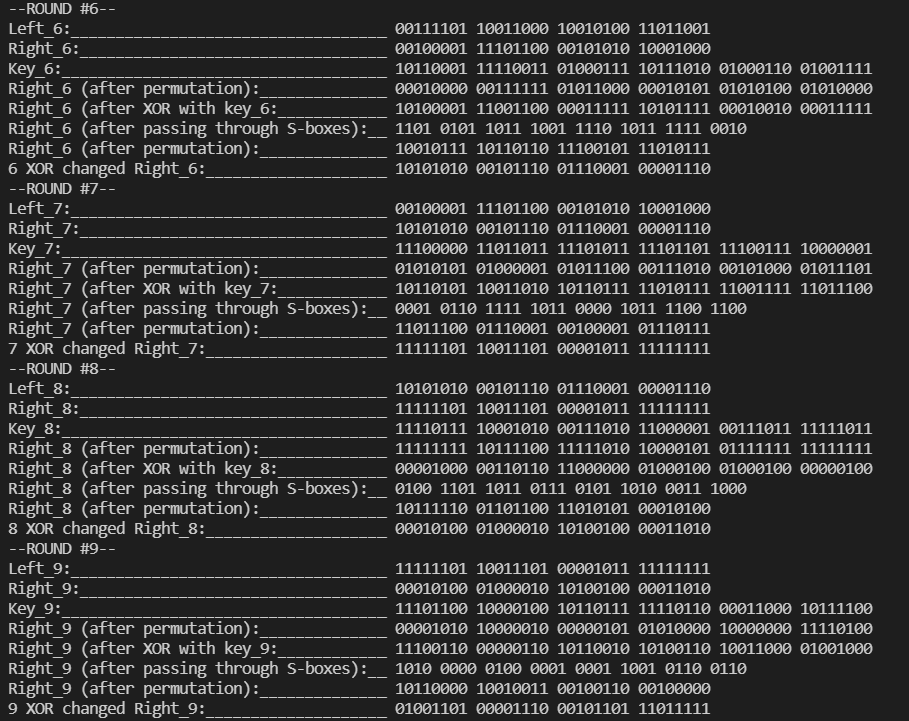
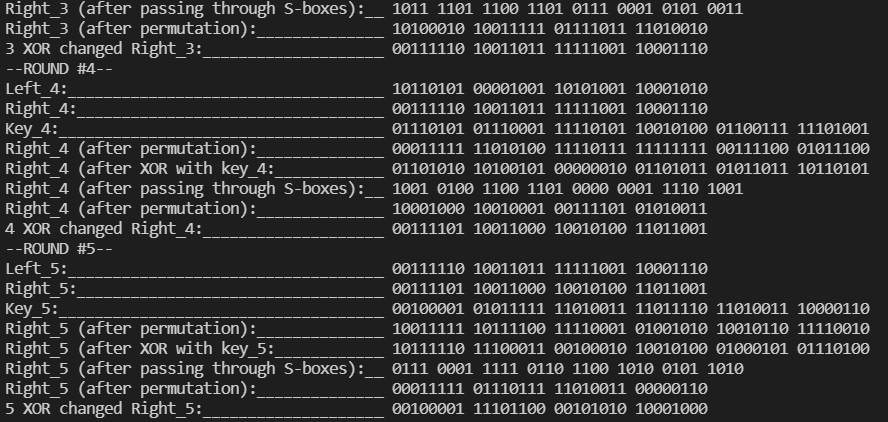
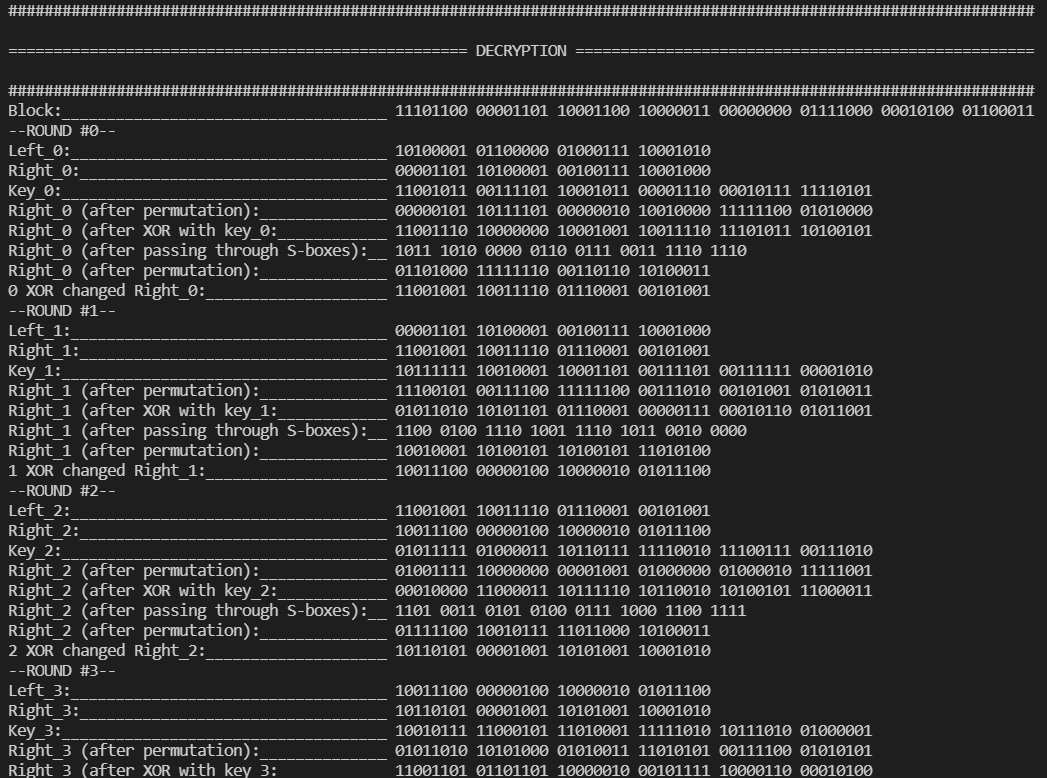
    print("Ciphered (hex): {0}".format(text\_to\_hex\_string(r)))

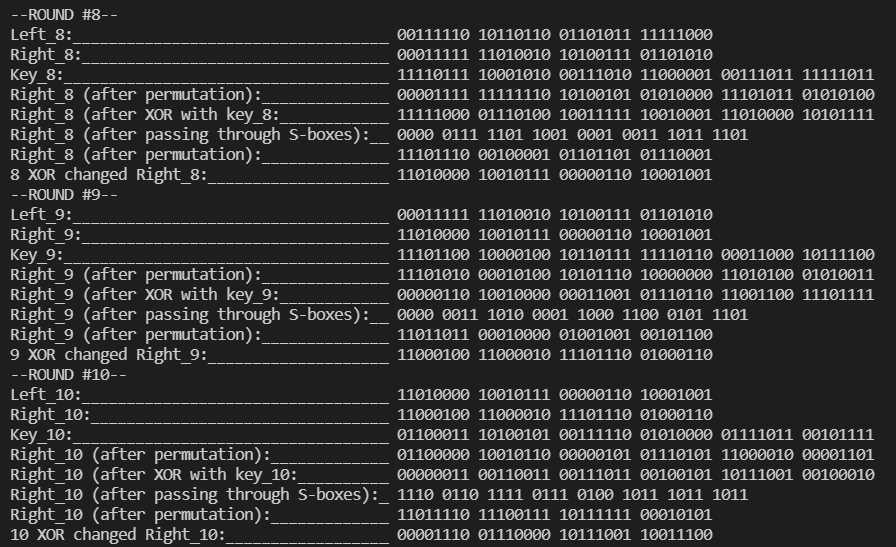
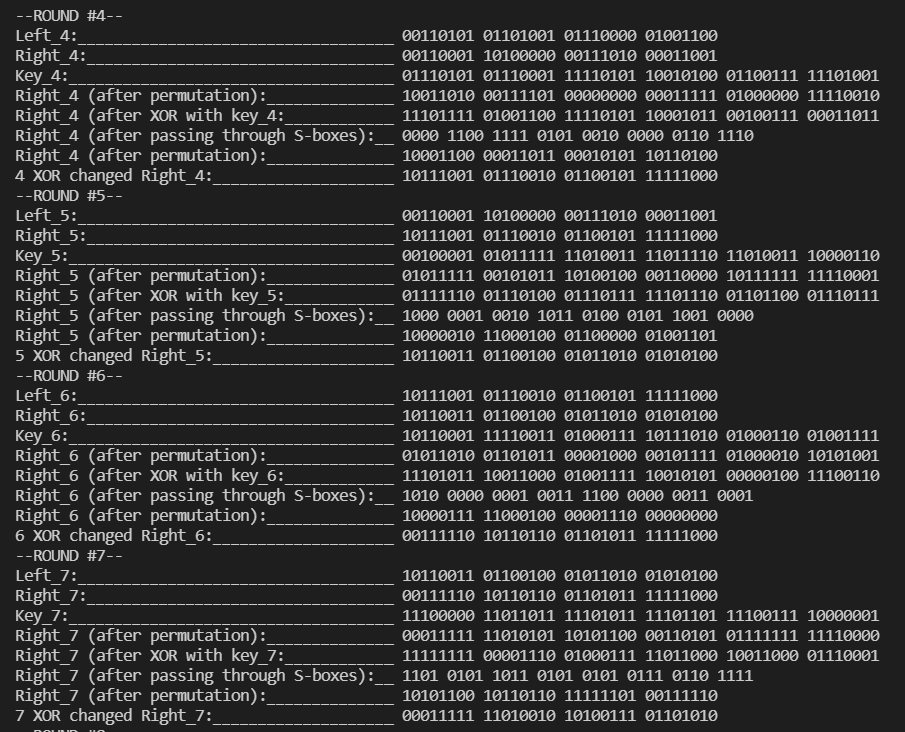
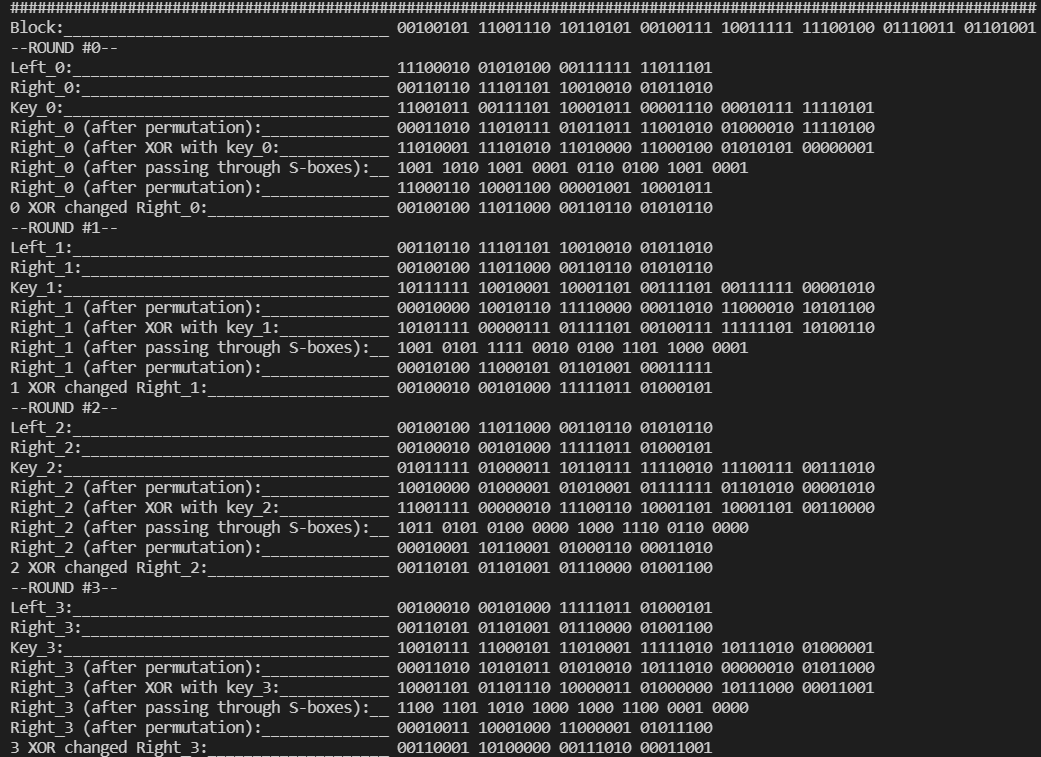
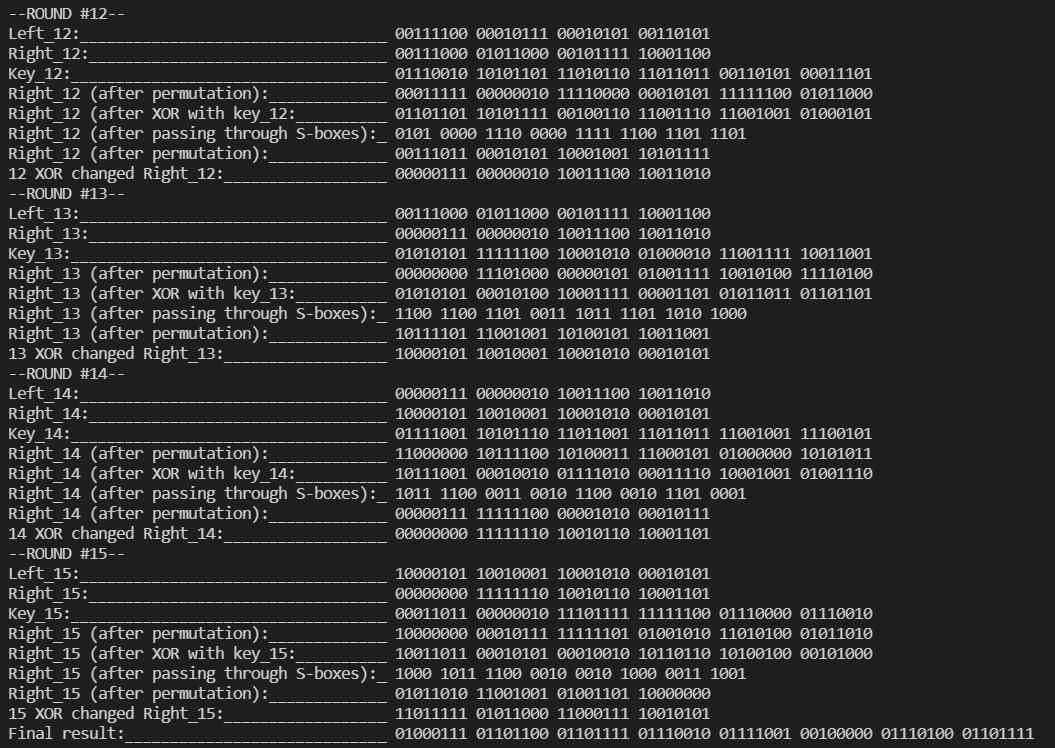
    # print("Deciphered (hex): {0}".format(text\_to\_hex\_string(r2)))

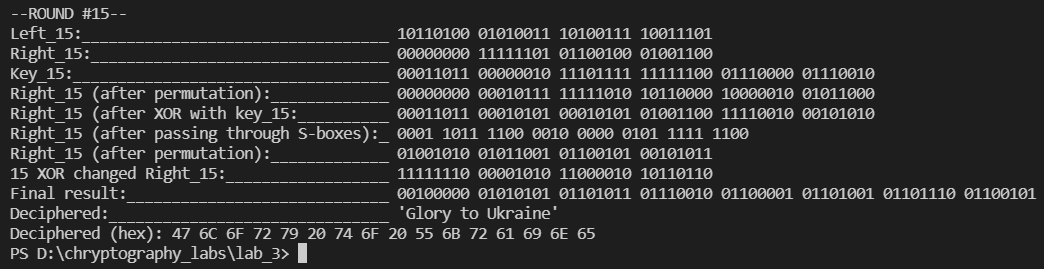
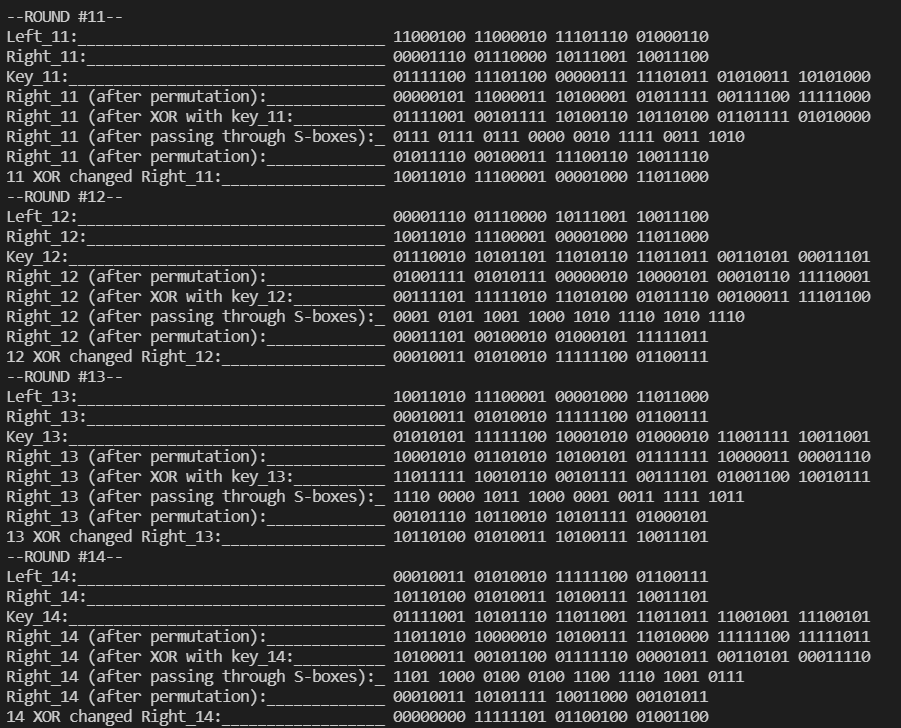
# Додаток 2

Скріншоти шифрування та розшифрування текстів









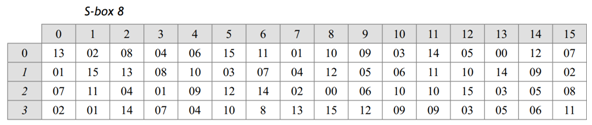
# Відповіді на контрольні запитання

5. Яка ефективна довжина ключа шифру DES?

Відповідь: 56 бітів, оскільки кожен восьмий біт з даного 64-бітного ключа відкидається.

6. Як працюють нелінійні блоки *S*1 … *S*8? Яким буде результат проходження 6-бітового блоку 111000 через блок *S*8 ?

Відповідь: На вході маємо 48 бітів, що розділяємо на шестибітні блоки кожен і-ий блок опрацьовуємо через і-ий S-box. З і-го шестибітного блоку перший та останній біт в злитті утворюють бінарне число, що відповідає рядку з відповідного і-го S-box’у, біти з другого по п’ятий утворюють бінарне число стовчика і-го S-box`ую. На перетині з’ясованого рядка та стовпчика в і-му S-box’і знаходимо число, що перевоимо в двійкове чотирьохбітне. Наприклад, на вході маємо восьмий шестибітний блок 111000, рядок 01 , стовпчик – 1100,



мал. 3.1 – восьмий S-box

Отже, маємо на пертині число 10, у двійковому форматі – 1010.

1. Що означає лавиновий ефект? Продемонструвати лавиновий ефект при зміні одного біту на вході після трьох раундів шифрування.

Відповідь: Лавиновий ефект при незначній зміні вхідних даних на виході дані змінюються значно. Наприклад, нехай на вході маємо блок

|  |  |  |
| --- | --- | --- |
| Initial | 01000110 01101100 01101111 01110111 01100011 01101000 01100001 01110010 | 01000110 01111100 01101111 01110111 01100011 01101000 01100001 01110010 |
| Left 0 | 11111111 10001000 00001111 01011100 | 11111111 10001010 00001111 01011100 |
| Right 0 | 00000000 11111110 00100110 10011101 | 00000000 11111110 00100110 10011101 |
| Left 1 | 00000000 11111110 00100110 10011101 | 00000000 11111110 00100110 10011101 |
| Right 1 | 00111111 01110111 01011110 01111101 | 00111111 01110101 01011110 01111101 |
| Left 2 | 00111111 01110111 01011110 01111101 | 00111111 01110101 01011110 01111101 |
| Right 2 | 11010101 00101101 01100100 11000100 | 01010101 01101101 01110100 10000100 |
| Left 3 | 11010101 00101101 01100100 11000100 | 01010101 01101101 01110100 10000100 |
| Right 3 | 11110101 01000100 00011111 01101010 | 11110010 11100001 00011010 11000100 |
| Result | 11010101 00101101 01100100 11000100  11110101 01000100 00011111 01101010 | 01010101 01101101 01110100 10000100  11110010 11100001 00011010 11000100 |

1. Які існують режими роботи DES? У чому полягає режим CBC?

Відповідь: існують такі режими роботи DES: електронна кодова книга ЄСВ (Electronic Code Book); зчеплення блоків шифру СВС (Cipher Block Chaining); зворотний зв'язок по шіфротекста СРВ (Cipher FeedBack); зворотний зв'язок по виходу OFB (Output FeedBack). Принцип роботи CBC: дані розбиваються на 64-бітні блоки, перший блок XORиться з вектором ініціалізації (теж 64-бітний блок), у результаті блок шифрується звичним DES алгоритмом, результований зашифрований блок XORиться з наступним блоком тексту, після чого проводиться шифрування за DES і так далі. Дешифрування відбувається так: зашифровані дані, розбиваються на 64-бітні блоки, дешифрується перший блок за DES, а потім результат (а) XORиться з вектором ініціалізації – результат перший розшифрований блок, далі наступний зашифрований блок розшифровується за DES та XORиться з (а) – результат другий розшифрований тект і так далі.