Министерство науки и высшего образования РФ

Федеральное государственное автономное образовательное учреждение

высшего образования

«Московский политехнический университет»

(Московский политех)

Отчёт по курсу «Программирование криптографических алгоритмов»

Лабораторная работа 7. КОМБИНАЦИОННЫЕ ШИФРЫ



Выполнил:

Студент группы 221-352

Иванов В. В.

Проверил преподаватель: Бутакова Н. Г.

Москва 2024г.

**Аннотация**

* **Среда программирования**
  + Visual Studio Code
* **Язык программирования**
  + Python
* **Процедуры для запуска программы** 
  + Visual Studio Code (main.py)
* **Пословица-тест**
  + Тот, кто ложится на два стула, падает на ребра.
* **Текст для проверки работы (не меньше 1000 знаков (1430))**

Жизнь - это удивительное приключение, полное разнообразных событий и встреч. В каждом моменте мы находим что-то новое и уникальное. Стремление к росту и саморазвитию вдохновляет нас на поиск новых горизонтов. Важно помнить, что каждый шаг вперед приносит с собой уроки и опыт.

Разнообразие культур, языков и традиций делает наш мир удивительно богатым. Общение с людьми разных национальностей расширяет кругозор, позволяя нам понимать и уважать друг друга. Взаимное уважение и терпимость создают основу для гармоничного сосуществования.

Природа тоже играет важную роль в нашей жизни. Красота закатов, шум океана, пение птиц - все это напоминает нам о величии мира природы. Забота о окружающей среде становится неотъемлемой частью ответственного образа жизни.

Работа и творчество придают смысл нашим усилиям. Стремление к достижению целей мотивирует нас на новые начинания. Каждый проект, даже самый маленький, приносит удовлетворение и чувство выполненного долга.

Семья и друзья являются надежной опорой в нашей жизни. Обмен историями, веселые посиделки и поддержка в трудные моменты создают теплую атмосферу взаимопонимания и любви.

Таким образом, наша жизнь - это мозаика различных моментов, соединенных воедино. Важно ценить каждый момент и стремиться делать мир вокруг нас ярче и лучше. С любовью, терпением и целеустремленностью мы можем создавать свою уникальную историю, наполненную смыслом и радостью.

**17.МАГМА**

Шифр Магма - симметричный блочный шифр, работает на основе принципа подстановки и перестановки битов данных в блоках. Методом применения ключа к исходному тексту выполняются последовательные перестановки и замены битов, что приводит к зашифрованию сообщения. Этот процесс повторяется для каждого блока данных в тексте.

**Блок-схема программы**

**Режим простой замены**

**Режим гаммирования**

**Код программы с комментариями**

from feistelsNetwork import feistelsNetworkCheckParameters, feistelsNetwork

def MagmaCheckParameters(key, init\_vector, alphabet):

    for digit in init\_vector:

        if not digit.isdigit():

            return False

    for letter in key:

        if letter not in alphabet:

            return False

    return True

def magma(open\_text, key, init\_vector, mode, alphabet):

    def xor\_hex(hex1, hex2):

        binary1 = int(hex1, 16)

        binary2 = int(hex2, 16)

        xor\_result = (binary1 ^ binary2).to\_bytes((max(binary1.bit\_length(), binary2.bit\_length()) + 7)// 8, byteorder='big').hex()

        return xor\_result

    encrypted\_text = ""

    init\_vector = (init\_vector + "0000000000000000")[:16]

    keys = [key[i:i+8] for i in range(0, len(key), 8)]

    rev\_keys = keys = keys[::-1]

    keys = keys + keys + keys

    keys = keys + rev\_keys

    total =""

    for i in range(0, len(open\_text), 16):

        p\_i = open\_text[i:i+16]

        a0 = init\_vector[:8]

        a1 = init\_vector[8:16]

        ek = feistelsNetwork(init\_vector, key, "encrypt", [])

        c\_i = xor\_hex(ek, p\_i)

        total += c\_i

        init\_vector = hex(int(init\_vector, 16)+ 1)[2:]

    encrypted\_text = total

    return encrypted\_text

*# Magma (GOST) Block Cipher*

pi0 = [12, 4, 6, 2, 10, 5, 11, 9, 14, 8, 13, 7, 0, 3, 15, 1]

pi1 = [6, 8, 2, 3, 9, 10, 5, 12, 1, 14, 4, 7, 11, 13, 0, 15]

pi2 = [11, 3, 5, 8, 2, 15, 10, 13, 14, 1, 7, 4, 12, 9, 6, 0]

pi3 = [12, 8, 2, 1, 13, 4, 15, 6, 7, 0, 10, 5, 3, 14, 9, 11]

pi4 = [7, 15, 5, 10, 8, 1, 6, 13, 0, 9, 3, 14, 11, 4, 2, 12]

pi5 = [5, 13, 15, 6, 9, 2, 12, 10, 11, 7, 8, 1, 4, 3, 14, 0]

pi6 = [8, 14, 2, 5, 6, 9, 1, 12, 15, 4, 11, 0, 13, 10, 3, 7]

pi7 = [1, 7, 14, 13, 0, 5, 8, 3, 4, 15, 10, 6, 9, 12, 11, 2]

pi = [pi0, pi1, pi2, pi3, pi4, pi5, pi6, pi7]

MASK32 = 2 \*\* 32 - 1

*# вводимое число x в 32 bits*

*# выводимое число y в 32-bits*

def t(x):

    y = 0

    for i in reversed(range(8)):

        j = (x >> 4 \* i) & 0xf

        y <<= 4

        y ^= pi[i][j]

    return y

*# x 32-bit integer*

def rot11(x):

    return ((x << 11) ^ (x >> (32 - 11))) & MASK32

*# x и k это 32-bit integers*

def g(x, k):

    return rot11(t((x + k) % 2 \*\* 32))

*# x это 64 bits*

*# деление на тюплы*

def split(x):

    L = x >> 32

    R = x & MASK32

    return (L, R)

*# Левый и правый по 32 бита*

*# Возвращается 64-bits*

def join(L, R):

    return (L << 32) ^ R

*# k равно 256-bits.*

*# возвращается список из 32 ключей по 32 бита*

*# первые 8 ключей берутся из деления ключа на 8 частей по 32 бита*

*# оставшиеся ключи повотряют первые восемь*

def magma\_key\_schedule(k):

    keys = []

    for i in reversed(range(8)):

        keys.append((k >> (32 \* i)) & MASK32)

    for i in range(8):

        keys.append(keys[i])

    for i in range(8):

        keys.append(keys[i])

    for i in reversed(range(8)):

        keys.append(keys[i])

    return keys

*# число x (текст) 64 bits.*

*# k 256 bits*

*# шифртекст 64-bits*

def magma\_encrypt(x, k):

    keys = magma\_key\_schedule(k)

    (L, R) = split(x)

    for i in range(31):

        print(f'round {i}', hex(L), hex(R))

        (L, R) = (R, L ^ g(R, keys[i]))

    print(f'round {i+1}', hex(L), hex(R))

    return join(L ^ g(R, keys[-1]), R)

*# число x (шифртекст) is 64 bits.*

*# k 256 bits*

*# текст 64-bits*

def magma\_decrypt(x, k):

    keys = magma\_key\_schedule(k)

    keys.reverse()

    (L, R) = split(x)

    for i in range(31):

        print(f'round {i}', hex(L), hex(R))

        (L, R) = (R, L ^ g(R, keys[i]))

    print(f'round {i+1}', hex(L), hex(R))

    return join(L ^ g(R, keys[-1]), R)

def magma\_cipher():

*# ключ*

    k = int('ffeeddccbbaa99887766554433221100f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff', 16)

    my\_text = int('db54c704f8189d20', 16)   *# текст*

    print('\n\nplain text', hex(my\_text))

    print('\n\nstarting encrypt 32-rounds')

    CT = magma\_encrypt(my\_text, k)   *# шифртекст*

    print('\n\nencrypt:', hex(CT))

    print('\n\nstarting decrypt 32-rounds')

    decrypt\_text = magma\_decrypt(CT, k)   *# расшифровка*

    print('\n\ndecrypt:', hex(decrypt\_text))

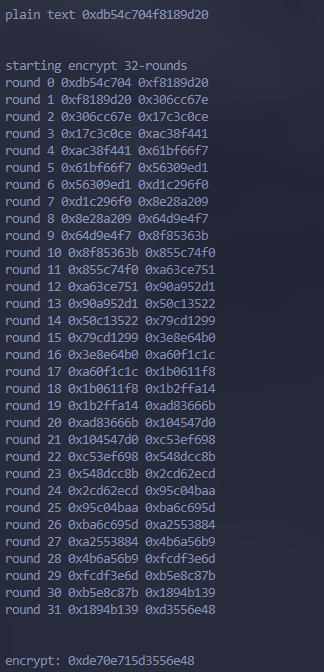
*# расшифровка = текст?*

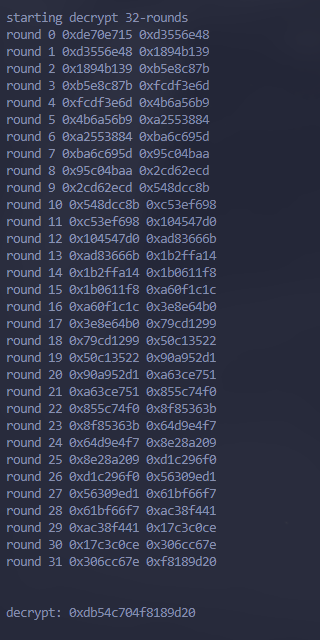
    print('\n\nthey are similar?', decrypt\_text == my\_text)

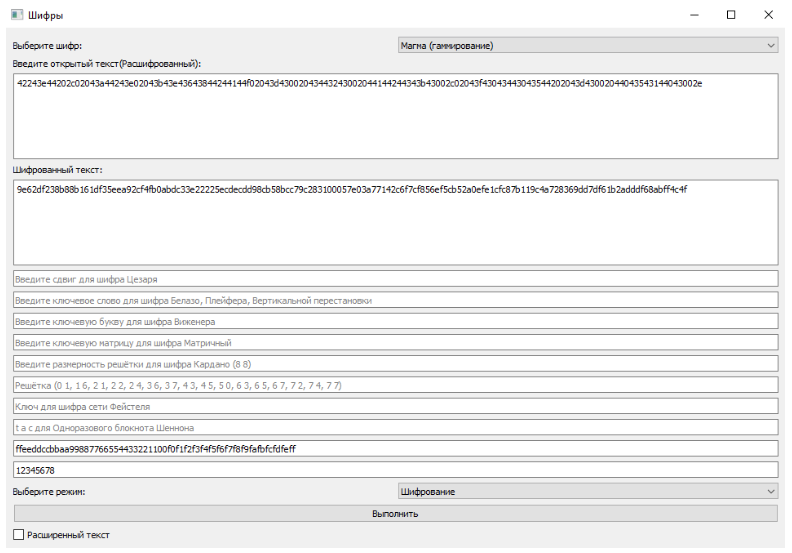
magma\_cipher()

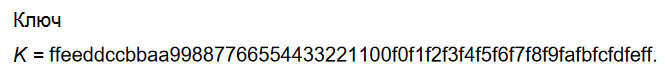
**Тестирование (скрин работы программы и фото карточки**

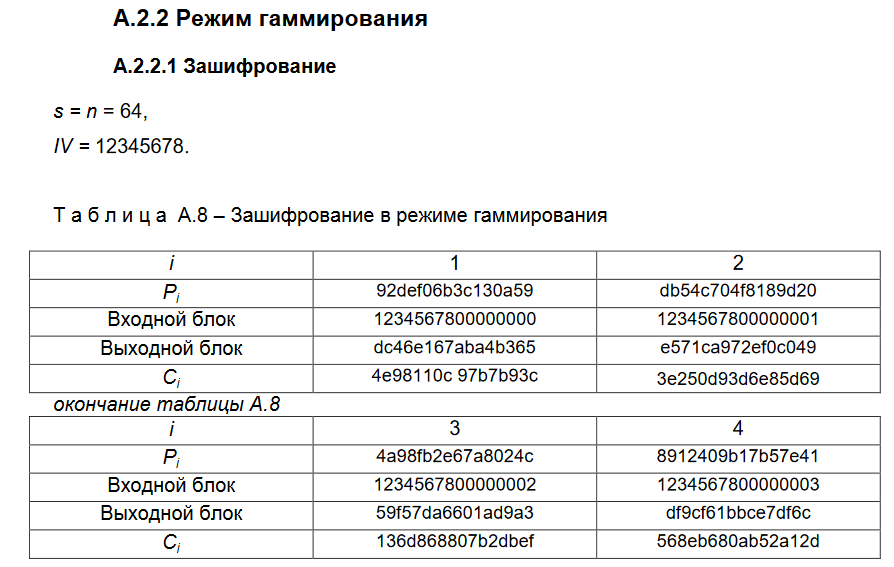
**ручного теста с указанием ключа)**







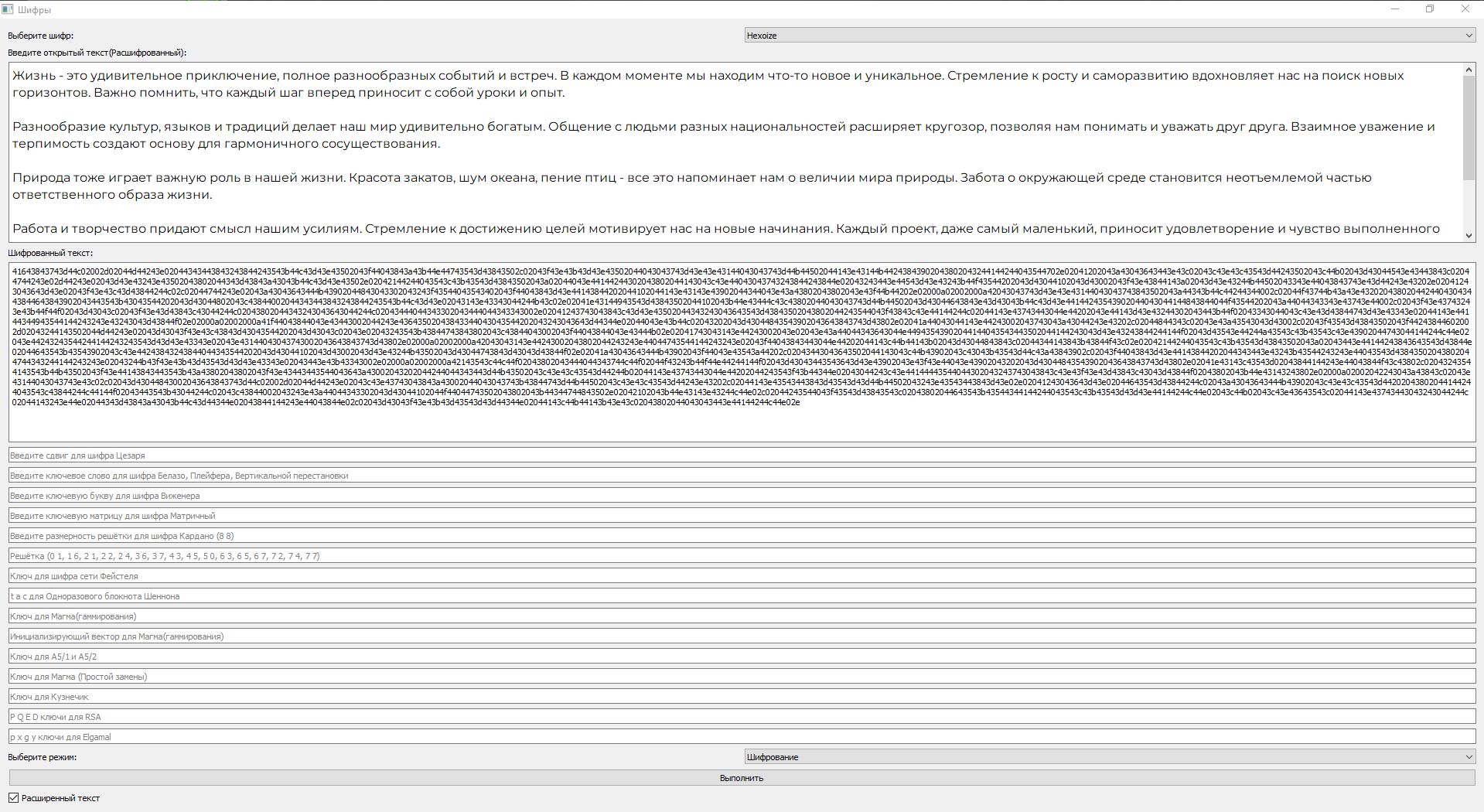
****

****

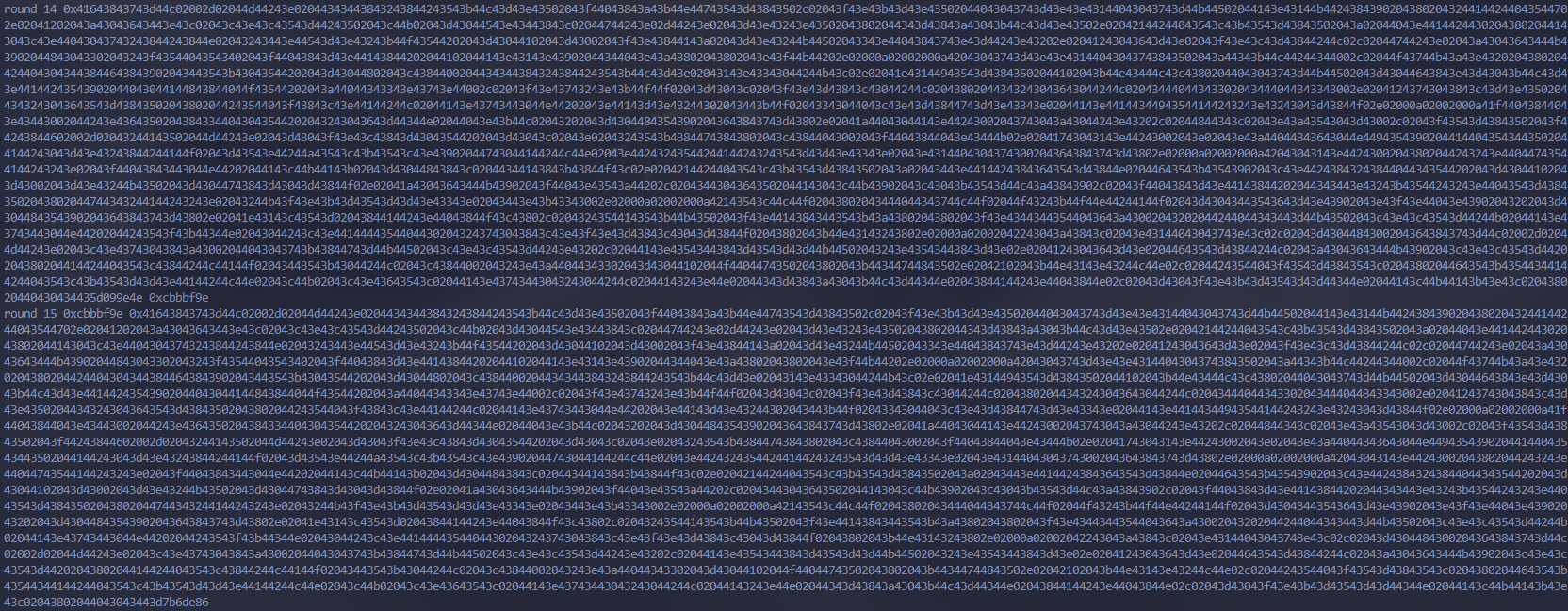
**Работа с текстом не менее 1000 знаков (шифрование и**

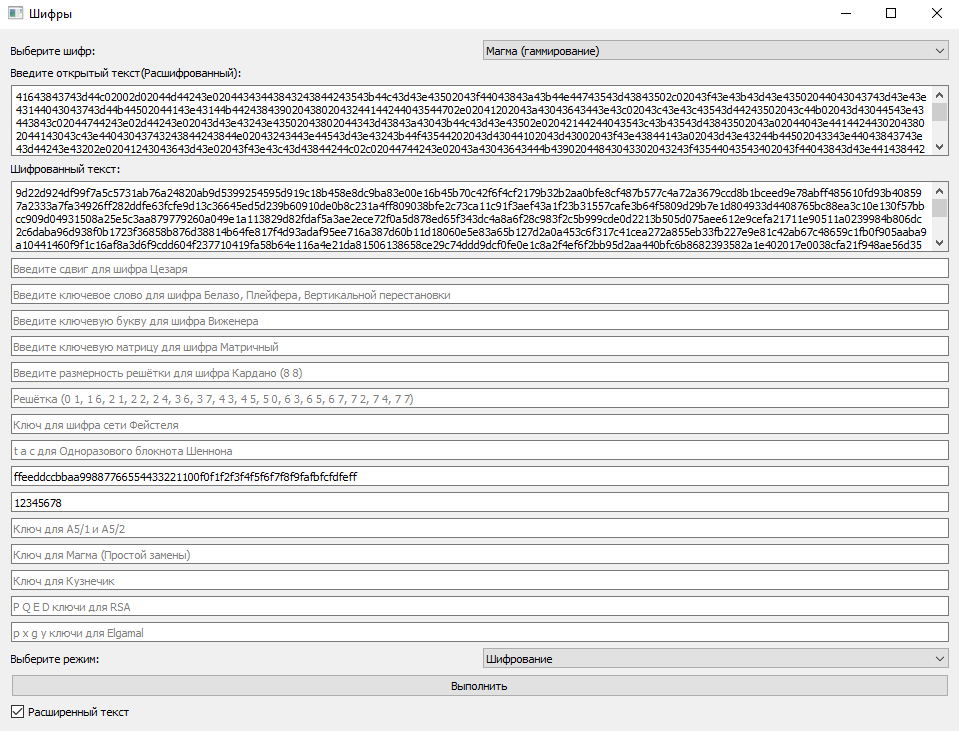
**расшифрование с указанием ключа)**

**Текст в hex**

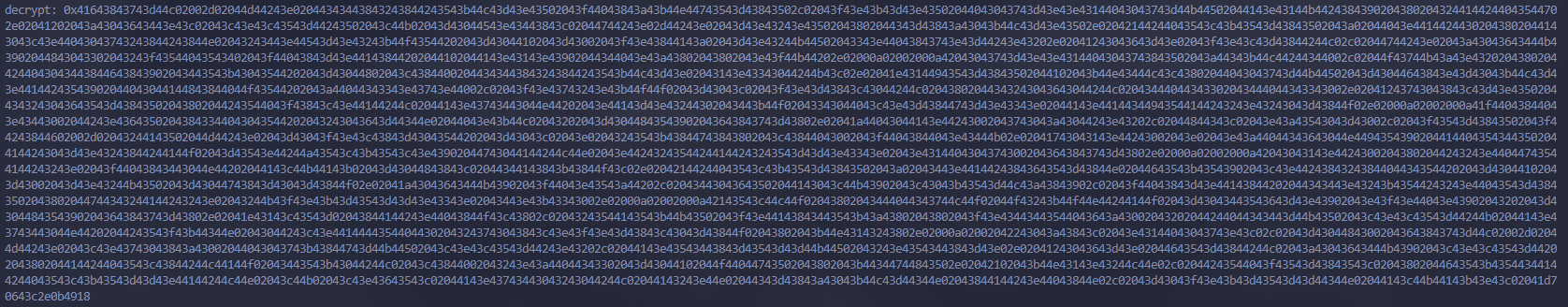


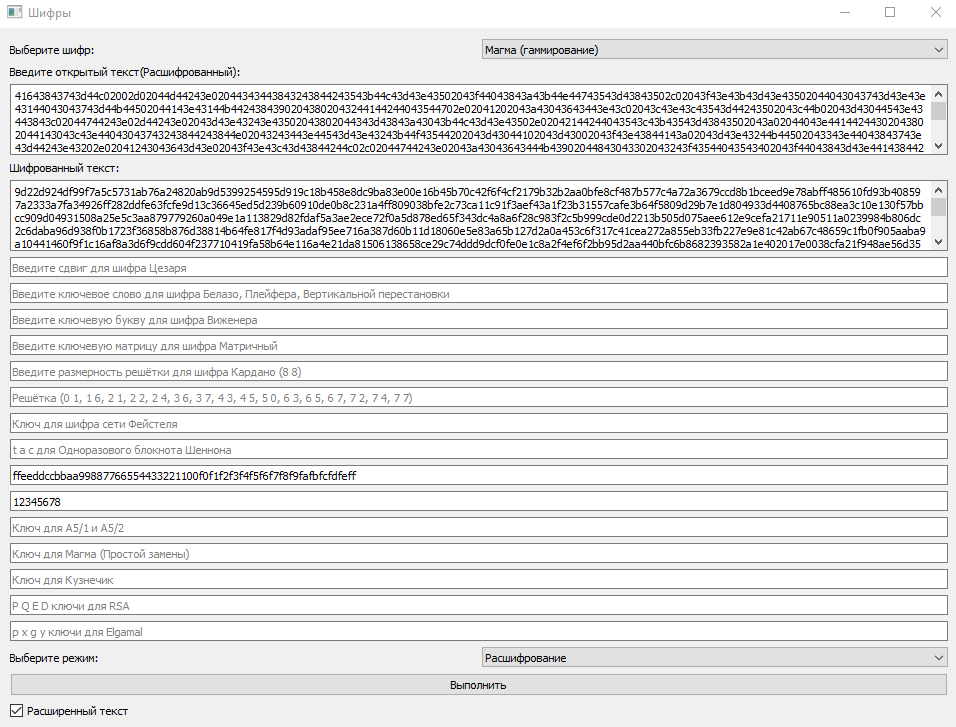
**Зашифрование**



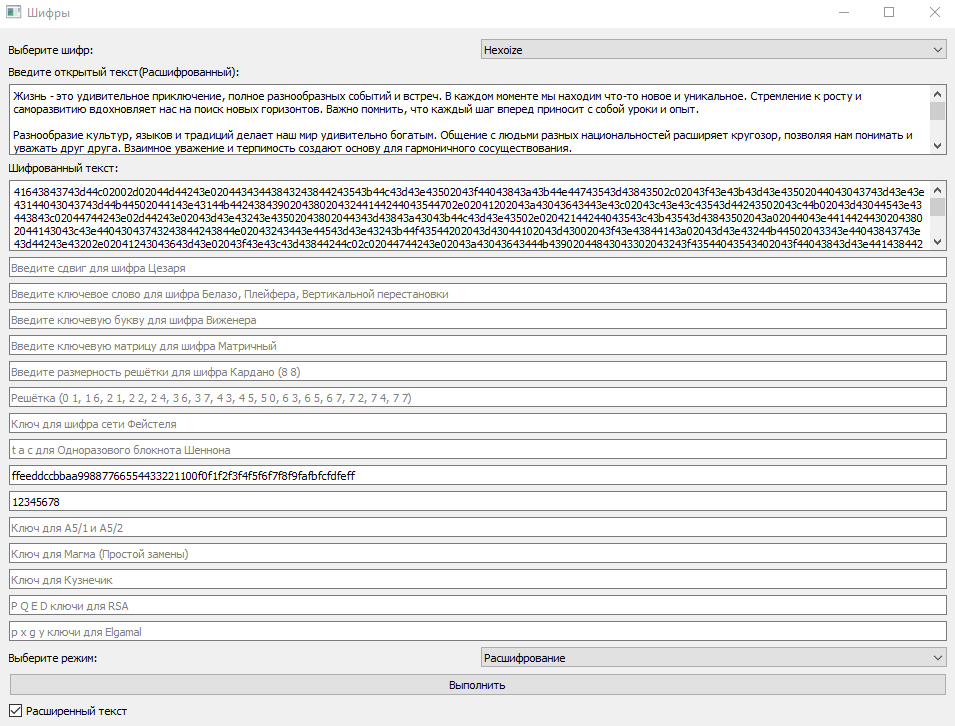


**Расшифрование**





**Hex в текст**



**20.КУЗНЕЧИК**

Шифр Кузнечик - симметричный блочный шифр, основан на принципе замены и перестановки битов входного текста с использованием ключа. Процесс шифрования включает в себя множество раундов замены и перестановки. Размер блока в шифре Кузнечик составляет 128 бит, ключа 256 бит.

**Блок-схема программы**

**Код программы с комментариями**

pi = [252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77,

      233, 119, 240, 219, 147, 46, 153, 186, 23, 54, 241, 187, 20, 205, 95, 193,

      249, 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142, 79,

      5, 132, 2, 174, 227, 106, 143, 160, 6, 11, 237, 152, 127, 212, 211, 31,

      235, 52, 44, 81, 234, 200, 72, 171, 242, 42, 104, 162, 253, 58, 206, 204,

      181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156, 183, 93, 135,

      21, 161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177,

      50, 117, 25, 61, 255, 53, 138, 126, 109, 84, 198, 128, 195, 189, 13, 87,

      223, 245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3,

      224, 15, 236, 222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74,

      167, 151, 96, 115, 30, 0, 98, 68, 26, 184, 56, 130, 100, 159, 38, 65,

      173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165, 125, 105, 213, 149, 59,

      7, 88, 179, 64, 134, 172, 29, 247, 48, 55, 107, 228, 136, 217, 231, 137,

      225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144, 202, 216, 133, 97,

      32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82,

      89, 166, 116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182]

pi\_inv = [165, 45, 50, 143, 14, 48, 56, 192, 84, 230, 158, 57, 85, 126, 82, 145,

          100, 3, 87, 90, 28, 96, 7, 24, 33, 114, 168, 209, 41, 198, 164, 63,

          224, 39, 141, 12, 130, 234, 174, 180, 154, 99, 73, 229, 66, 228, 21, 183,

          200, 6, 112, 157, 65, 117, 25, 201, 170, 252, 77, 191, 42, 115, 132, 213,

          195, 175, 43, 134, 167, 177, 178, 91, 70, 211, 159, 253, 212, 15, 156, 47,

          155, 67, 239, 217, 121, 182, 83, 127, 193, 240, 35, 231, 37, 94, 181, 30,

          162, 223, 166, 254, 172, 34, 249, 226, 74, 188, 53, 202, 238, 120, 5, 107,

          81, 225, 89, 163, 242, 113, 86, 17, 106, 137, 148, 101, 140, 187, 119, 60,

          123, 40, 171, 210, 49, 222, 196, 95, 204, 207, 118, 44, 184, 216, 46, 54,

          219, 105, 179, 20, 149, 190, 98, 161, 59, 22, 102, 233, 92, 108, 109, 173,

          55, 97, 75, 185, 227, 186, 241, 160, 133, 131, 218, 71, 197, 176, 51, 250,

          150, 111, 110, 194, 246, 80, 255, 93, 169, 142, 23, 27, 151, 125, 236, 88,

          247, 31, 251, 124, 9, 13, 122, 103, 69, 135, 220, 232, 79, 29, 78, 4,

          235, 248, 243, 62, 61, 189, 138, 136, 221, 205, 11, 19, 152, 2, 147, 128,

          144, 208, 36, 52, 203, 237, 244, 206, 153, 16, 68, 64, 146, 58, 1, 38,

          18, 26, 72, 104, 245, 129, 139, 199, 214, 32, 10, 8, 0, 76, 215, 116]

*# На вход и выход подаются 128 битные блоки*

*# Реализация функция преобразования S*

def S(x):

    y = 0

    for i in reversed(range(16)):

        y <<= 8

        y ^= pi[(x >> (8 \* i)) & 0xff]

    return y

*# На вход и выход подаются 128 битные блоки*

*# Реализация обратной функции преобразования S*

def S\_inv(x):

    y = 0

    for i in reversed(range(16)):

        y <<= 8

        y ^= pi\_inv[(x >> (8 \* i)) & 0xff]

    return y

*# На вход подаются неотрицательные целые числа*

*# Связанные с ними двоичные многочлены умножаются*

*# Возвращается целое число*

def multiply\_ints\_as\_polynomials(x, y):

    if x == 0 or y == 0:

        return 0

    z = 0

    while x != 0:

        if x & 1 == 1:

            z ^= y

        y <<= 1

        x >>= 1

    return z

*# Returns the number of bits that are used*

*# to store the positive integer integer x.*

def number\_bits(x):

    nb = 0

    while x != 0:

        nb += 1

        x >>= 1

    return nb

*# x is a nonnegative integer*

*# m is a positive integer*

def mod\_int\_as\_polynomial(x, m):

    nbm = number\_bits(m)

    while True:

        nbx = number\_bits(x)

        if nbx < nbm:

            return x

        mshift = m << (nbx - nbm)

        x ^= mshift

*# x,y are 8-bits*

*# The output value is 8-bits*

def kuznyechik\_multiplication(x, y):

    z = multiply\_ints\_as\_polynomials(x, y)

    m = int('111000011', 2)

    return mod\_int\_as\_polynomial(z, m)

*# The input x is 128-bits (considered as a vector of sixteen bytes)*

*# The return value is 8-bits*

def kuznyechik\_linear\_functional(x):

    C = [148, 32, 133, 16, 194, 192, 1, 251, 1, 192, 194, 16, 133, 32, 148, 1]

    y = 0

    while x != 0:

        y ^= kuznyechik\_multiplication(x & 0xff, C.pop())

        x >>= 8

    return y

*# На вход и выход подаются 128 битные блоки*

*# Реализация функции R*

def R(x):

    a = kuznyechik\_linear\_functional(x)

    return (a << 8 \* 15) ^ (x >> 8)

*# На вход и выход подаются 128 битные блоки*

*# Реализация обраной функции R*

def R\_inv(x):

    a = x >> 15 \* 8

    x = (x << 8) & (2 \*\* 128 - 1)

    b = kuznyechik\_linear\_functional(x ^ a)

    return x ^ b

*# На вход и выход подаются 128 битные блоки*

*# Реализация функции L*

def L(x):

    for \_ in range(16):

        x = R(x)

    return x

*# На вход и выход подаются 128 битные блоки*

*# Реализация обратной функции L*

def L\_inv(x):

    for \_ in range(16):

        x = R\_inv(x)

    return x

*# k - ключ, является 256-bits*

*# Алгоритм развертывания ключей*

def kuznyechik\_key\_schedule(k):

    keys = []

    a = k >> 128

    b = k & (2 \*\* 128 - 1)

    keys.append(a)

    keys.append(b)

    for i in range(4):

        for j in range(8):

            c = L(8 \* i + j + 1)

            (a, b) = (L(S(a ^ c)) ^ b, a)

        keys.append(a)

        keys.append(b)

    return keys

*# Входной текст 128 бит*

*# Ключ 256 бит*

def kuznyechik\_encrypt(x, k):

    keys = kuznyechik\_key\_schedule(k)

*# запуск раундов*

    for round in range(9):

        print(hex(x))

        x = L(S(x ^ keys[round]))

    print(hex(x))

    return x ^ keys[-1]

*# The ciphertext x is 128-bits*

*# The key k is 256-bits*

def kuznyechik\_decrypt(x, k):

    keys = kuznyechik\_key\_schedule(k)

    keys.reverse()

    for round in range(9):

        print(hex(x))

        x = S\_inv(L\_inv(x ^ keys[round]))

    print(hex(x))

    return x ^ keys[-1]

def kuznyechik\_cipher():

*# the key*

    k = int('8899aabbccddeeff0011223344556677fedcba98765432100123456789abcdef', 16)

    my\_text = int('0x42243e44202c02043a44243e02043b43e43643844244144f02043d43002043443243002044144244343b43002c02043f4304344304354cb5b7b072df04ca7', 16)

    print('\n\nplain text', hex(my\_text))

    print('\n\nstarting encrypt 9-rounds')

    CT = kuznyechik\_encrypt(my\_text, k)

    print('\n\nencrypt:', hex(CT))

    print('\n\nstarting decrypt 9-rounds')

    decrypt\_text = kuznyechik\_decrypt(CT, k)

    print('\n\ndecrypt:', hex(decrypt\_text))

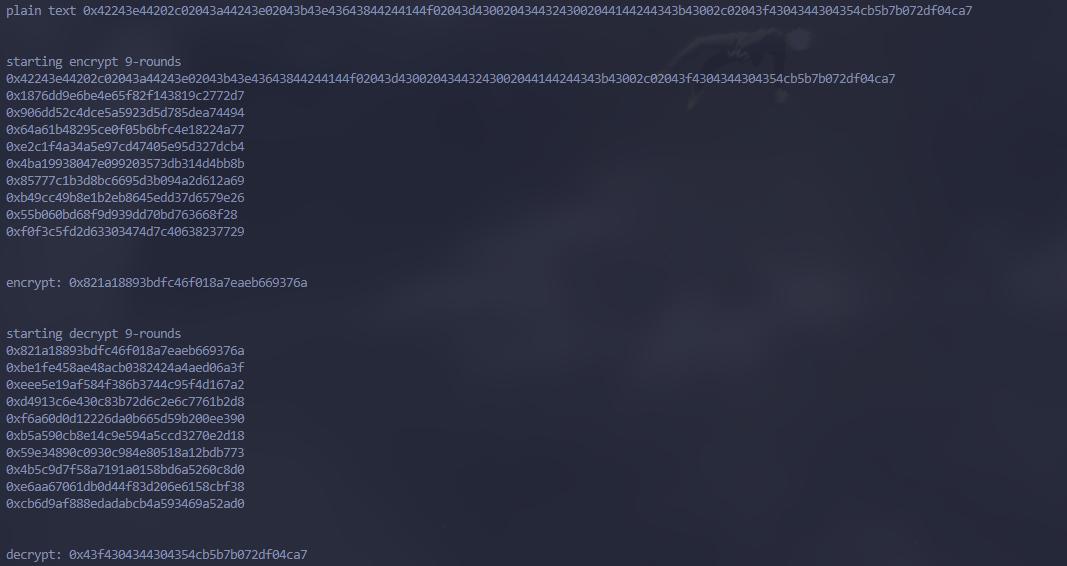
*# Сравнение текста исходного и конечного*

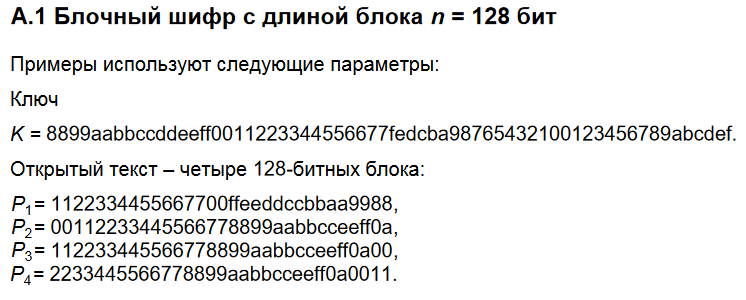
    print('\n\nthey are similar?', decrypt\_text == my\_text)

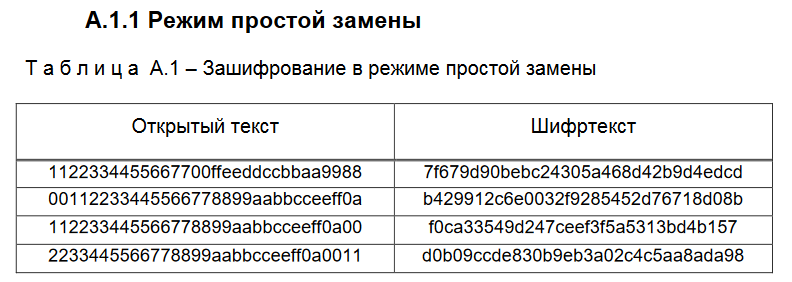
kuznyechik\_cipher()

**Тестирование (скрин работы программы и фото карточки**

**ручного теста с указанием ключа)**



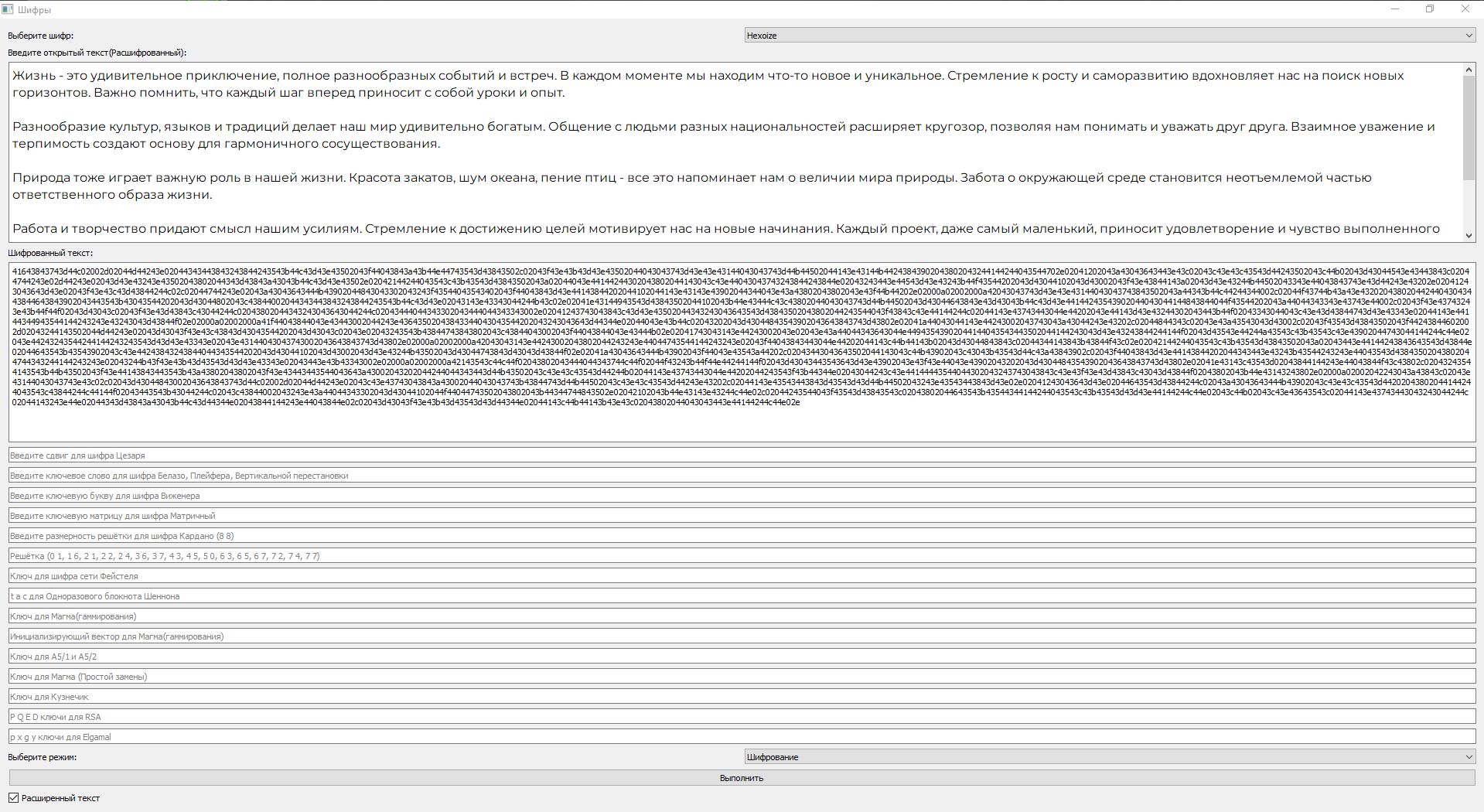




**Работа с текстом не менее 1000 знаков (шифрование и**

**расшифрование с указанием ключа)**

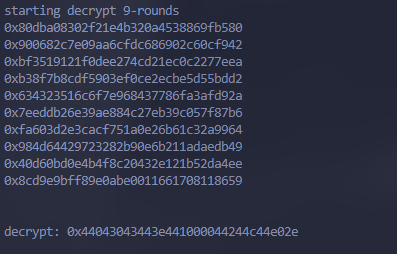
**Текст в hex**



**Зашифрование**



**Расшифрование**



**Hex в текст**

