Оглавление

Алгоритм SPN

Рассмотрим алгоритм шифрования, построенный на основе сети SPN, структура которого показана на рис. 1. Здесь $X = (x_1, x_2, ..., x_{16})$ - 16-ти битовый блок открытого (исходного) сообщения, $Y = (y_1, y_2, ..., y_{16})$ - 16-ти битовый блок закрытого (зашифрованного) сообщения.

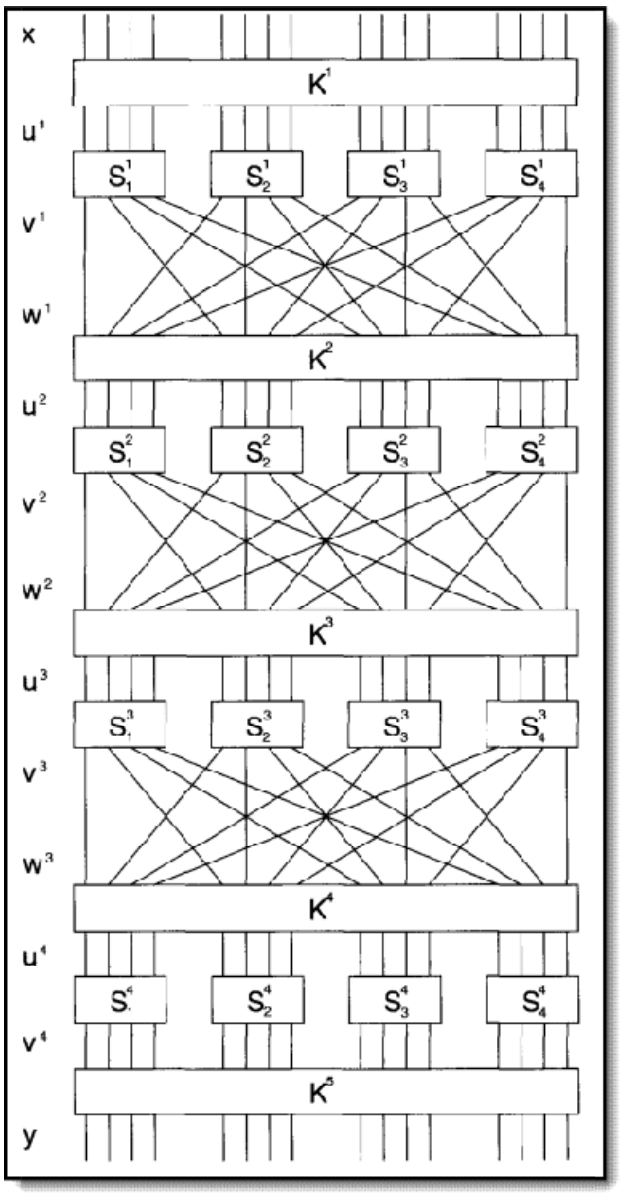


Рисунок 1 – Структура алгоритма шифрования, построенного на основе сети SPN [1, 2, 3]

В основе алгоритма – последовательное применение двух основных преобразований: замены π_S

$$\pi_S : \{0,1\}^l \to \{0,1\}^l$$

и перестановки π_P

$$\pi_P : \{1,...lm\} \to \{1,...,lm\},$$

где $lm \in Z$ - размер блока. В алгоритме, представленном на рис. 1, l=4, m =4.

Замена бит в блоке

Преобразование π_S можно задать в виде таблицы, где первая строка задает вход (z), а вторая строка – выход ($\pi_S(z)$). Табл.1 задает используемое в данном алгоритме преобразование π_S.

Таблица 1 - Замена

Вход	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Выход	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7

На схемах, описывающих алгоритмы шифрования, преобразование замены принято обозначать, как показано на рис.2.

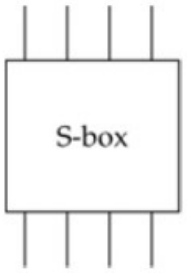


Рисунок 2 – Графическое обозначение замены

В частности, в схеме алгоритма на рис. 1 операции замены обозначены именно таким образом – в виде S-блоков замены. На вход блока замены поступает 4-х битовое значение, на выходе блока замены – измененное в

соответствии с табл.1 4-х битовое значение. Табл.1 можно описать в виде массива (рис.3).

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

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

Рисунок 3 – Реализация блока замены

Для выполнения замены π_S необходимо выполнить следующие шаги:

1. lm-битовый блок разбить на m l-битовых подблоков. Такое разбиение для lm-битового блока $u = (u_1,...,u_{lm})$ можно записать таким образом

$$u = u_{\langle 1\rangle} \| ... \| u_{\langle m\rangle},$$

где $u_{\langle i\rangle} = \left(u_{(i-1)\cdot l+1},...,u_{i\cdot l}\right), i = 1,...,m$

2. Применить преобразование π_S над каждым подблоком:

$$v_{\langle i\rangle} = \pi_S(u_{\langle i\rangle}), i = 1,...,m$$

3. Объединить подблоки в один lm-битовый вход

$$v = v_{\langle 1\rangle} \| ... \| v_{\langle m\rangle}.$$

Пример выполнения замены для 16-битового блока приведен на рис.4.

Путь $l = 4, m = 4$. Тогда 16-ти битовый блок $u = (u_1,...,u_{16})$ разбивается на 4 подблока $u = u_{\langle 1\rangle} \| ... \| u_{\langle 4\rangle}$, где

$$u_{\langle 1\rangle} = (u_1,...,u_4),$$
$$u_{\langle 2\rangle} = (u_5,...,u_8),$$
$$u_{\langle 3\rangle} = (u_9,...,u_{12}),$$
$$u_{\langle 4\rangle} = (u_{13},...,u_{16}).$$

Для блока $u = (0,1,1,0,1,1,0,0,0,1,1,1,0,0,1,0)$ получим 4 подблока:

$$u_{\langle 1\rangle} = (0,1,1,0).$$
$$u_{\langle 2\rangle} = (1,1,0,0),$$
$$u_{\langle 3\rangle} = (0,1,1,1),$$
$$u_{\langle 4\rangle} = (0,0,1,0).$$

После применения преобразования π_S (табл.1) над каждым из подблоков получим

$$v_{\langle 1\rangle} = (1,0,1,1),$$
$$v_{\langle 2\rangle} = (0,1,0,1),$$
$$v_{\langle 3\rangle} = (1,0,0,0),$$
$$v_{\langle 4\rangle} = (1,1,0,1).$$

Таким образом, результатом применения преобразования замены π_S над блоком u будет блок $v = (1,0,1,1,0,1,0,1,1,0,0,0,1,1,0,1)$.

Рисунок 4 – Пример выполнения операции замены

Перестановка бит в блоке

Преобразование π_P задает перестановку бит внутри блока. Данное преобразование удобно задать в виде таблицы, в которой в первой строке (вход z) заданы порядковые номера i бит блока (самый младший бит в блоке имеет номер 0), а во второй строке - выход ($\pi_P(z)$) – результат перестановки бит внутри блока, т.е. на i-ю позицию ставится $\pi_P(i)$ бит блока (табл.2).

Таблица 2 - Перестановка

вход z	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Выход $\pi_p(z)$	15	11	7	3	14	10	6	2	13	9	5	1	12	8	4	0

Указанный способ перестановки реализован с помощью метода pbox() (рис.5).

```
def pbox(self, x):
    y = 0
    for i in range(len(self.p)):
        if (x & (1 << i)) != 0:
            y ^= (1 << self.p[i])
    return y
```

Рисунок 5

Например, для блока $v^1 = 0100010111010001$ результатом применения преобразования π_P (табл.2) будет блок $w^1 = \pi_P(v^1) = 0010111000000111$ (рис.6).

вход z	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		1				1		1	1	1		1				1
выход $\pi_P(z)$	15	11	7	3	14	10	6	2	13	9	5	1	12	8	4	0
			1		1	1	1							1	1	1

Рисунок 6

На рис.6 пустая ячейка в таблице означает нуль. Имеет смысл переставлять только единицы, как это реализовано в методе pbox() (рис.5). Закрашенные одинаковым цветом ячейки таблицы показывают перемещение соответствующих единиц в результате перестановки.

Перестановку (табл.2) удобно задать в виде массива:

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

В схеме на рис.1 перестановка показана традиционным графическим способом как в примере на рис.7.

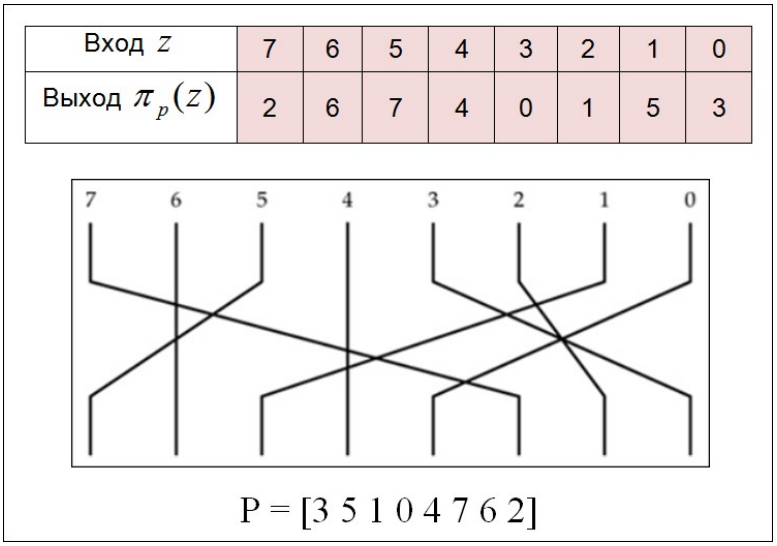


Рисунок 7

Необходимо обеспечить взаимно однозначное соответствие между входом z и выходом $\pi_p(z)$. Также, для задания перестановки в программе требуется существенно меньше памяти. Так, для задания 16-ти битовой перестановки требуется массив из 16 чисел, тогда как для задания 16-ти битовой замены требуется хранить массив из 65536 чисел.

Алгоритм генерации подключей для шифрования

Составной частью алгоритма является описание процедуры получения раундовых ключей – так называемой процедуры генерации подключей. Для рассматриваемого алгоритма шифрования процедура генерации подключей заключается в следующем: все пять подключей получаются последовательным выбором 16 бит из 32 битного ключа $K = (k_1,...,k_{32}) \in \{0,1\}^{32}$ по следующему правилу. Ключ K^r $(1 \le r \le 5)$ состоит из 16 последовательных бит ключа K, начиная с $k_{4 \cdot r-3}$. Например, для ключа $K =$ 982832703 (0011 1010 1001 0100 1101 0110 0011 1111) в результате применения процедуры генерации подключей (расширения ключа) получены следующие раундовые ключи (рис.8).

$K^1 =$ 0011 1010 1001 0100
$K^2 =$1010 1001 0100 1101
$K^3 =$1001 0100 1101 0110
$K^4 =$0100 1101 0110 0011
$K^5 =$1101 0110 0011 1111

Рисунок 8 – Раундовые ключи (подключи)

Алгоритм реализован с помощью метода round_keys() (рис.9).

```
def round_keys(self, k):
    rk = []
    rk.append((k >> 16) & (2**16-1))
    rk.append((k >> 12) & (2**16-1))
    rk.append((k >> 8) & (2**16-1))
    rk.append((k >> 4) & (2**16-1))
    rk.append(k & (2**16-1))
    return rk
```

Рисунок 9

Алгоритм шифрования

Псевдокод алгоритма шифрования приведен на рис.10. N_r - количество раундов шифрования.

Algorithm: $\mathrm{SPN}(x, \pi_S, \pi_P, (K^1, \ldots, K^{\mathrm{Nr}+1}))$

$w^0 \leftarrow x$

for $r \leftarrow 1$ **to** $\mathrm{Nr} - 1$

do
- $u^r \leftarrow w^{r-1} \oplus K^r$
- **for** $i \leftarrow 1$ **to** m
  - **do** $v^r_{<i>} \leftarrow \pi_S(u^r_{<i>})$
- $w^r \leftarrow (v^r_{\pi_P(1)}, \ldots, v^r_{\pi_P(\ell m)})$

$u^{\mathrm{Nr}} \leftarrow w^{\mathrm{Nr}-1} \oplus K^{\mathrm{Nr}}$

for $i \leftarrow 1$ **to** m

do $v^{\mathrm{Nr}}_{<i>} \leftarrow \pi_S(u^{\mathrm{Nr}}_{<i>})$

$y \leftarrow v^{\mathrm{Nr}} \oplus K^{\mathrm{Nr}+1}$

output (y)

Рисунок 10 – Псевдокод алгоритма шифрования

Для открытого 16-ти битового блока $x = 9911$ (0010 0110 1011 0111) и ключа $K = 982832703$ (0011 1010 1001 0100 1101 0110 0011 1111) последовательное применение алгоритма (рис.10) дает результаты, приведенные на рис.11.

$$w^0 = x = 0010\ 0110\ 1011\ 0111$$

Раунд 1

$$u^1 = w^0 \oplus K^1 = 0001\ 1100\ 0010\ 0011$$

$$v^1 = 0100\ 0101\ 1101\ 0001$$

$$w^1 = 0010\ 1110\ 0000\ 0111$$

Раунд 2

$$u^2 = w^1 \oplus K^2 = 1000\ 0111\ 0100\ 1010$$

$$v^2 = 0011\ 1000\ 0010\ 0110$$

$$w^2 = 0100\ 0001\ 1011\ 1000$$

Раунд 3

$$u^3 = w^2 \oplus K^3 = 1101\ 0101\ 0110\ 1110$$

$$v^4 = 1001\ 1111\ 1011\ 0000$$

$$w^3 = 1110\ 0100\ 0110\ 1110$$

Раунд 4

$$u^4 = w^3 \oplus K^4 = 1010\ 1001\ 0000\ 1101$$

$$v^4 = 0110101011101001$$

$$y = v^4 \oplus K^5 = 1011\ 1100\ 1101\ 0110$$

Рисунок 11

В последнем раунде отсутствует перестановка бит после операции замены и выполняется дополнительное сложение по модулю 2 с пятым подключом. Так сделано для того, чтобы использовать ту же самую схему (рис.1) и для расшифрования данных.

Задание 1

а) В файле spn1.py содержится реализация алгоритма шифрования. Пояснить, что делает функция demux():

```
import spn1
e = spn1.SPN1()
x = 15324
print('x={}'.format(bin(x)[2:].zfill(16)))
y = e.demux(x)
print('y={}'.format(y))
```

б) Пояснить, что делает функция mux():

```
Import spn1
e = spn1.SPN1()
x = [9, 11, 4, 2]
y = e.mux(x)
print('y={}'.format(bin(y)[2:].zfill(16)))
```

Задание 2

Написать функцию encrypt_data(self, data, key, rounds), где data – список чисел (данные, прочитанные из файла), key – ключ шифра, rounds – количество раундов.

В этой функции надо сформировать список раундовых ключей шифрования и для каждого числа (16 бит) в списке data вызывать функцию encrypt. Функция возвращает список зашифрованных данных (рис.12).

```
data = [15324, 3453, 34, 12533]
k = 734533245
e = spn1.SPN1()
cypher_data = e.encrypt_data(data, key=k, rounds=4)
print('cypher_data={}'.format(cypher_data))

cypher_data=[8144, 26070, 3827, 38912]
```

Рисунок 12

Задание 3

а) Добавить в класс SPN1 метод asbox(), который выполняет обратную замену:

```
import spn1
e = spn1.SPN1()
x = 9
sx = e.sbox(x)
print('x={}--->s[{}]={}'.format(x, x, sx))
x_ = e.asbox(sx)
print('as[{}]={}'.format(sx, x_))
```

Можно использовать метод списка index().

б) Обратная перестановка π_p^{-1} реализована с помощью метода apbox() (рис.13).

```
def apbox(self, x):
    y = 0
    for i in range(len(self.p)):
        if (x & (1 << self.p[i])) != 0:
            y ^= (1 << i)
return y
```

Рисунок 13 – Обратная перестановка

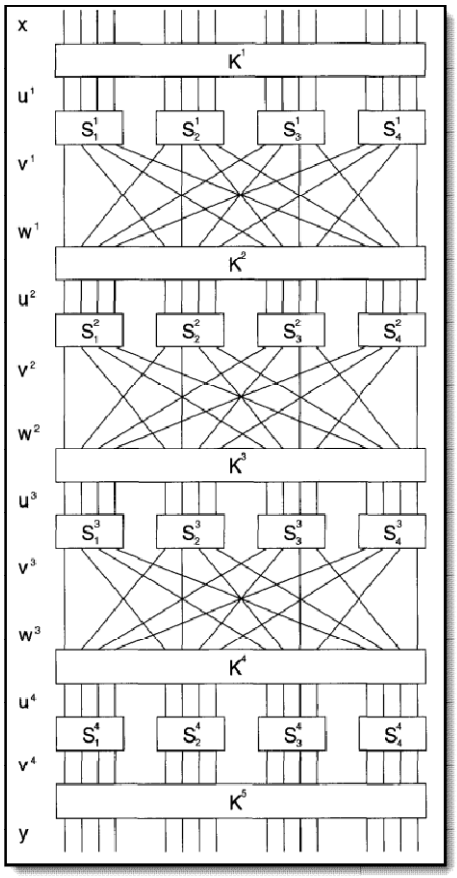
Проверьте корректность выполнения обратной перестановки для p = [2, 5, 6, 8, 4, 14, 0, 7, 11, 10, 12, 1, 15, 9, 3, 13]

```
import spn1
e = spn1.SPN1()
x = int('0010011010110111', 2)
px = e.pbox(x)
print('x={}--->px={}'.format(bin(x)[2:].zfill(16), bin(px)[2:].zfill(16)))
x_ = e.apbox(px)
print('px={}--->x_={}'.format(bin(px)[2:].zfill(16), bin(x_)[2:].zfill(16)))
```

в) Проверьте выполнение равенства $\pi_p^{-1}(x \oplus y) = \pi_p^{-1}(x) \oplus \pi_p^{-1}(y)$, например, для $x = 15324$ и $y = 24681$.

Алгоритм расшифрования

Для расшифрования можно использовать ту же схему, что и для шифрования данных (рис.1). В самом деле, посмотрим, как можно выполнить расшифрование по этой схеме, выполняя обратные преобразования, двигаясь снизу вверх по схеме, т.е. из y получить x (рис.14).



$$v^4 = y \oplus K^5$$
$$u^4 = \pi_s^{-1}(v^4)$$
$$w^3 = u^4 \oplus K^4$$
$$v^3 = \pi_p^{-1}(w^3) = \pi_p^{-1}(u^4 \oplus K^4) = \pi_p^{-1}(u^4) \oplus \pi_p^{-1}(K^4)$$

$$u^3 = \pi_s^{-1}(v^3)$$
$$w^2 = u^3 \oplus K^3$$
$$v^2 = \pi_p^{-1}(w^2) = \pi_p^{-1}(u^3 \oplus K^3) = \pi_p^{-1}(u^3) \oplus \pi_p^{-1}(K^3)$$

$$u^2 = \pi_s^{-1}(v^2)$$
$$w^1 = u^2 \oplus K^2$$
$$v^1 = \pi_p^{-1}(w^1) = \pi_p^{-1}(u^2 \oplus K^2) = \pi_p^{-1}(u^2) \oplus \pi_p^{-1}(K^2)$$

$$u^1 = \pi_s^{-1}(v^1)$$
$$x = u^1 \oplus K^1$$

Рисунок 14

Можно заметить, что получились выражения, такие же, как и при шифровании с учетом того, что операция замены меняется на обратную, перестановка меняется на обратную перестановку и используются другие ключи (рис.15).

$$u^1 = y \oplus L^1$$
$$v^1 = \pi_s^{-1}(u^1)$$
$$u^2 = \pi_p^{-1}(v^1) \oplus L^2$$

$$v^2 = \pi_s^{-1}(u^2)$$
$$u^3 = \pi_p^{-1}(v^2) \oplus L^3$$

$$v^3 = \pi_s^{-1}(u^3)$$
$$u^4 = \pi_p^{-1}(v^3) \oplus L^4$$

$$v^4 = \pi_s^{-1}(u^4)$$
$$y = v^4 \oplus L^5$$

Рисунок 15 – Расшифрование по схеме на рис.1

Таким образом, для расшифрования будет использоваться та же схема, что и для шифрования (рис.1), в которой операции замены и перестановки заменены на их обратные и вместо подключей шифрования будут использоваться подключи расшифрования, алгоритм вычисления которых непосредственно следует из выражений на рис.14, 15.

Алгоритм генерации подключей для расшифрования

Раундовые ключи для расшифрования рассчитываются из раундовых ключей шифрования $[K^1, K^2, K^3, K^4, K^5]$ по следующим формулам:

$$\begin{aligned} L^1 &= K^5 \\ L^2 &= \pi_p^{-1}(K^4) \\ L^3 &= \pi_p^{-1}(K^3) \\ L^4 &= \pi_p^{-1}(K^2) \\ L^5 &= K^1 \end{aligned} \qquad (1)$$

Для ключа key = 0011 1010 1001 0100 1101 0110 0011 1111 рассчитанные значения раундовых ключей для расшифрования приведены на рис.16.

```
K0=0011 1010 1001 0100,    L0=1101 0110 0011 1111
K1=1010 1001 0100 1101,    L1=0100 1110 0011 0101
K2=1001 0100 1101 0110,    L2=1010 0111 0001 1010
K3=0100 1101 0110 0011,    L3=1101 0011 1000 0101
K4=1101 0110 0011 1111,    L4=0011 1010 1001 0100
```

Рисунок 16 – Подключи шифрования и расшифрования

Задание 4

Написать метод round_keys_to_decrypt(self, key) , где key – ключ шифрования (рис.17). Функция формирует список раундовых ключей для расшифрования по формуле (1).

```
def round_keys_to_decrypt(self, key):
    K = self.round_keys(key)
    L = [ ]
    % код
    return L
```

Рисунок 17

Для K =734533245 результат работы функции приведен на рис.18.

```
L0=0001011001111101
L1=1000001100110101
L2=1100100100010010
L3=1110010010001001
L4=0010101111001000
```

Рисунок 18

Задание 5

Написать метод decrypt(self, x, rl, rounds), который выполняет расшифрование одного блока данных (числа x). rl-список ключей для расшифрования, rounds-количество раундов (в данном случае - 4). Структурно метод совпадает с методом шифрования encrypt (рис.19), т.к. используется одна и та же схема (рис.1).

```
# Шифруем одно число
def encrypt(self, x, rk, rounds):
    for i in range(rounds-1):
        x = self.round(x, rk[i])
    x = self.last_round(x, rk[rounds-1], rk[rounds])
return x
```

Рисунок 19

Отличие заключается в том, что вместо round и last_round надо вызывать другие функции, т.к. обработка внутри раундов меняется – используются обратные перестановки и замены. Следовательно, надо еще добавить методы round_decrypt и last_round_decrypt.

Пример вызова метода приведен на рис.20.

```
x = 9911
k = 982832703
print('x={}'.format(bin(x)[2:].zfill(16)))
rk = e.round_keys(k)
y = e.encrypt(x, rk, rounds=4)
lk = e.round_keys_to_decrypt(k)
x_ = e.decrypt(y, lk, rounds=4)
print('y={}'.format(bin(y)[2:].zfill(16)))
print('x_={}'.format(bin(x_)[2:].zfill(16)))

x=0010011010110111
y=1011110011010110
x_=0010011010110111
```

Рисунок 20

Задание 6

Написать функцию decrypt_data(self, data, key, rounds), где data –список чисел (данные, прочитанные из зашифрованного файла), key – ключ шифра, rounds – количество раундов. В этой функции надо сформировать список раундовых ключей расшифрования и для каждого числа (16 бит) в списке data вызвать функцию decrypt. Функция возвращает список расшифрованных данных (рис.21).

```
x = [9911, 12432, 456, 21]
k = 982832703
print('x={}'.format(x))
y = e.encrypt_data(x, k, rounds=4)
x_ = e.decrypt_data(y, k, rounds=4)
print('y={}'.format(y))
print('x_={}'.format(x_))

x=[9911, 12432, 456, 21]
y=[48342, 41317, 8756, 23451]
x_=[9911, 12432, 456, 21]
```

Рисунок 21

Задание 7

Зашифровать и расшифровать содержимое файла (‘123.txt’) с помощью функций encrypt_data и decrypt_data. Для получения содержимого файла в виде списка чисел использовать функцию **read_data_2byte**. Для записи функции в файл использовать функцию **write_data_2byte**.

Убедиться, что расшифрованный после шифрования файл совпадает с исходным (рис.22).

```
from read_write_file import read_data_2byte as read2
from read_write_file import write_data_2byte as write2
e = spn1.SPN1()
# шифрование
data = read2('123.txt')
cypher_data = e.encrypt_data(data, key=452342216, rounds=4)
write2('123_encrypt.txt', cypher_data)
# расшифрование
data = read2('123_encrypt.txt')
decrypt_data = e.decrypt_data(data, key=452342216, rounds=4)
write2('123_decrypt.txt', decrypt_data)
```

Рисунок 22

Задание 8

Расшифровать файл d5_spn_c_all.bmp – зашифрованное шифром на основе сети SPN изображение в формате bmp. Ключ равен 34523456231.

Полученное изображение в формате bmp зашифровать. Сохранить в файле следующие данные: первые 50 байт – исходные (незашифрованные) данные, все последующие байты – зашифрованные. Полученный файл открыть в редакторе. Вставить в отчет исходное и зашифрованное таким образом изображение.

Задание 9

Расшифровать файл d9_spn_c_cbc_all.bmp – зашифрованное шифром на основе сети SPN изображение в формате bmp. Режим шифрования CBC. Ключ равен 345238754631. Вектор инициализации равен 9.

Полученное изображение в формате bmp зашифровать, используя режим шифрования CBC. Сохранить в файле следующие данные: первые 50 байт – исходные (незашифрованные) данные, все последующие байты – зашифрованные. Полученный файл открыть в редакторе.

Задание 10

Расшифровать файл im28_spn_c_ofb_all.bmp. Шифр SPN. Режим OFB. Key = 898387587921 iv= 3253. Зашифровать, оставив первые 50 байт без изменения.

Задание 11

Расшифровать файл im29_spn_c_cfb_all.bmp. Шифр SPN. Режим CFB. Key = 78384265902 iv= 4245. Зашифровать, оставив первые 50 байт без изменения.

Задание 12

Расшифровать файл im30_spn_c_ctr_all.bmp. Шифр SPN. Режим CTR. Key = 3136432567 iv= 7546. Зашифровать, оставив первые 50 байт без изменения.

Задание 13

Дешифровать файл im31_spn_c_ctr_all.bmp. Шифр SPN. Режим CTR. Известны младшие биты ключа: 011010101101001110000111 1, iv= 552211. Зашифровать, оставив первые 50 байт без изменения.

Литература

[1] Бабенко Л.К., Ищукова Е.А. "Современные алгоритмы блочного шифрования и методы их анализа. Учебное пособие для вузов", Гелиос АРВ, 2006, 376 с.

[2] Douglas R. Stinson. Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications), p.616, 2005.

[3] Christopher Swenson. Modern Cryptanalysis: Techniques for Advanced Code Breaking, Wiley Publishing, 2008

[4] Heys, H. (2002). A tutorial on linear and differential cryptanalysis. Cryptologia, 26(3), pp. 189-221.