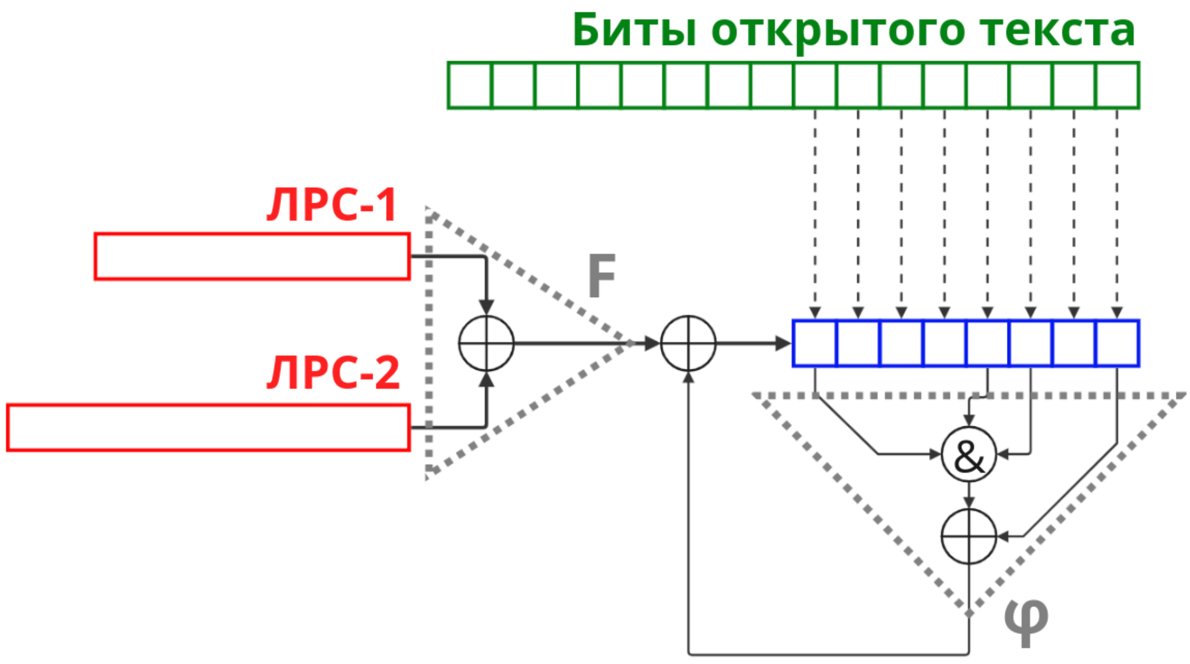
Шифр Колонной Замены



В данной схеме шифрования используется 2 ключа длины 24 и 32 бита. Они являются начальным заполнением ЛРС-1 и ЛРС-2 соответственно. Каждый байт открытого текста помещается в неавтономный регистр сдвига, после чего совершается 10 тактов.

В каждом такте вычисляется:

1. значение функции обратной связи ϕ и складывается с очередным битом ключевого потока, который получается на выходе функции F.
2. Результат помещается со сдвигом в неавтономный регистр.

Полученная после 10 таких тактов байт в регистре является байтом шифртекста. Шифрование продолжается (ЛРС при этом не сбрасываются), пока в открытом тексте не кончатся буквы.

В данной работе предлагается вскрыть шифр по паре открытого и шифр текста, используя криптографическую «слабость» схемы: малая длина управляющей комбинации.

Рассматривая пару батов открытого и шифртекста, требуется найти все подстановки, переводящие байты открытого текста в байт шифртекста. Каждая из подстановок определяется 10 битами ключевого потока. Получая на каждую пару букв открытого и шифртекста список возможных отрезков ключевого потока, нужно перебрать варианты гаммы используя критерий на открытый текст.

1. Реализуем шифр

Реализуем все необходимые компоненты ширфа, в соответствии со схемой шифрования

1.1. Компоненты управляющего блока

В первую очередь определим функцию $\phi(x_7, \dots x_0) = x_0 \oplus x_2 x_3 x_7$.
При этом считаем, что $x = x_7 \dots x_0$ - битовое представление очередного байта, а результрующий бит функции храним в младшем бите выходного байта

```
def phi(x):
    return (
        (x) ^ ((x>>2)&(x>>3)&(x>>7))
    ) & 1
```

Далее определим функцию одного такта g которая зависит от байта состояния $x = (x_7, \dots, x_0)$ и бита ключевого потока γ_i

Причем $g_{\gamma_i}(x) = (\gamma_i \oplus \phi(x), x_7, \dots, x_1) = (\gamma_i \oplus x_0 \oplus x_2 x_3 x_7, x_7, \dots, x_1)$

```
def g(gamma, x):

    return (
        ((x>>1) &0x7F)
        ^
        ((phi(x) ^ (gamma &1)) << 7)
    )
```

Обратное преобразование для g можно вычислить как сдивг влево:

$g_{\gamma_i}^{-1}(y) = (y_6, \dots, y_0, \phi(y_6, \dots, y_0, y_7) \oplus \gamma_i)$

Действительно,

$g_{\gamma_i}(g_{\gamma_i}^{-1}(y)) = g_{\gamma_i}(y_6, \dots, y_0, \phi(y_6, \dots, y_0, y_7) \oplus \gamma_i) =$

$= (\gamma_i \oplus \phi(y_6, \dots, y_0, \phi(y_6, \dots, y_0, y_7) \oplus \gamma_i, y_6, \dots y_0) =$

$= (y_6 y_2 y_1 \oplus \phi(y_6, \dots, y_0, y_7), y_6, \dots, y_0) =$

$= (y_6 y_2 y_1 \oplus y_6 y_2 y_1 \oplus y_7, y_6, \dots, y_0) = (y_7, y_6, \dots, y_0) = y$

```
def inv_g(gamma, y):
    return (
        ((y<<1) &0xFE)
        ^
        phi(
            ((y<<1) &0xFF)
            ^
            ((y>>7) &1)
        )
        ^
        (gamma &1)
    )
```

1.2. Генератор ключевого потока

Теперь необходимо определить регистры, реализуемые шифром в генераторе ключевого потока

Для этого создадим класс, который будет принимать длину состояния в битах, функцию обратной связи и начальное состояние

```
class R:
    def __init__(self, stateLen, feedBack, initState):
        self.s = initState
        self.l = stateLen
        self.f = feedBack

    # извлечение младшего бита состояния
    # с последующм сдивгом вправо и изменением старшего бита
    # в соответствии с функцией обратной связи
    def pop(self):
        #self.debug()
        state = self.s
        self.s = (
            ((self.f(state) & 1) << (self.l-1)) # формрование
старшего бита
            ^
            ((state >> 1) & ((1 << (self.l-1))-1)) # сдвиг вправо
        )
        return state & 1 # младший бит

    def debug(self):
        print(f"\tR[{self.l}]: {bin(self.s)[2:].rjust(self.l, '0')}
: {self.s}")
```

Зададим регистры битами, которые формируют обратную связь:

```
F1=0x1155105
F2=0x101000001
```

Такое представление эквивалентно представлению многочленами:

```
from sympy import Poly, gcd, symbols, parse_expr
```

```
x = symbols('x')
```

```
F1x = Poly(map(int, list(bin( F1 )[2:])), x, modulus=2)
F2x = Poly(map(int, list(bin( F2 )[2:])), x, modulus=2)
```

```
F1x
```

$$\mathrm{Poly}\left(x^{24} + x^{20} + x^{18} + x^{16} + x^{14} + x^{12} + x^{8} + x^{2} + 1, x, domain = \mathbb{F}_2\right)$$

```
F2x
```

$$\mathrm{Poly}\left(x^{32} + x^{24} + 1, x, domain = \mathbb{F}_2\right)$$

1.3. Функции шифрования и расшифрования

В соответствии со схемой, зашифрование каждого байта $x \in X$ реализуется:

1. вычисление 10 знаков гаммы $\gamma_1, \ldots, \gamma_{10}$ как $R1 \oplus R2$
2. 10 тактами функции g на соответствующих знаках гаммы, следующих в прямом порядке - $\gamma_1, \ldots, \gamma_{10}$

Определим соответствующие функции обратной связи регистров

```
def f1(s):
    return bin(s & F1 & 0xFFFFFF).count('1') % 2

def f2(s):
    return bin(s & F2 & 0xFFFFFFFF).count('1') % 2
```

```
def encryption(X, key1, key2):
    R24 = R(stateLen=24, feedBack=f1, initState=key1)
    R32 = R(stateLen=32, feedBack=f2, initState=key2)
    Y = b''
    for x in X:
        y = x
        gammas = [R24.pop() ^ R32.pop() for _ in range(10)]
        for gamma in gammas:
            y = g(gamma, y)
        Y += bytes([y])
    return Y
```

Расшифрование каждого байта $y \in Y$ реализуется:

1. вычисление 10 знаков гаммы $\gamma_1, \ldots, \gamma_{10}$ как $R1 \oplus R2$
2. 10 тактами функции g^{-1} на соответствующих знаках гаммы, следующих в обратном порядке - $\gamma_{10}, \ldots, \gamma_1$

```
def decipher(Y, key1, key2):
    R24 = R(stateLen=24, feedBack=f1, initState=key1)
    R32 = R(stateLen=32, feedBack=f2, initState=key2)
    X = b''
    for y in Y:
        x = y
        gammas = [R24.pop() ^ R32.pop() for _ in range(10)]
        for gamma in gammas[::-1]: # обратный порядок гамм
            x = inv_g(gamma, x)
        X += bytes([x])
    return X
```

1.4. Примеры шифрования сообщения

Попробуем зашифровать сообщение: "Криптографические методы защиты информации"

В первую очередь, сгенерируем ключи:

```
import random

k1 = random.randint(0, 1 << 24)
k2 = random.randint(0, 1 << 32)

print(f"k1: {bin(k1)[2:]}")
print(f"k2: {bin(k2)[2:]}")
```

```
k1: 100001010100101010101110
k2: 10010100000110010110101101000101
```

```
print(f"k1: {hex(k1)[:]}")
print(f"k2: {hex(k2)[:]}")
```

```
k1: 0x854aae
k2: 0x94196b45
```

```
message = "Криптографические методы защиты информации"
bMessage = bytes(message, encoding="utf-8")
```

Вычислим результат зашифрования:

```
b = bytes("test", encoding="utf-8")
encryption(b, k1, k2)
```

```
b'\xa6\x99\xc5`'
```

```
cip = encryption(bMessage, k1, k2)
print(cip)
```

```
b'\x9adj\\?\xacC/\x81\xe1]\xdb\x9a\xe7j\\?
\xae\x03i\xc1\xee\x15\xd5\x9agj\x14?-
C_\xc1%a\x00\xd4\xf27\x18\xaf\x93\xbdt\xc9-
\xfb,\x9a&*q\xf7\xe1C_\x81\xe1\x15V\xb2\xf2\xadH`\xd3Rt\x1a-
I\x00\xd4\xf2>\x18n\x93\xd4t['
```

Расшифруем сообщение:

```
p = decipher(cip, k1, k2)
print(p.decode('utf-8'))
```

```
Криптографические методы защиты информации
```

2. Пробуем выскрыть шифр

Будем рассматривать каждую пару байтов открытого и шифртекста.

2.1. Сообщение для анализа

Пусть имеются пары о.т./ш.т. текстов, использованных на одном ключе. Сгенерируем ключи и сформируем такие пары:

```
k1 = random.randint(0, 1 << 24)
k2 = random.randint(0, 1 << 32)

print(f"k1: {hex(k1)}")
print(f"k2: {hex(k2)}")
```

```
k1: 0xea1b0e
k2: 0x79405758
```

```
bMessage = b'Meet me on Monday at 5pm'
```

```
cip = encryption(bMessage, k1, k2)
print(cip)
```

```
b"F\x1fi2\xb1\x85\xa1Q\xf5!\xd0\x05\xce\xfd\x19uE\xd4\xa2'\x04\xf5\x
c4\x0f"
```

2.2. Нахождение эквивалентного регистра

Для удобства анализа, мы представим два регистра одним.
Для этого вычислим НОК их многочленов:

```
LCMx = ( (F1x * F2x) // gcd(F1x, F2x) ).as_poly()
LCMx
```

$$\operatorname{Poly}\left(x^{40}+x^{34}+x^{26}+x^{24}+x^{8}+x^{2}+1, x, domain=\mathbb{F}_2\right)$$

И в соответствующих бинарной и шеснадцатиричной записях:

```
LCM = int(''.join(map(str, LCMx.as_list())), base=2)
print(f"hex: {hex(LCM)[2:]}")
print(f"bin: {bin(LCM)[2:]}")
```

```
hex: 10405000105
bin: 10000010000000101000000000000000100000101
```

Заметим, что поскольку на каждом символе открытого текста происходит 10 тактов, для хватит k пар о.т./ш.т., где $10k \geq deg(HOK(F_1, F_2))$

В нашем случае, хватит 4-ех пар

```
k = 4
```

2.3. Вычисление всевозможных подстановок управляющей комбинации

Найдем для каждой из 4-х пар подстановки, которые переводят символ открытого текста в соответствующие символ шифрованного текста

Заметим, что, подстановок, реализуемых 10 тактами - ровно $2^{10} = 1024$, каждая из которых определяется битом ключевого потока.
Причем каждая подстановка определяется последовательностью из 2^8 бит

Трудозатраты на вычисление всех подстановок небольшие. Вычислим их:

```
# gammas - 10 битное число
# i-ый бит соотвествует знаку i-ому знаку гаммы
# gammas = 1111000011 соответствует подстановке G:
# G = g[1]g[1]g[0]g[0]g[1]g[1]...g[1],
# где g[k] = функция одного такта со знаком гаммы k
S = {

    gammas: [

        g( (gammas>>9) &1,
```

```
        g( (gammas>>8) &1,
        g( (gammas>>7) &1,
        g( (gammas>>6) &1,
        g( (gammas>>5) &1,
        g( (gammas>>4) &1,
        g( (gammas>>3) &1,
        g( (gammas>>2) &1,
        g( (gammas>>1) &1,
        g( (gammas>>0) &1, x ))))))))))

        for x in range(1<<8)
    ]
    for gammas in range(1<<10)
}
```

Кажое B_i состоит из значений отрезка гаммы (10 знаков), которые может порождать ключевой поток на i-ой букве

```
possibleGammas = {}

for i in range(k):
    possibleGammas[i] = []
    for gammas, G in S.items():
        if G[ bMessage[ i ] ] == cip[ i ]:
            possibleGammas[i].append(gammas)

possibleGammas
```

```
{0: [86, 343, 596, 853],
 1: [17, 272, 659, 978],
 2: [200, 457, 650, 971],
 3: [188, 445, 698, 955]}
```

Проинтерпритируем полученый результат. Первый байт открытого текста может перейти в соответствующий байт шифртекста 4-мя вариантами в зависимости от ключевого потока.

2.4. Определяем эквивалентные функции шифрования и расшифрования

Определим функцю обратной связи эквивалентного регистр сдвига, а также эквивалентные функции ширования и расшифрования через регистр R_{40}. Используем для этого найденный НОК многочленов в пункте 2.2

```python
def lcmF(s):
    return bin(s & LCM & 0xFFFFFFFFFF).count('1') % 2
```

```python
def eq_encryption(Y, key):
    R40 = R(stateLen=40, feedBack=lcmF, initState=key)
    X = b''
    for y in Y:
        x = y
        gammas = [R40.pop() for _ in range(10)]
        for gamma in gammas:
            x = g(gamma, x)
        X += bytes([x])
    return X
```

```python
def eq_decipher(Y, key):
    R40 = R(stateLen=40, feedBack=lcmF, initState=key)
    X = b''
    for y in Y:
        x = y
        gammas = [R40.pop() for _ in range(10)]
        for gamma in gammas[::-1]:
            x = inv_g(gamma, x)
        X += bytes([x])
    return X
```

Проверим, что функции корректны. Для этого попробуем зашифровать и расшифровать сообщение:

```python
key = random.randint(0, 1<<40)
eq_decipher(eq_encryption(b'Secure message!', key), key)
```

```
b'Secure message!'
```

Если посмотрим на первые 40 знаков гаммы, которые порождаются регистрами R_{24} и R_{32}, а также на знаки гаммы регистра R_{40}, то заметим, что они действительно эквивалентны:

```python
R24 = R(stateLen=24, feedBack=f1, initState=k1)
R32 = R(stateLen=32, feedBack=f2, initState=k2)
R24_R32_gammas_40 = ''.join(map(str, [R24.pop() ^ R32.pop() for _ in range(40)]))[::-1]

R40 = R(stateLen=40, feedBack=lcmF, initState=int(R24_R32_gammas_40, base=2))
R40_gammas_40 = ''.join(map(str, [R40.pop() for _ in range(40)]))[::-1]

print(f'R24^R32: {R24_R32_gammas_40}')
print(f'    R40: {R40_gammas_40}')
```

```
R24^R32: 0010111100101000101010100100110001010110
    R40: 0010111100101000101010100100110001010110
```

Здесь первый знак гаммы соответствует правому (младшему) биту - также, как они лежат в регистре на схеме

И вообще, любой отрезок:

```python
Len = 100000

R24 = R(stateLen=24, feedBack=f1, initState=k1)
R32 = R(stateLen=32, feedBack=f2, initState=k2)


R24_R32_gammas = ''.join(map(str, [R24.pop() ^ R32.pop() for _ in range(Len)]))[::-1]

R40 = R(stateLen=40, feedBack=lcmF, initState=int(R24_R32_gammas_40, base=2))

R40_gammas = ''.join(map(str, [R40.pop() for _ in range(Len)]))[::-1]

print(R24_R32_gammas == R40_gammas)
```

```
True
```

2.5. Восстановление отрезка гаммы

```python
possibleGammas
```

```
{0: [86, 343, 596, 853],
 1: [17, 272, 659, 978],
 2: [200, 457, 650, 971],
 3: [188, 445, 698, 955]}
```

Поскольку эквивалентный регистр сдвига длиной 40 бит, то один из 4^4 вариантов ключевого потока является начальным заполнением регистра. Вариантов не много, а значит можно просто перебрать.

```
i = i = 1
for gamma_4 in possibleGammas[3]:
    for gamma_3 in possibleGammas[2]:
        for gamma_2 in possibleGammas[1]:
            for gamma_1 in possibleGammas[0]:
                key = int(
                      bin( gamma_4 )[2:].rjust(10, '0')
                    + bin( gamma_3 )[2:].rjust(10, '0')
                    + bin( gamma_2 )[2:].rjust(10, '0')
                    + bin( gamma_1 )[2:].rjust(10, '0'),
                    base=2
                )
                try:
                    print("{:3d}. eq_decipher( cip, '{}' ) = '{}'".
format(
                        i,
                        hex(key)[2:],
                        eq_decipher(cip, key).decode('utf-8') #
критерий на корректные символы юникода
                    ))
                    i+=1
                except:
                    continue
```

```
  1. eq_decipher( cip, '2f28aa4c56' ) = 'Meet me on Monday at 5pm'
  2. eq_decipher( cip, '2f28aa4d57' ) = 'Meeti|-|&•`•onda21!(a 0{'
  3. eq_decipher( cip, '2f28af4856' ) = 'Meet%%q0jn0Yo&da~`ud%5`}'
  4. eq_decipher( cip, '2f28af4957' ) = 'Meetl4;l#}p    o&da7q58d
i'
  5. eq_decipher( cip, '2f3cba4c56' ) = 'Meet}yu~2zt•onda"4q(q%$1'
  6. eq_decipher( cip, '2f3cba4d57' ) = 'Meet4h= yk4Mondak%1t00d%'
  7. eq_decipher( cip, '2f3cbf4856' ) = 'Meetz1al7xd
o&da'te:t%4!'
  8. eq_decipher( cip, '2f3cbf4957' ) = 'Meet1 )0~k$[o&dane%d50t5'
  9. eq_decipher( cip, '6f68aa4c56' ) = 'Meeth{e!'x Londa34au`%p$'
 10. eq_decipher( cip, '6f68aa4d57' ) = 'Meet!h-}nk`•ondaz%!)!000'
```

```
eq_decipher( cip, '6f68af4856' ) = 'Meetm1q1"z0Zo&da6tuee%`4'
eq_decipher( cip, '6f68af4957' ) = 'Meet$ ;mkko&da}e59$0  '
eq_decipher( cip, '6f7cba4c56' ) = 'Meet5mu•xnt•ondaj q)15$z'
eq_decipher( cip, '6f7cba4d57' ) = 'Meet~~=!3}4Londa#11up dl'
eq_decipher( cip, '6f7cbf4856' ) = 'Meet0%am•no&dao`e;454h'
eq_decipher( cip, '6f7cbf4957' ) = 'Meety4)16•$Xo&da&q%eu t|'
```

Таким образом мы сократили перебор ключа с 2^{40} до 2^{8} !