**Задание:** Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма СТБ 34.101.31-2011 в различных режимах.

СТБ 34.101.31 – блочный шифр с 256-битным ключом и 8 циклами криптопреобразований, оперирующий с 128-битными блоками. Криптографические алгоритмы стандарта построены на основе базовых режимов шифрования блоков данных. Алгоритмы шифрования, описанные в стандарте:

* режим простой замены;
* режим сцепления блоков;
* режим гаммирования с обратной связью;
* режим счётчика;

Также в стандарте описаны:

* выработка имитовставки;
* одновременное шифрование и имитозащиты данных;
* одновременное шифрование и имитозащиты ключей;
* алгоритм хэширования;

Каждая группа включает алгоритм зашифрования и алгоритм расшифрования с использованием секретного ключа. Предполагается, что стороны будут обмениваться сообщениями, используя один ключ, заранее распределен между ними. В режимах простой замены и сцепления блоков шифруются сообщения, которые содержат хотя бы один блок, а в режимах гаммирования с обратной связью и счётчика — сообщения произвольной длины.

**Алгоритм**

1. **Шифрование блока**

**Входные данные и выходные данные**

Входными данными алгоритмов зашифрования и расшифрования являются блок  {\displaystyle X\in \{0,1\}^{128}}и ключ {\displaystyle \theta \in \{0,1\}^{256}.}

Выходными данными является блок {\displaystyle Y\in \{0,1\}^{128}}— результат зашифрования либо расшифрования слова на ключе  либо 

Входные данные для шифрования подготавливаются следующим образом:

* Слово {\displaystyle X} записывается в виде {\displaystyle X=X\_{1}\|X\_{2}\|X\_{3}\|X\_{4},X\_{i}\in \{0,1\}^{32}.}
* Ключ записывается в виде  и определяются тактовые ключи

**Обозначения и вспомогательные преобразования**

Преобразование  ставит в соответствие слову , слово



* циклический сдвиг влево на r бит.

 операция замены 8-битной входной строки подстановкой с рисунка 1.

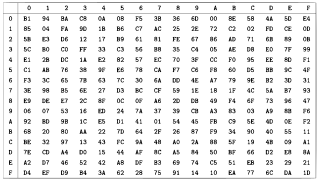


Рисунок 1 – Преобразование Н

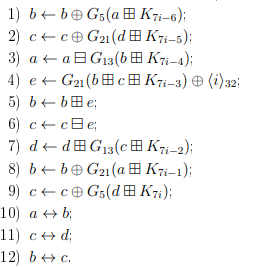
Подстановка {\displaystyle H:\{0,1\}^{8}\rightarrow \{0,1\}^{8}}задается фиксированной таблицей. В таблице используется шестнадцатеричное представление слов {\displaystyle u\in \{0,1\}^{8}.}

{\displaystyle \boxplus } и  {\displaystyle \boxminus -}операции сложения и вычитания по модулю 232{\displaystyle 2^{32}.}

1. **Зашифрование**

Для зашифрования блока {\displaystyle X} на ключе {\displaystyle \theta } выполняются следующие шаги:

1. Установить {\displaystyle a\leftarrow X\_{1},b\leftarrow X\_{2},c\leftarrow X\_{3},d\leftarrow X\_{4}.}
2. Для {\displaystyle i}i = 1,2,… ,8 выполнить:



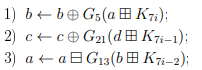
3. Установить {\displaystyle Y\leftarrow b\|d\|a\|c.}

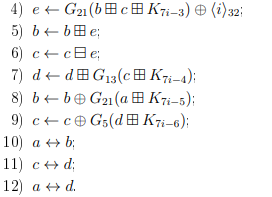
4. Возвратить {\displaystyle Y.}

**3. Расшифрование**

Для расшифрования блока {\displaystyle X} на ключе {\displaystyle \theta } выполняются следующие шаги:

1. Установить {\displaystyle a\leftarrow X\_{1},b\leftarrow X\_{2},c\leftarrow X\_{3},d\leftarrow X\_{4}.}
2. Для {\displaystyle i}i = 8,7,… ,1 выполнить:





3. Установить {\displaystyle Y\leftarrow c\|a\|d\|b.} 

4. Возвратить {\displaystyle Y.} 

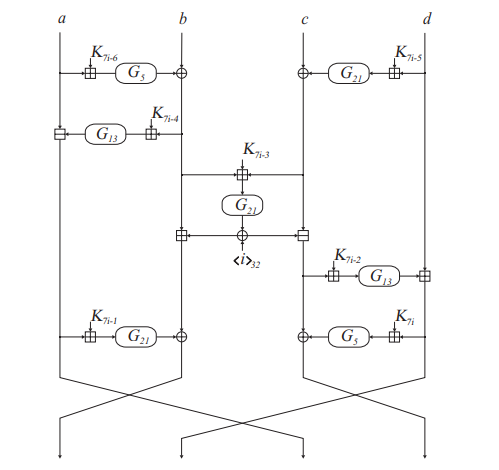


Рисунок 2 – Вычисления на i-м такте шифрования

**Результат работы программы**

Рассмотрим пример программы. В исходном файле содержится текст:

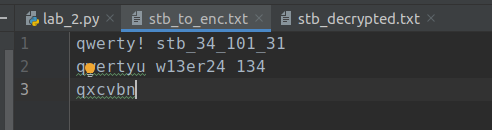


Рисунок 1. Исходный файл для СТБ 34.101.31-2011 шифрования



Рисунок 2. Зашифрованный при помощи СТБ 34.101.31-2011 текст

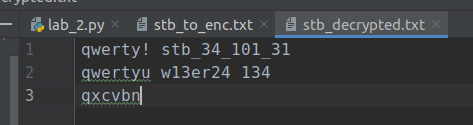


Рисунок 3. Результаты шифрования при помощи СТБ 34.101.31-2011

**Программнй код**

import sys

PLAIN = 1

BLOCK = 2

GAMMING = 3

COUNTER = 4

H = [

[177, 148, 186, 200, 10, 8, 245, 59, 54, 109, 0, 142, 88, 74, 93, 228],

[133, 4, 250, 157, 27, 182, 199, 172, 37, 46, 114, 194, 2, 253, 206, 13],

[91, 227, 214, 18, 23, 185, 97, 129, 254, 103, 134, 173, 113, 107, 137, 11],

[92, 176, 192, 255, 51, 195, 86, 184, 53, 196, 5, 174, 216, 224, 127, 153],

[225, 43, 220, 26, 226, 130, 87, 236, 112, 63, 204, 240, 149, 238, 141, 241],

[193, 171, 118, 56, 159, 230, 120, 202, 247, 198, 248, 96, 213, 187, 156, 79],

[243, 60, 101, 123, 99, 124, 48, 106, 221, 78, 167, 121, 158, 178, 61, 49],

[62, 152, 181, 110, 39, 211, 188, 207, 89, 30, 24, 31, 76, 90, 183, 147],

[233, 222, 231, 44, 143, 12, 15, 166, 45, 219, 73, 244, 111, 115, 150, 71],

[6, 7, 83, 22, 237, 36, 122, 55, 57, 203, 163, 131, 3, 169, 139, 246],

[146, 189, 155, 28, 229, 209, 65, 1, 84, 69, 251, 201, 94, 77, 14, 242],

[104, 32, 128, 170, 34, 125, 100, 47, 38, 135, 249, 52, 144, 64, 85, 17],

[190, 50, 151, 19, 67, 252, 154, 72, 160, 42, 136, 95, 25, 75, 9, 161],

[126, 205, 164, 208, 21, 68, 175, 140, 165, 132, 80, 191, 102, 210, 232, 138],

[162, 215, 70, 82, 66, 168, 223, 179, 105, 116, 197, 81, 235, 35, 41, 33],

[212, 239, 217, 180, 58, 98, 40, 117, 145, 20, 16, 234, 119, 108, 218, 29]

]

class STB:

def \_\_init\_\_(self, key, mode=COUNTER):

count = self.get\_key\_chunks\_counts(key)

self.tmp\_keys = []

# key partition

for i in range(count):

self.tmp\_keys.append(key & 0xFFFF)

key >>= 32

if count == 4:

self.tmp\_keys.extend(self.tmp\_keys[:])

elif count == 6:

self.tmp\_keys.extend([

self.tmp\_keys[0] ^ self.tmp\_keys[1] ^ self.tmp\_keys[2],

self.tmp\_keys[3] ^ self.tmp\_keys[4] ^ self.tmp\_keys[5]

])

self.K = []

for \_ in range(8):

self.K.extend(self.tmp\_keys[:])

self.mode = mode

def get\_key\_chunks\_counts(self, key):

l = len(bin(key)[2:])

l &= (1 << 256) - 1

if 256 >= l > 192:

return 8

elif 192 >= l > 128:

return 6

elif l <= 128:

return 4

def rot\_hi(self, u):

if u < 1 << 31:

return (2 \* u) % (1 << 32)

else:

return (2 \* u + 1) % (1 << 32)

def rot\_hi\_r(self, u, r):

result = u

for i in range(r):

result = self.rot\_hi(result)

return result

def square\_plus(self, u, v):

return (u + v) % (1 << 32)

def square\_minus(self, u, v):

return (u - v) % (1 << 32)

def L(self, X, Y):

l = len(bin(X)) - 2

res = int(bin(Y)[2:l+2], base=2)

return Y

def G(self, r, word):

mask = (1 << 8) - 1

final = 0

for i in range(4):

part = word & mask

word >>= 8

r = part & 0x0F

l = (part & 0xF0) >> 4

result = H[l][r]

result <<= 8 \* i

final += result

return self.rot\_hi\_r(final, r)

def encrypt\_block(self, X):

if self.get\_key\_chunks\_counts(X) != 4:

raise ValueError()

d = X & 0xFFFFFFFF

X >>= 32

c = X & 0xFFFFFFFF

X >>= 32

b = X & 0xFFFFFFFF

X >>= 32

a = X

for i in range(1, 9):

b = b ^ self.G(5, self.square\_plus(a, self.K[7\*i - 7]))

c = c ^ self.G(21, self.square\_plus(d, self.K[7\*i - 6]))

a = self.square\_minus(a, self.G(13, self.square\_plus(b, self.K[7\*i - 5])))

e = self.G(21, self.square\_plus(self.square\_plus(b, c), self.K[7\*i - 4] )) ^ (i % (2 \*\* 32))

b = self.square\_plus(b, e)

c = self.square\_minus(c, e)

d = self.square\_plus(d, self.G(13,self.square\_plus(c, self.K[7\*i - 3])))

b = b ^ self.G(21, self.square\_plus(a, self.K[7\*i - 2]))

c = c ^ self.G(5, self.square\_plus(d, self.K[7\*i - 1]))

a, b = b, a

c, d = d, c

b, c = c, b

return (b << 96) + (d << 64) + (a << 32) + c

def decrypt\_block(self, X):

if self.get\_key\_chunks\_counts(X) != 4:

raise ValueError()

d = X & 0xFFFFFFFF

X >>= 32

c = X & 0xFFFFFFFF

X >>= 32

b = X & 0xFFFFFFFF

X >>= 32

a = X

for i in range(8, 0, -1):

b = b ^ self.G(5, self.square\_plus(a, self.K[7\*i - 1]))

c = c ^ self.G(21, self.square\_plus(d, self.K[7\*i - 2]))

a = self.square\_minus(a, self.G(13, self.square\_plus(b, self.K[7\*i - 3])))

e = self.G(21, self.square\_plus(self.square\_plus(b, c), self.K[7\*i - 4] )) ^ (i % (2 \*\* 32))

b = self.square\_plus(b, e)

c = self.square\_minus(c, e)

d = self.square\_plus(d, self.G(13,self.square\_plus(c, self.K[7\*i - 5])))

b = b ^ self.G(21, self.square\_plus(a, self.K[7\*i - 6]))

c = c ^ self.G(5, self.square\_plus(d, self.K[7\*i - 7]))

a, b = b, a

c, d = d, c

a, d = d, a

return (c << 96) + (a << 64) + (d << 32) + b

def split\_message(self, message):

chunks = []

chunk = 0

while message:

chunk = message & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

chunks.append(chunk)

message >>= 128

return chunks

def join\_message(self, chunks):

answer = 0

for chunk in chunks:

answer <<= 128

answer += chunk

return answer

############################################

# Plain #

############################################

def encrypt\_block\_plain(self, chunks):

results = []

for X in chunks:

Y = self.encrypt\_block(X)

results.append(Y)

return results

def decrypt\_block\_plain(self, chunks):

results = []

for X in chunks:

Y = self.decrypt\_block(X)

results.append(Y)

return results

##############################################

# Coupled #

##############################################

def encrypt\_block\_coupled(self, chunks, S):

results = []

for X in chunks:

Y = self.encrypt\_block(X ^ S)

results.append(Y)

S = Y

return results

def decrypt\_block\_coupled(self, chunks, S):

results = []

for X in chunks:

Y = self.decrypt\_block(X) ^ S

results.append(Y)

S = X

return results

##############################################

# Gamming #

##############################################

def encrypt\_block\_gamming(self, chunks, S):

results = []

for X in chunks:

Y = X ^ self.L(X, self.encrypt\_block(S))

results.append(Y)

S = Y

return results

def decrypt\_block\_gamming(self, chunks, S):

results = []

for X in chunks:

Y = X ^ self.L(X, self.encrypt\_block(S))

results.append(Y)

S = X

return results

##############################################

# Counter #

##############################################

def encrypt\_counter(self, chunks, S):

results = []

s = self.encrypt\_block(S)

for X in chunks:

s = self.square\_plus(s, 1)

Y = X ^ self.L(X, self.encrypt\_block(s))

results.append(Y)

return results

def decrypt\_counter(self, chunks, S):

return self.encrypt\_counter(chunks, S)

def encrypt(self, message: str, \*\*kwargs):

plain\_msg = int.from\_bytes(message.encode(), 'big')

chunks = self.split\_message(plain\_msg)

synchro = kwargs['S']

if self.mode == BLOCK:

results = self.encrypt\_block\_coupled(chunks, synchro)

elif self.mode == PLAIN:

results = self.encrypt\_block\_plain(chunks)

elif self.mode == GAMMING:

results = self.encrypt\_block\_gamming(chunks, synchro)

elif self.mode == COUNTER:

results = self.encrypt\_counter(chunks, synchro)

answer = self.join\_message(results)

return answer.to\_bytes((answer.bit\_length() + 7) // 8, 'big')

def decrypt(self, message: bytes, \*\*kwargs):

plain\_msg = int.from\_bytes(message, 'big')

chunks = reversed(self.split\_message(plain\_msg))

synchro = kwargs['S']

if self.mode == BLOCK:

results = self.decrypt\_block\_coupled(chunks, synchro)

elif self.mode == PLAIN:­­

results = self.decrypt\_block\_plain(chunks)

elif self.mode == GAMMING:

results = self.decrypt\_block\_gamming(chunks, synchro)

elif self.mode == COUNTER:

results = self.decrypt\_counter(chunks, synchro)

answer = self.join\_message(reversed(results))

return answer.to\_bytes((answer.bit\_length() + 7) // 8, 'big').decode()­­­­