Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 1

Компьютерная реализация блочных шифров на примере DES и ГОСТ

Выполнил: Нетецкая Ю.В.

Проверил: Олисейчик В.В.

Минск 2021

**Постановка задачи**

Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритмов DES и ГОСТ 28147-89. Проверить корректность выполнения программных средств на примерах.

**Описание использованных алгоритмов**

**DES** (Data Encryption Standard) — алгоритм для симметричного шифрования, в котором данные шифруются 64-битными блоками и специальным ключом с длиной 56 бит. Алгоритм использует комбинацию нелинейных (S-блоки) и линейных (перестановки E, IP, IP-1) преобразований. Алгоритм преобразует за несколько раундов 64-битный вход в 64-битный выход.

Процесс шифрования состоит из четырех этапов:

1. Выполняется начальная перестановка (IP) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей.
2. Этап состоит из 16 раундов одной и той же функции, которая использует операции сдвига и подстановки.
3. Левая и правая половины выхода последней (16-й) итерации меняются местами.
4. Выполняется перестановка IP-1 результата, полученного на третьем этапе. Перестановка IP-1 инверсна начальной перестановке.

Прямым развитием DES в настоящее время является алгоритм Doube DES (2DES) и Triple DES (3DES). В них шифрование/расшифровка выполняются путём двоекратного и троекратного выполнения алгоритма DES соответственно.

Схема шифрования с помощью алгоритма DES:



Рисунок 1 – Схема шифрования с помощью алгоритма DES.

**Начальная и конечная перестановки.**

Начальная перестановка и ее инверсия определяются стандартной таблицей. Если М – это произвольные 64 бита, то X = IP (M) – это переставленные 64 бита. Если применить обратную функцию перестановки Y = IP-1 (X) = IP-1 (IP(M)), то получится первоначальная последовательность бит.

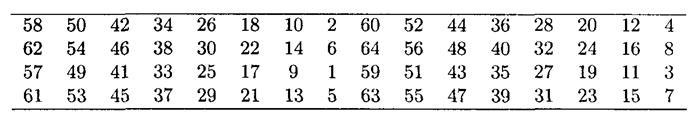


Рисунок 2 - DES. Начальная перестановка

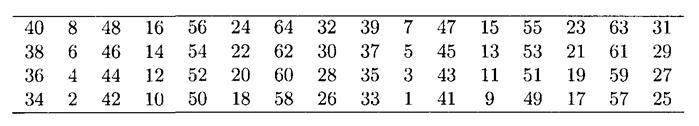


Рисунок 3 - DES. Заключительная перестановка

**Последовательность преобразований отдельного раунда.**

Теперь рассмотрим последовательность преобразований, используемую в каждом *раунде*.

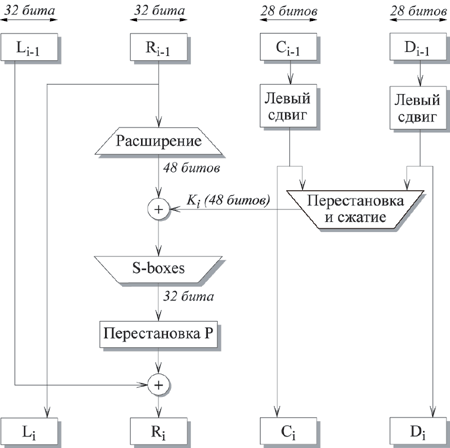


Рисунок 4 - i-ый раунд DES

64-битный входной блок проходит через 16 *раундов*, при этом на каждой итерации получается промежуточное 64-битное значение. Левая и правая части каждого промежуточного значения трактуются как отдельные 32-битные значения, обозначенные *L* и *R*.

Каждую итерацию можно описать следующим образом:

Li = Ri-1

Ri = Li-1 F(Ri-1, Ki) где, – это операцию XOR.

Таким образом, выход левой половины *Li* равен входу правой половины *Ri-1*. Выход правой половины *Ri*является результатом применения операции XOR к *Li-1* и функции *F*, зависящей от *Ri-1* и *Ki*.

Рассмотрим функцию *F* более подробно:

*Ri*, которое подается на вход функции *F*, имеет длину 32 бита. Вначале Ri расширяется до 48 бит, используя таблицу, которая определяет перестановку плюс расширение на 16 бит. Расширение происходит следующим образом. 32 бита разбиваются на группы по 4 бита и затем расширяются до 6 бит, присоединяя крайние биты из двух соседних групп. Например, если часть входного сообщения:

**. . . efgh ijkl mnop . . .**

то в результате расширения получается сообщение:

**. . . defghi hijklm lmnopq . . .**

После этого для полученного 48-битного значения выполняется операция XOR с 48-битным *подключом Ki*. Затем полученное 48-битное значение подается на вход функции подстановки, результатом которой является 32-битное значение.

Подстановка состоит из восьми *S-boxes,* каждый из которых на входе получает 6 бит, а на выходе создает 4 бита. Эти преобразования определяются специальными таблицами. Первый и последний биты входного значения *S-box* определяют номер строки в таблице, средние 4 бита определяют номер столбца. Пересечение строки и столбца определяет 4-битный выход. Например, если входом является 011011, то номер строки равен 01 (строка 1) и номер столбца равен 1101 (столбец 13). Значение в строке 1 и столбце 13 равно 5, т.е. выходом является 0101.

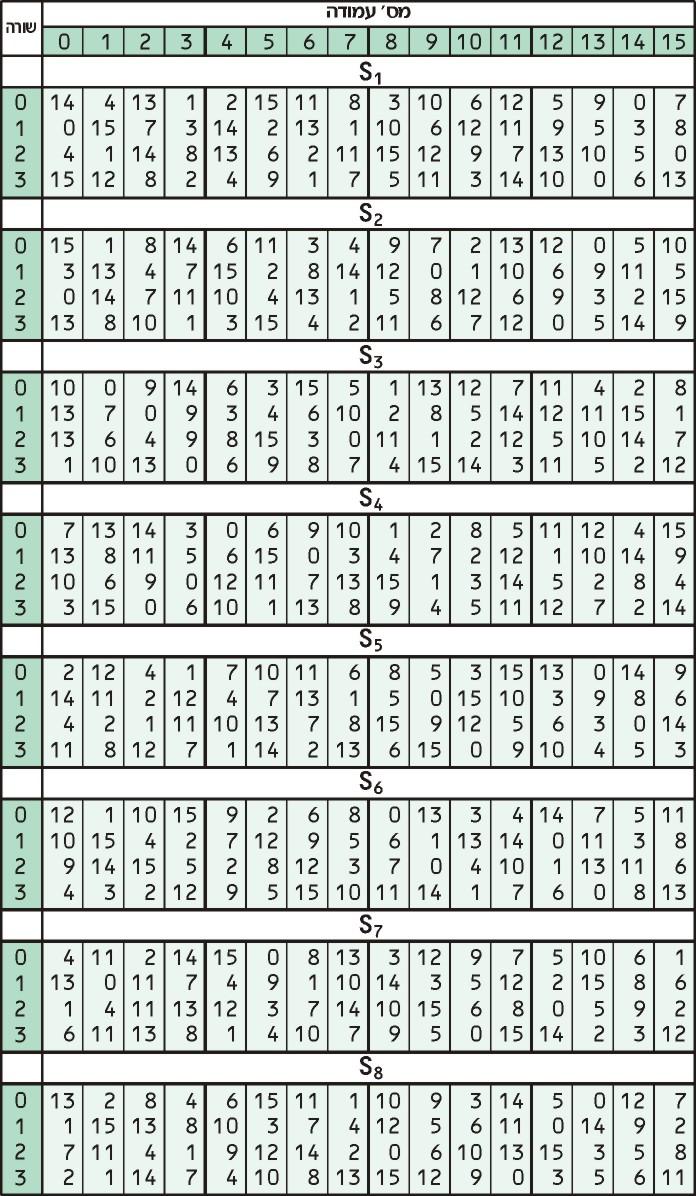


Рисунок 5 - S-boxes

Далее полученное 32-битное значение обрабатывается с помощью перестановки *Р*, целью которой является максимальное переупорядочивание бит, чтобы в следующем *раунде* шифрования с большой вероятностью каждый бит обрабатывался другим *S-box*.

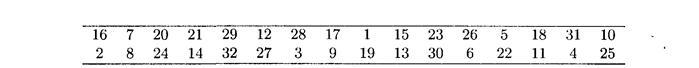


Рисунок 6 - Перестановка с помощью Р-блоков

**Двойной DES.**

Наиболее логичным способом противодействия полному перебору ключа DES выглядит многократное шифрование данных алгоритмом DES с различными ключами. Следующий алгоритм получил название Double DES (двойной DES):

где: — половины двойного ключа алгоритма Double DES, каждая из которых представляет собой обычный 56-битный ключ DES; Е — функция шифрования блока данных обычным алгоритмом DES. Если бы при двойном шифровании DES выполнялось следующее свойство: для любых значений, то двойное шифрование не приводило бы к усилению против полного перебора ключа — всегда нашелся бы такой ключ *k*, *однократное* шифрование которым было бы эквивалентно двукратному шифрованию на ключах, а для нахождения ключа к достаточно было бы перебрать 255 ключей. Double DES действительно удваивает эффективный размер ключа— до 112 битов, а при современном развитии вычислительной техники полный перебор 112-битного ключа невозможен.

**Тройной DES.**

Triple DES (3DES) — симметричный [блочный шифр](https://ru.wikipedia.org/wiki/Блочный_шифр), созданный [Уитфилдом Диффи](https://ru.wikipedia.org/wiki/Диффи,_Уитфилд), [Мартином Хеллманом](https://ru.wikipedia.org/wiki/Хеллман,_Мартин) и Уолтом Тачманном в [1978 году](https://ru.wikipedia.org/wiki/1978_год) на основе алгоритма [DES](https://ru.wikipedia.org/wiki/DES) с целью устранения главного недостатка последнего — малой длины ключа (56 бит), который может быть взломан методом [полного перебора](https://ru.wikipedia.org/wiki/Полный_перебор) ключа. Скорость работы 3DES в 3 раза ниже, чем у DES, но криптостойкость намного выше — время, требуемое для криптоанализа 3DES, может быть в миллиард раз больше, чем время, нужное для вскрытия DES. 3DES используется чаще, чем DES, который легко взламывается при помощи сегодняшних технологий (в [1998 году](https://ru.wikipedia.org/wiki/1998_год) организация [Electronic Frontier Foundation](https://ru.wikipedia.org/wiki/Electronic_Frontier_Foundation), используя специальный компьютер *DES Cracker*, вскрыла DES за 3 дня). 3DES является простым способом устранения недостатков DES. Алгоритм 3DES построен на основе DES, поэтому для его реализации возможно использовать программы, созданные для DES.

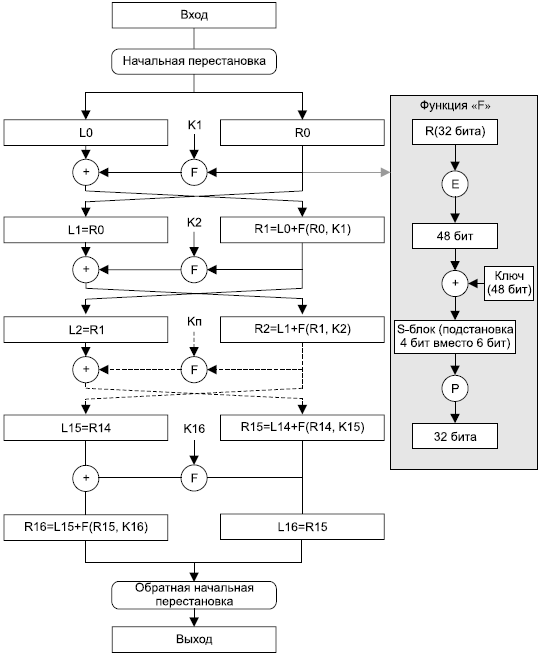
Существует 2 основных варианта шифрования алгоритмом 3DES: 3-key Triple DES и 2-key Triple DES. Как видно из названия, принципиальное различие этих методов - количество ключей (три и два соответственно). В свою очередь, каждый из этих алгоритмов имеет по 2 разных типа: EEE (encryption-encryption-encryption) и EDE (encryption-decryption-encryption) шифрование.

Начнём с трёх ключей:

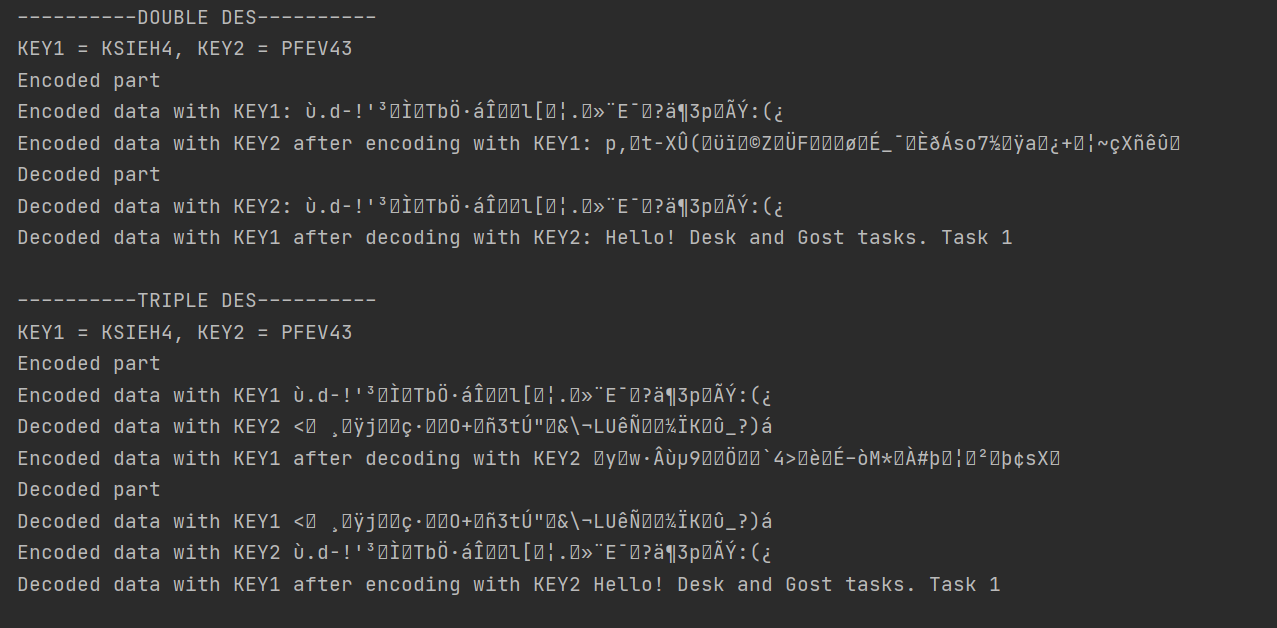
* DES-EEE3: Исходный текст шифруется три раза, используя разные ключи.
* DES-EDE3: Исходный текст шифруется, затем дешифруется (уже другим ключом), затем снова шифруется (третьим ключом).
* DES-EEE2: Исходный текст шифруется три раза, однако ключи на первом и последнем шаге одинаковые.
* DES-EDE2: Исходный текст шифруется, затем дешифруется (другим ключом), затем снова шифруется (ключом, используемым при первом шифровании).

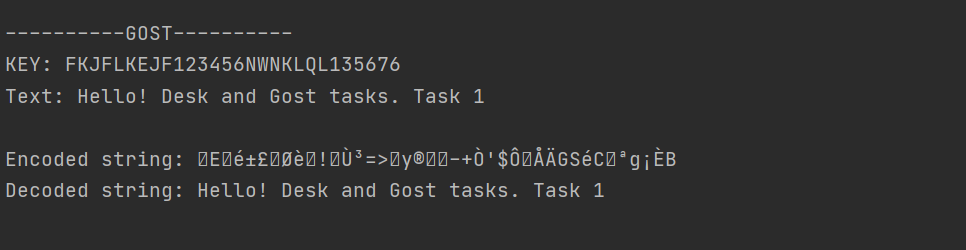
На практике, самый используемый тип 3DES шифрования - DES-EDE3.

Блок схема алгоритма шифрования DES.



**Результат работы программы**





**Код программы**

Des.py

class DES:

def \_\_init\_\_(self):

self.IP = [

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

]

self.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

]

self.S = [

[

[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]

]

]

self.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

]

self.C = [

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,

]

self.D = [

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

]

self.shift = [

1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1

]

self.KP = [

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

]

self.final\_IP = [

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

]

def encrypt(self, data, key):

m = 0

data = [1] + data

m = 0

while m < len(data):

m += 64

data = add\_zeros\_items(data, m)

res = []

for block in range(0, m, 64):

res += self.encrypt\_64(data[block:block+64], key)

return res

def encrypt\_64(self, data, key):

len\_block = 64

key =add\_zeros\_items(key, 56)

data\_IP = []

for i in range(len\_block):

data\_IP.append(data[self.IP[i] - 1])

left = data\_IP[:len\_block//2]

right = data\_IP[len\_block//2:]

for i in range(16):

k\_i = self.generate\_k48(key, i)

left, right = self.feistel\_transform(left, right, k\_i)

new\_data = left + right

ans = [new\_data[self.final\_IP[i] - 1] for i in range(len\_block)]

return ans

def generate\_k48(self, k\_56, iteration):

k\_64 = []

for i in range(8):

k\_64 += k\_56[i \* 7:i \* 7 + 7]

if Counter(k\_56[i \* 7:i \* 7 + 7])[1] % 2 == 0:

k\_64 += [1]

else:

k\_64 += [0]

cd\_56 = [0 for i in range(56)]

for i in range(28):

cd\_56[i] = k\_64[self.C[i] - 1]

j = 0

for i in range(28, 56):

cd\_56[i] = k\_64[self.D[j] - 1]

j += 1

k\_48 = [0 for i in range(48)]

for i in range(48):

k\_48[i] = cd\_56[self.KP[i] - 1]

self.C = self.C[self.shift[iteration]:] + self.C[:self.shift[iteration]]

self.D = self.D[self.shift[iteration]:] + self.D[:self.shift[iteration]]

return k\_48

def feistel\_transform(self, left, right, k):

L\_i = right

right\_48\_bit = [right[self.E[i] - 1] for i in range(48)]

f\_right = [right\_48\_bit[i] ^ k[i] for i in range(48)]

s\_boxes\_32 = []

for i in range(8):

b\_i = f\_right[i \* 6:i \* 6 + 6]

a = int(str(b\_i[0]) + str(b\_i[-1]), 2)

str\_b = ''.join([str(i) for i in b\_i])

b = int(str\_b[1:-1], 2)

s = self.S[i]

b\_4 = bin(s[a][b])[2:]

if len(b\_4) < 4:

b\_4 = '0' \* (4 - len(b\_4)) + b\_4

s\_boxes\_32 += [int(i) for i in b\_4]

function\_r\_k = [s\_boxes\_32[self.P[i] - 1] for i in range(32)]

R\_i = [left[i] ^ function\_r\_k[i] for i in range(32)]

return L\_i, R\_i

def decrypt(self, data, key):

n = len(data)

res = []

for i in range(0, n, 64):

res += self.decrypt\_64(data[i:i+64], key)

while res[0] != 1:

res = res[1:]

return res[1:]

def decrypt\_64(self, data, key):

key = add\_zeros\_items(key, 56)

len\_n = 64

data\_IP = [data[self.IP[i] - 1] for i in range(len\_n)]

l\_i = data\_IP[:len\_n//2]

r\_i = data\_IP[len\_n//2:]

keys = []

for i in range(16):

k\_i = self.generate\_k48(key, i)

keys.append(k\_i)

for i in reversed(range(16)):

l\_i, r\_i = self.feistel\_transform\_dec(l\_i, r\_i, keys[i])

data\_IP = l\_i + r\_i

ans = [data\_IP[self.final\_IP[i] - 1] for i in range(len\_n)]

return ans

def feistel\_transform\_dec(self, left, right, k):

R\_i = left

right\_48\_bit = [left[self.E[i] - 1] for i in range(48)]

f\_right = [right\_48\_bit[i] ^ k[i] for i in range(48)]

s\_boxes\_32 = []

for i in range(8):

b\_i = f\_right[i \* 6:i \* 6 + 6]

a = int(str(b\_i[0]) + str(b\_i[-1]), 2)

str\_b = ''.join([str(i) for i in b\_i])

b = int(str\_b[1:-1], 2)

s = self.S[i]

b\_4 = bin(s[a][b])[2:]

if len(b\_4) < 4:

b\_4 = '0' \* (4 - len(b\_4)) + b\_4

s\_boxes\_32 += [int(i) for i in b\_4]

function\_r\_k = [s\_boxes\_32[self.P[i] - 1] for i in range(32)]

L\_i = [right[i] ^ function\_r\_k[i] for i in range(32)]

return L\_i, R\_i

Gost.py

def add\_zeros\_items(data, k):  
 if len(data) <= k:  
 zeros\_size = k - len(data)  
 data2 = [0 for i in range(zeros\_size)] + data  
 return data2  
  
  
class GOST:  
 def \_\_init\_\_(self):  
 self.TABLE\_S = [  
 [4, 10, 9, 2, 13, 8, 0, 14, 6, 11, 1, 12, 7, 15, 5, 3],  
 [14, 11, 4, 12, 6, 13, 15, 10, 2, 3, 8, 1, 0, 7, 5, 9],  
 [5, 8, 1, 13, 10, 3, 4, 2, 14, 15, 12, 7, 6, 0, 9, 11],  
 [7, 13, 10, 1, 0, 8, 9, 15, 14, 4, 6, 12, 11, 2, 5, 3],  
 [6, 12, 7, 1, 5, 15, 13, 8, 4, 10, 9, 14, 0, 3, 11, 2],  
 [4, 11, 10, 0, 7, 2, 1, 13, 3, 6, 8, 5, 9, 12, 15, 14],  
 [13, 11, 4, 1, 3, 15, 5, 9, 0, 10, 14, 7, 6, 8, 2, 12],  
 [1, 15, 13, 0, 5, 7, 10, 4, 9, 2, 3, 14, 6, 11, 8, 12]  
 ]  
  
 def encrypt(self, data, key):  
 key = add\_zeros\_items(key, 256)  
 data = [1] + data  
 m = ((len(data) // 64) + 1) \* 64  
 data = add\_zeros\_items(data, m)  
 res = []  
 keys = [key[i:i + 32] for i in range(0, len(key), 32)]  
  
 for i in range(0, m, 64):  
 block\_data = data[i:i+64]  
 N1 = block\_data[:32]  
 N2 = block\_data[32:]  
  
 block\_keys = []  
 for i in range(32):  
 block\_keys.append(keys[i % 8])  
  
 for i in range(32):  
 pod\_key = block\_keys[i]  
 N1, N2 = self.transform(N1, N2, pod\_key)  
  
 res += N1 + N2  
  
 return res  
  
  
 def transform(self, l\_i\_prev, r\_i\_prev, key):  
 r\_i = l\_i\_prev  
 l\_i = [0 for \_ in range(32)]  
  
 l\_part = int("".join(str(i) for i in l\_i\_prev), 2)  
 k\_part = int("".join(str(i) for i in key), 2)  
  
 l\_and\_key\_mod = (l\_part + k\_part) % (2 \*\* 32)  
 l = add\_zeros\_items([int(i) for i in "{0:b}".format(l\_and\_key\_mod)], 32)  
 start\_s\_table = [l[i:i + 4] for i in range(0, 32, 4)]  
 res = []  
 for i in range(8):  
 s\_int = int("".join(str(i) for i in start\_s\_table[i]), 2)  
 s\_int = self.TABLE\_S[i][s\_int]  
 start\_s\_table[i] = add\_zeros\_items([int(i) for i in "{0:b}".format(s\_int)], 4)  
 res += start\_s\_table[i]  
 funck = res[11:] + res[:11]  
 for i in range(32):  
 l\_i[i] = r\_i\_prev[i] ^ funck[i]  
 return l\_i, r\_i  
  
  
 def decrypt(self, data, key):  
 result\_data = []  
 key = add\_zeros\_items(key, 256)  
 keys = [key[i:i + 32] for i in range(0, len(key), 32)]  
  
 for i in range(0, len(data), 64):  
 block\_data = data[i:i+64]  
 N1 = block\_data[:32]  
 N2 = block\_data[32:]  
  
 block\_keys = []  
 for i in reversed(range(32)):  
 block\_keys.append(keys[i % 8])  
  
 for i in range(32):  
 pod\_key = block\_keys[i]  
 N2, N1 = self.transform(N2, N1, pod\_key)  
 result\_data += N1 + N2  
  
 while result\_data[0] != 1:  
 result\_data = result\_data[1:]  
  
 return result\_data[1:]

Main.py

def get\_double\_des(KEY1, KEY2):  
 des = DES()  
  
 enc1 = des.encrypt(string\_to\_bin\_list(data), string\_to\_bin\_list(KEY1))  
 enc2 = des.encrypt(enc1, string\_to\_bin\_list(KEY2))  
  
 dec1 = des.decrypt(enc2, string\_to\_bin\_list(KEY2))  
 dec2 = des.decrypt(dec1, string\_to\_bin\_list(KEY1))  
  
 print('KEY1 = {}, KEY2 = {}'.format(KEY1, KEY2))  
 print('Encoded part')  
 print('Encoded data with KEY1: {}'.format(bin\_list\_to\_string(enc1)))  
 print('Encoded data with KEY2 after encoding with KEY1: {}'.format(bin\_list\_to\_string(enc2)))  
 print('Decoded part')  
 print('Decoded data with KEY2: {}'.format(bin\_list\_to\_string(dec1)))  
 print('Decoded data with KEY1 after decoding with KEY2: {}'.format(bin\_list\_to\_string(dec2)))  
  
  
def get\_triple\_des(KEY1, KEY2):  
 des = DES()  
  
 enc1 = des.encrypt(string\_to\_bin\_list(data), string\_to\_bin\_list(KEY1))  
 dec2 = des.decrypt(enc1, string\_to\_bin\_list(KEY2))  
 enc1\_2 = des.encrypt(dec2, string\_to\_bin\_list(KEY1))  
  
 dec1 = des.decrypt(enc1\_2, string\_to\_bin\_list(KEY1))  
 enc2 = des.encrypt(dec1, string\_to\_bin\_list(KEY2))  
 dec1\_2 = des.decrypt(enc2, string\_to\_bin\_list(KEY1))  
  
 print('KEY1 = {}, KEY2 = {}'.format(KEY1, KEY2))  
 print('Encoded part')  
 print('Encoded data with KEY1 {}'.format(bin\_list\_to\_string(enc1)))  
 print('Decoded data with KEY2 {}'.format(bin\_list\_to\_string(dec2)))  
 print('Encoded data with KEY1 after decoding with KEY2 {}'.format(bin\_list\_to\_string(enc1\_2)))  
 print('Decoded part')  
 print('Decoded data with KEY1 {}'.format(bin\_list\_to\_string(dec1)))  
 print('Encoded data with KEY2 {}'.format(bin\_list\_to\_string(enc2)))  
 print('Decoded data with KEY1 after encoding with KEY2 {}'.format(bin\_list\_to\_string(dec1\_2)))  
  
  
def get\_gost(KEY3):  
 gost = GOST()  
 enc = gost.encrypt(string\_to\_bin\_list(data), string\_to\_bin\_list(KEY3))  
 dec = gost.decrypt(enc, string\_to\_bin\_list(KEY3))  
 print('KEY: {}'.format(KEY3))  
 print('Text: {}'.format(data))  
 print('Encoded string: {}'.format(bin\_list\_to\_string(enc)))  
 print('Decoded string: {}'.format(bin\_list\_to\_string(dec)))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 file = open("data.txt", "r")  
 data = file.read()  
 file.close()  
 # установка ключей  
 KEY1 = 'KSIEH4'  
 KEY2 = 'PFEV43'  
 KEY3 = 'FKJFLKEJF123456NWNKLQL135676'  
  
 print('-' \* 10 + 'DOUBLE DES' + '-' \* 10)  
 get\_double\_des(KEY1, KEY2)  
  
 print('-' \* 10 + 'TRIPLE DES' + '-' \* 10)  
 get\_triple\_des(KEY1, KEY2)  
  
 print('-' \* 10 + 'GOST' + '-' \* 10)  
 get\_gost(KEY3)

**Вывод**

В ходе данной лабораторной работе было реализовано шифрование DES и шифрование GOST. Корректность работы алгоритмов было проверено на примере.