МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

ОТЧЁТ

по лабораторной работе 3

по дисциплине

Защита информации

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Симанов В.С.

(подпись) (фамилия, и.,о.)

21-ИС\_\_\_\_\_\_\_\_

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород

2024

Цель: Реализовать стандарт шифрования данных **DES** в режиме в режиме «Обратная связь по выходу».

Задания:

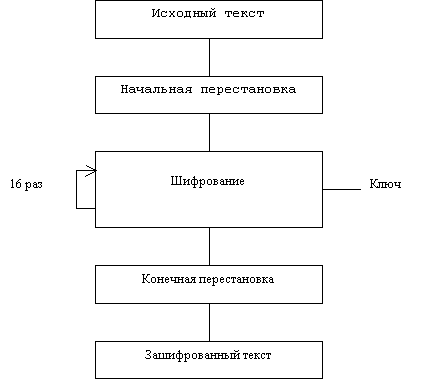
**Задание №3.3**

Реализовать стандарт шифрования данных **DES** в режиме «Обратная связь по шифру».

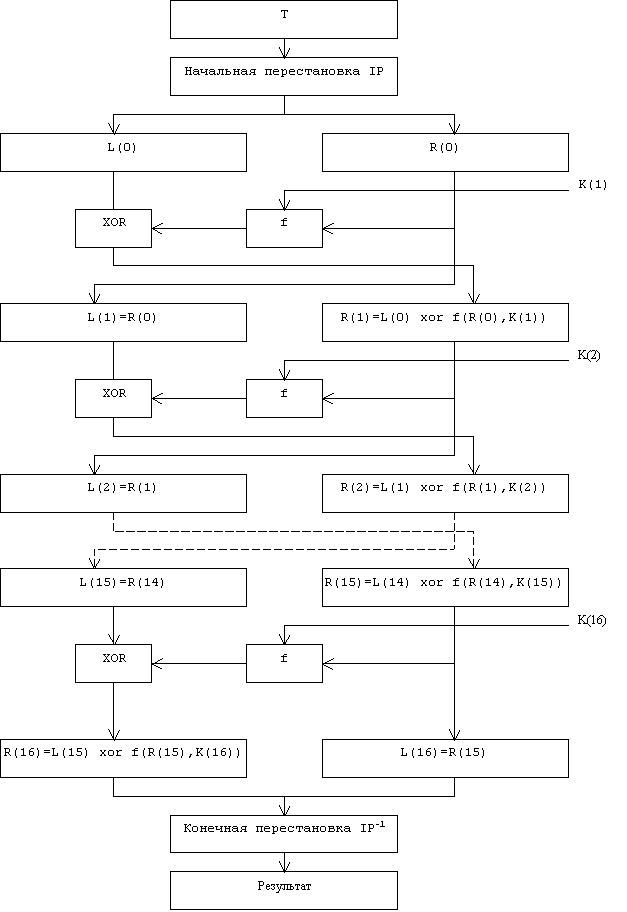
Описание алгоритма:

DES осуществляет шифрование 64-битовых блоков данных с помощью 56-битового ключа. Расшифрование в DES является операцией обратной шифрованию и выполняется путем повторения операций шифрования в обратной последовательности (несмотря на кажущуюся очевидность, так делается далеко не всегда. Позже мы рассмотрим шифры, в которых шифрование и расшифрование осуществляются по разным алгоритмам).

Процесс шифрования заключается в начальной перестановке битов 64-битового блока, шестнадцати циклах шифрования и, наконец, обратной перестановки битов (рис).

  
Рис. Обобщенная схема шифрования в алгоритме DES

Необходимо сразу же отметить, что ВСЕ таблицы, приведенные в данной статье, являются СТАНДАРТНЫМИ, а следовательно должны включаться в вашу реализацию алгоритма в неизменном виде. Все перестановки и коды в таблицах подобраны разработчиками таким образом, чтобы максимально затруднить процесс расшифровки путем подбора ключа. Структура алгоритма DES приведена на рис.

  
Рис. Структура алгоритма шифрования DES

Пусть из файла считан очередной 8-байтовый блок T, который преобразуется с помощью матрицы начальной перестановки IP (табл.1) следующим образом: бит 58 блока T становится битом 1, бит 50 - битом 2 и т.д., что даст в результате: T(0) = IP(T).

Полученная последовательность битов T(0) разделяется на две последовательности по 32 бита каждая: L(0) - левые или старшие биты, R(0) - правые или младшие биты.

Таблица 1: ***Матрица начальной перестановки IP***

58 50 42 34 26 18 10 02

60 52 44 36 28 20 12 04

62 54 46 38 30 22 14 06

64 56 48 40 32 24 16 08

57 49 41 33 25 17 09 01

59 51 43 35 27 19 11 03

61 53 45 37 29 21 13 05

63 55 47 39 31 23 15 07

Затем выполняется шифрование, состоящее из 16 итераций. Результат i-й итерации описывается следующими формулами:

|  |
| --- |
| L(i) = R(i-1)  R(i) = L(i-1) xor f(R(i-1), K(i)) , |

где xor - операция ИСКЛЮЧАЮЩЕЕ ИЛИ.

Функция f называется функцией шифрования. Ее аргументы - это 32-битовая последовательность R(i-1), полученная на (i-1)-ой итерации, и 48-битовый ключ K(i), который является результатом преобразования 64-битового ключа K. Подробно функция шифрования и алгоритм получения ключей К(i) описаны ниже.

На 16-й итерации получают последовательности R(16) и L(16) (без перестановки), которые конкатенируют в 64-битовую последовательность R(16)L(16).

Затем позиции битов этой последовательности переставляют в соответствии с матрицей IP-1 (табл.2).

Таблица 2: ***Матрица обратной перестановки IP***-1

40 08 48 16 56 24 64 32

39 07 47 15 55 23 63 31

38 06 46 14 54 22 62 30

37 05 45 13 53 21 61 29

36 04 44 12 52 20 60 28

35 03 43 11 51 19 59 27

34 02 42 10 50 18 58 26

33 01 41 09 49 17 57 25

Матрицы IP-1 и IP соотносятся следующим образом: значение 1-го элемента матрицы IP-1 равно 40, а значение 40-го элемента матрицы IP равно 1, значение 2-го элемента матрицы IP-1 равно 8, а значение 8-го элемента матрицы IP равно 2 и т.д.

Процесс расшифрования данных является инверсным по отношению к процессу шифрования. Все действия должны быть выполнены в обратном порядке. Это означает, что расшифровываемые данные сначала переставляются в соответствии с матрицей IP-1, а затем над последовательностью бит R(16)L(16) выполняются те же действия, что и в процессе шифрования, но в обратном порядке.

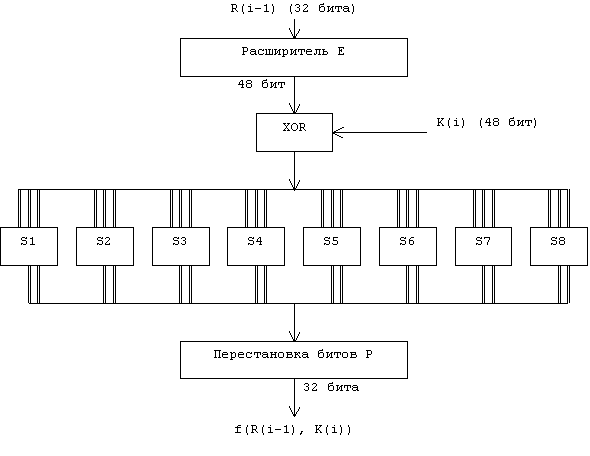
Итеративный процесс расшифрования может быть описан следующими формулами:

|  |
| --- |
| R(i-1) = L(i), i = 1, 2, ..., 16;  L(i-1) = R(i) xor f(L(i), K(i)), i = 1, 2, ..., 16 . |

На 16-й итерации получают последовательности L(0) и R(0), которые конкатенируют в 64-битовую последовательность L(0)R(0).

Затем позиции битов этой последовательности переставляют в соответствии с матрицей IP. Результат такой перестановки - исходная 64-битовая последовательность.

Теперь рассмотрим функцию шифрования f(R(i-1),K(i)). Схематически она показана на рис. 3.

  
Рис.3. Вычисление функции f(R(i-1), K(i))

Для вычисления значения функции f используются следующие функции-матрицы:

* Е - расширение 32-битовой последовательности до 48-битовой,
* S1, S2, ... , S8 - преобразование 6-битового блока в 4-битовый,
* Р - перестановка бит в 32-битовой последовательности.

Функция расширения Е определяется табл.3. В соответствии с этой таблицей первые 3 бита Е(R(i-1)) - это биты 32, 1 и 2, а последние - 31, 32 и 1.

Таблица 3:***Функция расширения E***

32 01 02 03 04 05

04 05 06 07 08 09

08 09 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 01

Результат функции Е(R(i-1)) есть 48-битовая последовательность, которая складывается по модулю 2 (операция xor) с 48-битовым ключом К(i). Получается 48-битовая последовательность, которая разбивается на восемь 6-битовых блоков B(1)B(2)B(3)B(4)B(5)B(6)B(7)B(8). То есть:

E(R(i-1)) xor K(i) = B(1)B(2)...B(8) .

Функции S1, S2, ... , S8 определяются табл.4.

|  |
| --- |
| Таблица 4  Функции преобразования S1, S2, ..., S8 |
| |  |  |  |  | | --- | --- | --- | --- | |  | | Номер столбца  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 |  | | Н  о  м  е  р  с  т  р  о  к  и | 0  1  2  3 | 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 | S1 | | 0  1  2  3 | 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 | S2 | | 0  1  2  3 | 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 | S3 | | 0  1  2  3 | 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 | S4 | | 0  1  2  3 | 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 | S5 | | 0  1  2  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 | S6 | | 0  1  2  3 | 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 | S7 | | 0  1  2  3 | 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 | S8 | |

К табл.4. требуются дополнительные пояснения. Пусть на вход функции-матрицы Sj поступает 6-битовый блок B(j) = b1b2b3b4b5b6, тогда двухбитовое число b1b6 указывает номер строки матрицы, а b2b3b4b5 - номер столбца. Результатом Sj(B(j)) будет 4-битовый элемент, расположенный на пересечении указанных строки и столбца.

Например, В(1)=011011. Тогда S1(В(1)) расположен на пересечении строки 1 и столбца 13. В столбце 13 строки 1 задано значение 5. Значит, S1(011011)=0101.

Применив операцию выбора к каждому из 6-битовых блоков B(1), B(2), ..., B(8), получаем 32-битовую последовательность S1(B(1))S2(B(2))S3(B(3))...S8(B(8)).

Наконец, для получения результата функции шифрования надо переставить биты этой последовательности. Для этого применяется функция перестановки P (табл.5). Во входной последовательности биты перестанавливаются так, чтобы бит 16 стал битом 1, а бит 7 - битом 2 и т.д.

Таблица 5:***Функция перестановки P***

16 07 20 21

29 12 28 17

01 15 23 26

05 18 31 10

02 08 24 14

32 27 03 09

19 13 30 06

22 11 04 25

Таким образом,

f(R(i-1), K(i)) = P(S1(B(1)),...S8(B(8)))

Чтобы завершить описание алгоритма шифрования данных, осталось привести алгоритм получения 48-битовых ключей К(i), i=1...16. На каждой итерации используется новое значение ключа K(i), которое вычисляется из начального ключа K. K представляет собой 64-битовый блок с восемью битами контроля по четности, расположенными в позициях 8,16,24,32,40,48,56,64.

Для удаления контрольных битов и перестановки остальных используется функция G первоначальной подготовки ключа (табл.6).

Таблица 6  
Матрица G первоначальной подготовки ключа

57 49 41 33 25 17 09

01 58 50 42 34 26 18

10 02 59 51 43 35 27

19 11 03 60 52 44 36

63 55 47 39 31 23 15

07 62 54 46 38 30 22

14 06 61 53 45 37 29

21 13 05 28 20 12 04

Результат преобразования G(K) разбивается на два 28-битовых блока C(0) и D(0), причем C(0) будет состоять из битов 57, 49, ..., 44, 36 ключа K, а D(0) будет состоять из битов 63, 55, ..., 12, 4 ключа K. После определения C(0) и D(0) рекурсивно определяются C(i) и D(i), i=1...16. Для этого применяют циклический сдвиг влево на один или два бита в зависимости от номера итерации, как показано в табл.7.

|  |
| --- |
| Таблица 7  Таблица сдвигов для вычисления ключа |
| |  |  | | --- | --- | | Номер итерации | Сдвиг (бит) | | 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16 | 1  1  2  2  2  2  2  2  1  2  2  2  2  2  2  1 | |

Полученное значение вновь "перемешивается" в соответствии с матрицей H (табл.8).

Таблица 8:***Матрица H завершающей обработки ключа***

14 17 11 24 01 05

03 28 15 06 21 10

23 19 12 04 26 08

16 07 27 20 13 02

41 52 31 37 47 55

30 40 51 45 33 48

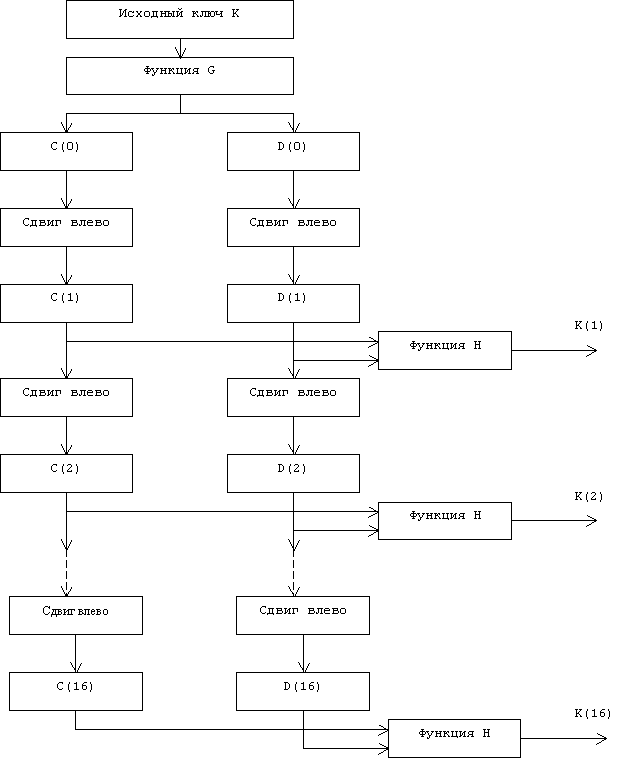
44 49 39 56 34 53

46 42 50 36 29 32

Ключ K(i) будет состоять из битов 14, 17, ..., 29, 32 последовательности C(i)D(i). Таким образом:

K(i) = H(C(i)D(i))

Блок-схема алгоритма вычисления ключа приведена на рис.

  
Рис. Блок-схема алгоритма вычисления ключа K(i)

Восстановление исходного текста осуществляется по этому алгоритму, но вначале вы используете ключ  
K(15), затем - K(14) и так далее. Теперь вам должно быть понятно, почему автор настойчиво рекомендует использовать приведенные матрицы. Если вы начнете самовольничать, вы, должно быть, получите очень секретный шифр, но вы сами не сможете его потом раскрыть!

**DES-CFB**

В этом режиме размер блока может отличаться от 64. Исходный файл M считывается последовательными t-битовыми блоками (t <= 64): M = M(1)M(2)...M(n) (остаток дописывается нулями или пробелами).

64-битовый сдвиговый регистр (входной блок) вначале содержит вектор инициализации IV, выравненный по правому краю. Для каждого сеанса шифрования используется новый IV.

Для всех i = 1...n блок шифртекста C(i) определяется следующим образом:

C(i) = M(i) xor P(i-1) ,

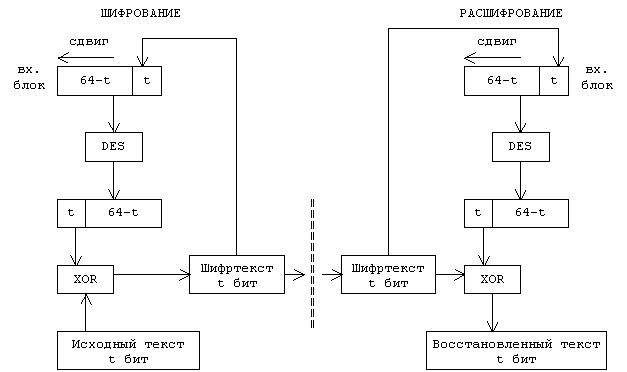
где P(i-1) - старшие t битов операции DES(С(i-1)), причем C(0)=IV.

Обновление сдвигового регистра осуществляется путем удаления его старших t битов и дописывания справа C(i).

Восстановление зашифрованных данных также не представляет труда: P(i-1) и C(i) вычисляются аналогичным образом и

M(i) = C(i) xor P(i-1) .

Блок-схема режима ***CFB*** приведена на рис.

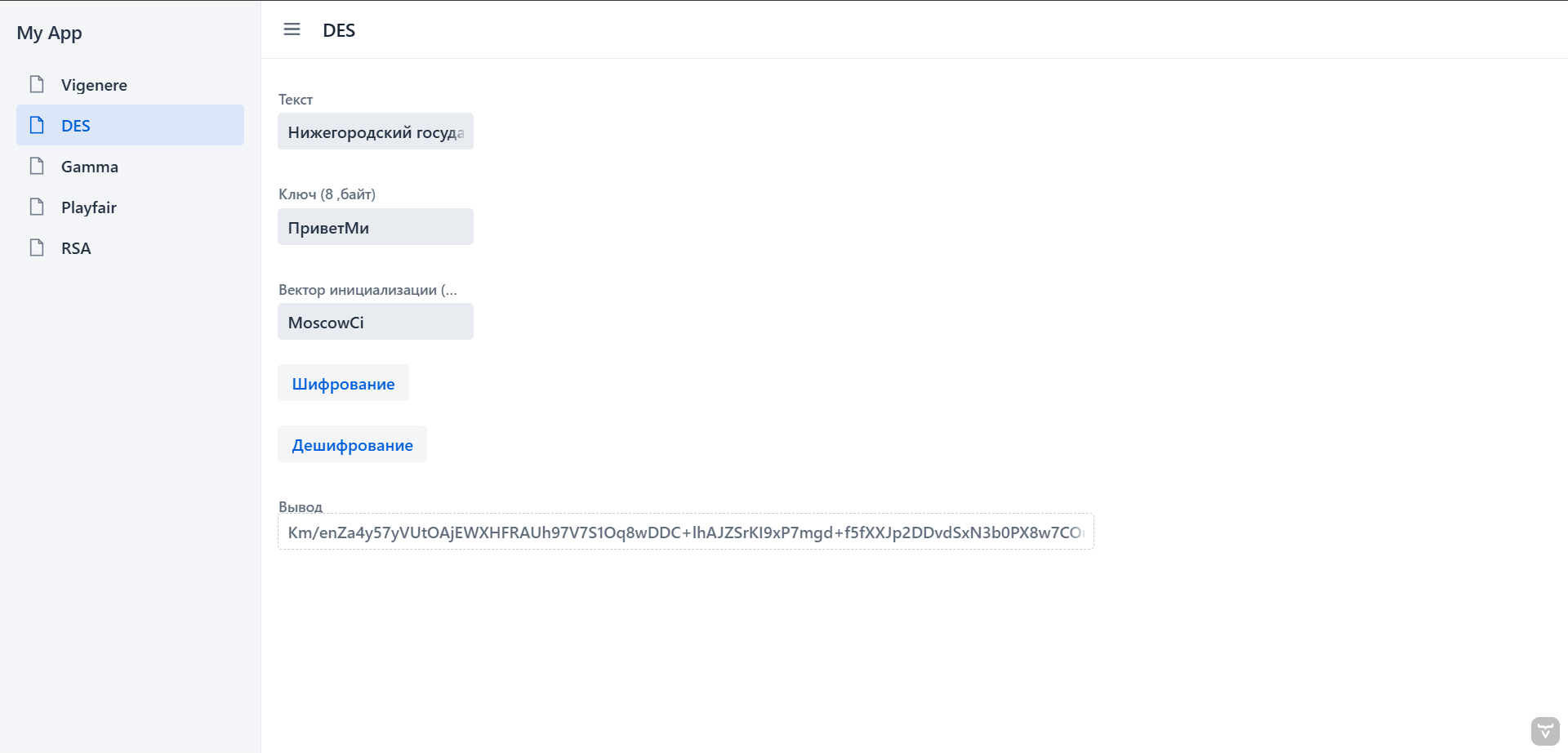
  
Рис. Работа алгоритма DES в режиме CFB

Примеры шифрования и расшифрования:

Нижегородский государственный технический университет

Шифрование:

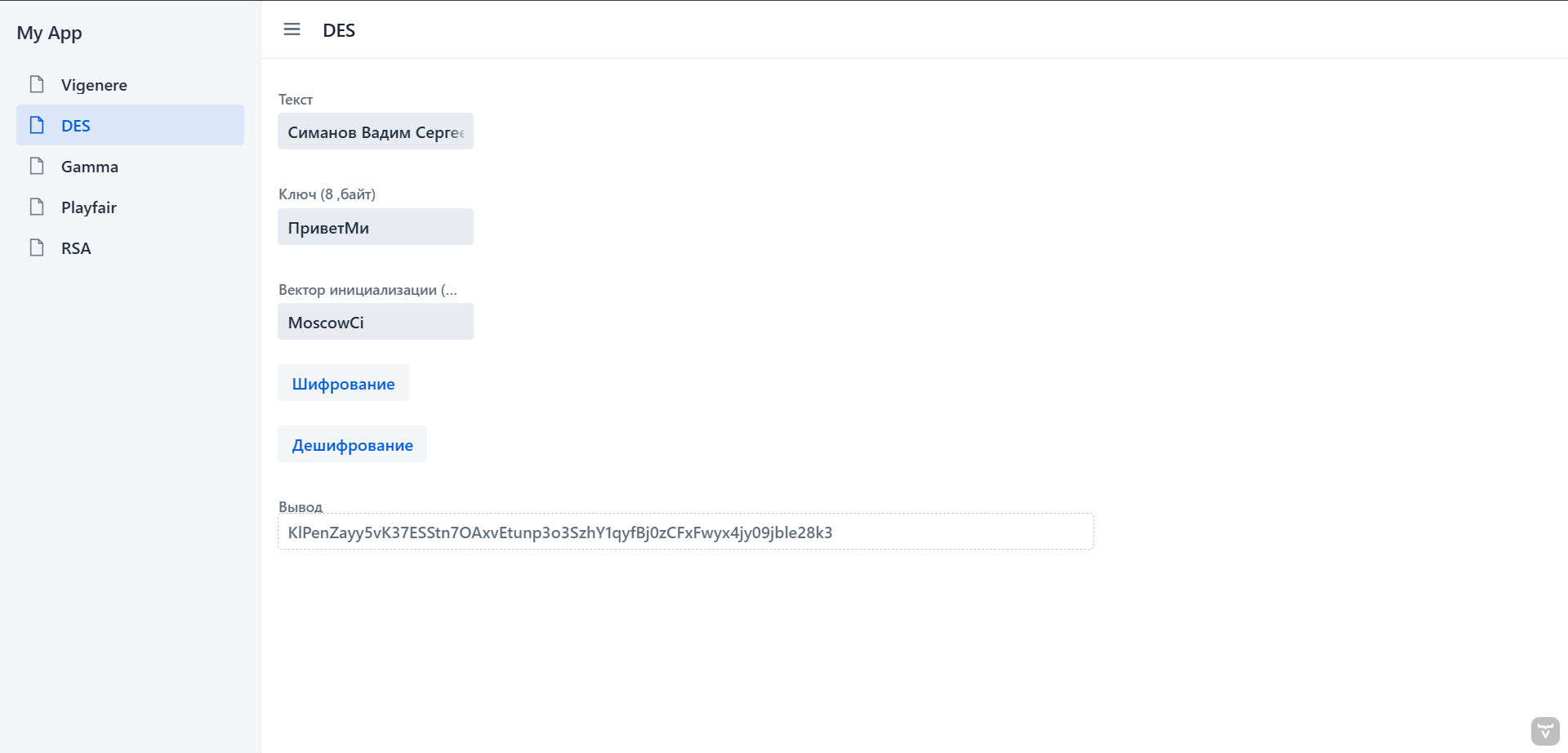
Нижегородский государственный технический университет:



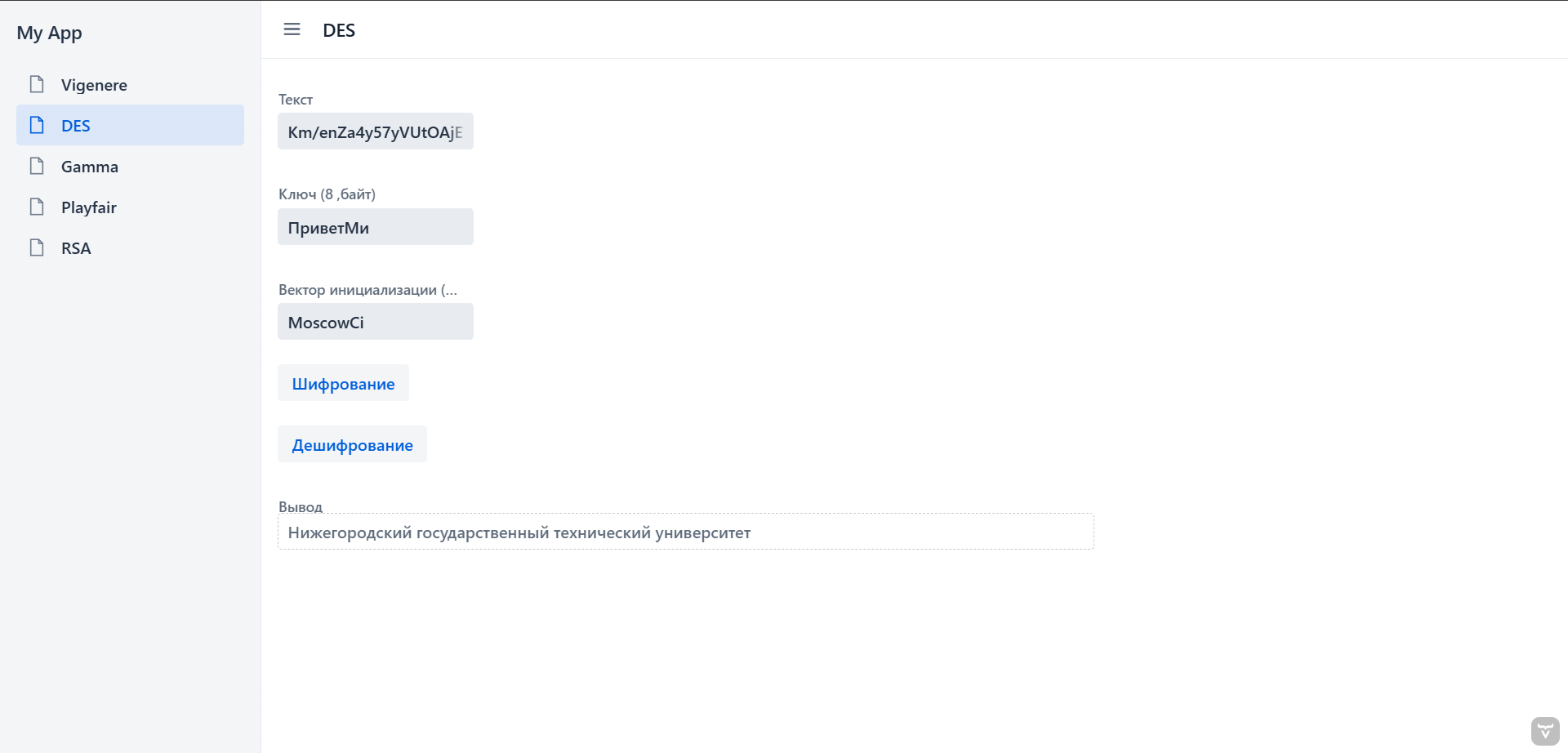
Полный вывод:

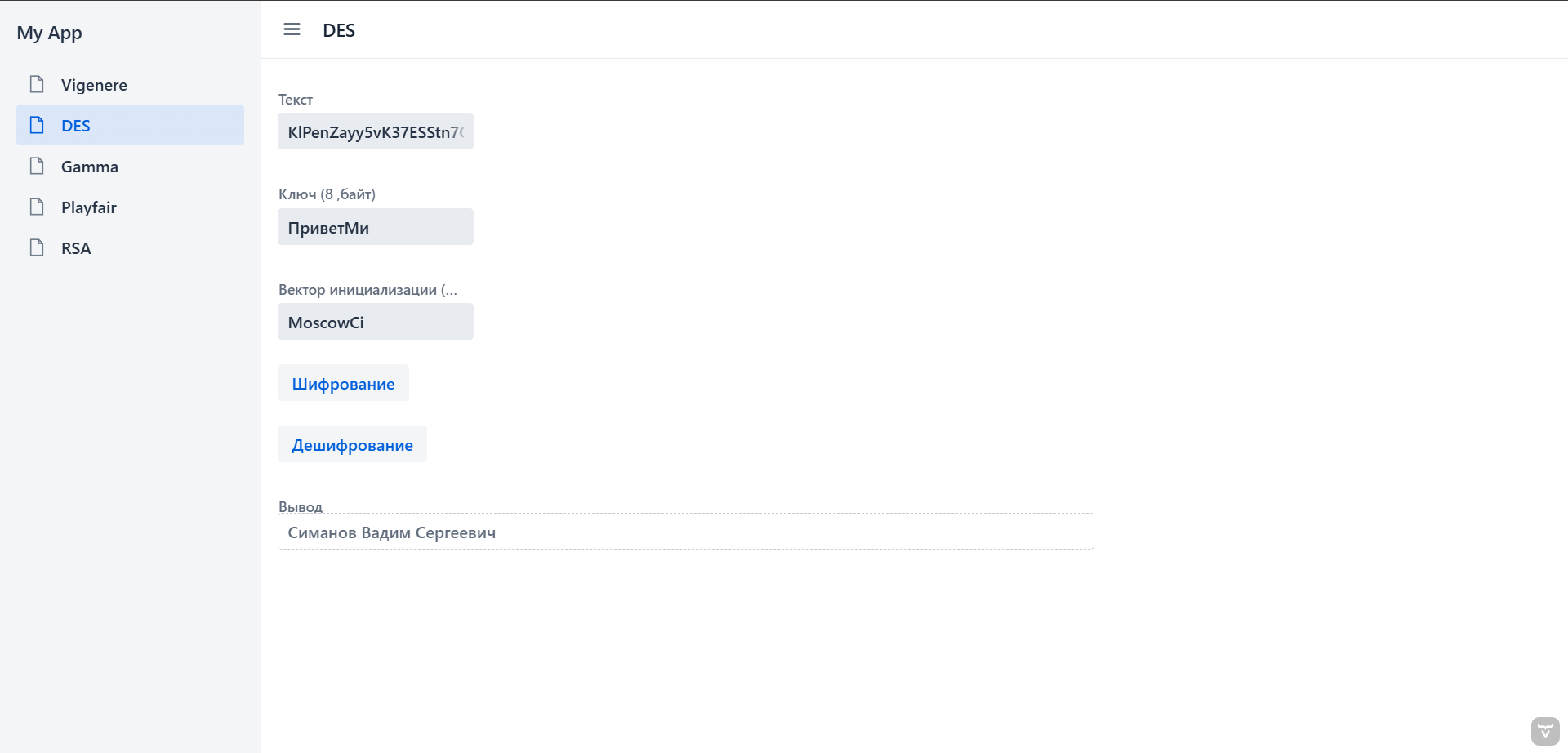
Km/enZa4y57yVUtOAjEWXHFRAUh97V7S1Oq8wDDC+lhAJZSrKI9xP7mgd+f5fXXJp2DDvdSxN3b0PX8w7COuG4eypA+GWjjDX1016jjAUKLClp2SiIFAowbKdrjQBaYb3UtCOpbjImg=

Симанов Вадим Сергеевич:



Расшифрование:





Текст программы с комментариями:

DesView – отвечает за отображении страницы

package com.example.application.views.lb;  
  
import com.vaadin.flow.component.button.Button;  
import com.vaadin.flow.component.notification.Notification;  
import com.vaadin.flow.component.orderedlayout.VerticalLayout;  
import com.vaadin.flow.component.textfield.TextField;  
import com.vaadin.flow.router.Menu;  
import com.vaadin.flow.router.PageTitle;  
import com.vaadin.flow.router.Route;  
  
import java.nio.charset.StandardCharsets;  
import java.util.Base64;  
  
@PageTitle("DES")  
@Menu(icon = "line-awesome/svg/file.svg", order = 0)  
@Route(value = "DES")  
public class DESView extends VerticalLayout {  
  
 private TextField inputField;  
 private TextField outputField;  
 private TextField keyField;  
 private TextField ivField;  
  
 public DESView() {  
 inputField = new TextField("Текст"); // Поле для ввода текста  
 keyField = new TextField("Ключ (8 ,байт)"); // Поле для ввода ключа  
 ivField = new TextField("Вектор инициализации (IV, 8 байт)"); // Поле для ввода IV  
 outputField = new TextField("Вывод"); // Поле для вывода результата  
 outputField.setWidth("800px");  
 outputField.setHeight("50px");  
 outputField.setReadOnly(true);  
  
 Button encryptButton = new Button("Шифрование", e -> encrypt()); // Кнопка шифрования  
 Button decryptButton = new Button("Дешифрование", e -> decrypt()); // Кнопка расшифрования  
  
 add(inputField, keyField, ivField, encryptButton, decryptButton, outputField);  
 }  
  
 private void encrypt() {  
 try {  
 // Получаем значения ключа, IV и входного текста из полей  
 String input = inputField.getValue();  
 String key = keyField.getValue();  
 String iv = ivField.getValue();  
  
 // Проверка длины ключа и IV (они должны быть по 8 байт)  
 if (key.length() != 8 || iv.length() != 8) {  
 Notification.*show*("Ключ и IV должны иметь длину ровно 8 байт.");  
 return;  
 }  
  
 if (input.length()==0) {  
 Notification.*show*("Заполните поле текста!");  
 return;  
 }  
  
 // Преобразуем строковые значения в байты  
 byte[] keyBytes = key.getBytes(StandardCharsets.*UTF\_8*);  
 byte[] ivBytes = iv.getBytes(StandardCharsets.*UTF\_8*);  
  
 // Создаем объект шифра DES в режиме CFB  
 cipherDESCFB cfb = new cipherDESCFB(keyBytes, ivBytes);  
  
 // Шифруем текст  
 byte[] encrypted = cfb.encrypt(input.getBytes(StandardCharsets.*UTF\_8*));  
  
 // Результат шифрования в base64 для корректного отображения  
 String encryptedBase64 = Base64.*getEncoder*().encodeToString(encrypted);  
 outputField.setValue(encryptedBase64);  
 } catch (Exception ex) {  
 Notification.*show*("Ошибка во время шифрования: " + ex.getMessage());  
 }  
 }  
  
 private void decrypt() {  
 try {  
 // Получаем значения ключа, IV и входного текста из полей  
 String input = inputField.getValue();  
 String key = keyField.getValue();  
 String iv = ivField.getValue();  
  
 // Проверка длины ключа и IV (они должны быть по 8 байт)  
 if (key.length() != 8 || iv.length() != 8) {  
 Notification.*show*("Ключ и IV должны иметь длину ровно 8 байт.");  
 return;  
 }  
  
 // Преобразуем строковые значения в байты  
 byte[] keyBytes = key.getBytes(StandardCharsets.*UTF\_8*);  
 byte[] ivBytes = iv.getBytes(StandardCharsets.*UTF\_8*);  
  
 // Декодируем зашифрованный текст из base64  
 byte[] encryptedBytes = Base64.*getDecoder*().decode(input);  
  
 // Создаем объект шифра DES в режиме CFB  
 cipherDESCFB cfb = new cipherDESCFB(keyBytes, ivBytes);  
  
 // Расшифровываем текст  
 byte[] decrypted = cfb.decrypt(encryptedBytes);  
  
 // Результат расшифровки  
 outputField.setValue(new String(decrypted, StandardCharsets.*UTF\_8*));  
 } catch (Exception ex) {  
 Notification.*show*("Ошибка при расшифровке: " + ex.getMessage());  
 }  
 }  
}

}

cipherDESCFB – метод реализующий алгоритм шифрования данных DES в режиме CFB

package com.example.application.views.lb;  
  
import java.util.Arrays;  
  
public class cipherDESCFB {  
  
 private DES des;  
 private byte[] iv; // Вектор инициализации (IV)  
 private static final int *BLOCK\_SIZE* = 8; // Размер блока DES (8 байт)  
  
 public cipherDESCFB(byte[] key, byte[] iv) {  
 this.des = new DES(key); // Инициализация DES с ключом  
 this.iv = iv; // Вектор инициализации  
 }  
  
 // Метод для добавления выравнивания PKCS5  
 private byte[] applyPadding(byte[] data) {  
 int paddingLength = *BLOCK\_SIZE* - (data.length % *BLOCK\_SIZE*); // Количество байт для выравнивания  
 byte[] paddedData = new byte[data.length + paddingLength];  
 System.*arraycopy*(data, 0, paddedData, 0, data.length);  
 for (int i = data.length; i < paddedData.length; i++) {  
 paddedData[i] = (byte) paddingLength; // Заполнение padding  
 }  
 return paddedData;  
 }  
  
 // Метод для удаления padding после расшифровки  
 private byte[] removePadding(byte[] data) {  
 int paddingLength = data[data.length - 1]; // Последний байт содержит количество padding байт  
 return Arrays.*copyOfRange*(data, 0, data.length - paddingLength);  
 }  
  
 // Шифрование с добавлением выравнивания  
 public byte[] encrypt(byte[] plaintext) {  
 // Добавляем padding к открытому тексту  
 byte[] paddedPlaintext = applyPadding(plaintext);  
  
 byte[] ciphertext = new byte[paddedPlaintext.length]; // Массив для зашифрованного текста  
 byte[] feedback = iv.clone(); // Копируем IV как начальный вектор обратной связи  
  
 // Проходим по каждому блоку текста  
 for (int i = 0; i < paddedPlaintext.length; i += *BLOCK\_SIZE*) {  
 // Извлекаем текущий блок открытого текста  
 byte[] block = Arrays.*copyOfRange*(paddedPlaintext, i, i + *BLOCK\_SIZE*);  
  
 // Шифруем текущий вектор обратной связи  
 byte[] encryptedFeedback = des.encrypt(feedback);  
  
 // XOR текущий блок текста с зашифрованным вектором обратной связи  
 byte[] cipherBlock = xor(block, encryptedFeedback);  
  
 // Копируем зашифрованный блок в результат  
 System.*arraycopy*(cipherBlock, 0, ciphertext, i, cipherBlock.length);  
  
 // ОБНОВЛЕНИЕ ВЕКТОРА ОБРАТНОЙ СВЯЗИ:  
 // Смещаем текущий вектор обратной связи и обновляем его новыми зашифрованными данными  
 System.*arraycopy*(feedback, *BLOCK\_SIZE*, feedback, 0, feedback.length - *BLOCK\_SIZE*); // Смещение  
 System.*arraycopy*(cipherBlock, 0, feedback, feedback.length - *BLOCK\_SIZE*, *BLOCK\_SIZE*); // Добавляем новые данные  
 }  
  
 return ciphertext; // Возвращаем зашифрованный текст  
 }  
  
 // Расшифровка с удалением padding  
 public byte[] decrypt(byte[] ciphertext) {  
 byte[] plaintext = new byte[ciphertext.length]; // Массив для расшифрованного текста  
 byte[] feedback = iv.clone(); // Начальный вектор обратной связи (IV)  
  
 // Проходим по каждому блоку зашифрованного текста  
 for (int i = 0; i < ciphertext.length; i += *BLOCK\_SIZE*) {  
 // Извлекаем текущий зашифрованный блок  
 byte[] block = Arrays.*copyOfRange*(ciphertext, i, i + *BLOCK\_SIZE*);  
  
 // Шифруем текущий вектор обратной связи  
 byte[] encryptedFeedback = des.encrypt(feedback);  
  
 // XOR текущий зашифрованный блок с зашифрованным вектором обратной связи для расшифровки  
 byte[] plainBlock = xor(block, encryptedFeedback);  
  
 // Копируем расшифрованный блок в результат  
 System.*arraycopy*(plainBlock, 0, plaintext, i, plainBlock.length);  
  
 // ОБНОВЛЕНИЕ ВЕКТОРА ОБРАТНОЙ СВЯЗИ:  
 // Смещаем текущий вектор обратной связи и добавляем новый зашифрованный блок  
 System.*arraycopy*(feedback, *BLOCK\_SIZE*, feedback, 0, feedback.length - *BLOCK\_SIZE*); // Смещение  
 System.*arraycopy*(block, 0, feedback, feedback.length - *BLOCK\_SIZE*, *BLOCK\_SIZE*); // Добавляем новые данные  
 }  
  
 // Удаляем padding после расшифровки  
 return removePadding(plaintext);  
 }  
  
 // Операция XOR для двух массивов байт  
 private byte[] xor(byte[] a, byte[] b) {  
 byte[] result = new byte[a.length];  
 for (int i = 0; i < a.length; i++) {  
 result[i] = (byte) (a[i] ^ b[i]); // XOR каждого байта  
 }  
 return result;  
 }  
}

DES – реализует алгоритм шифрования DES

package com.example.application.views.lb;  
  
import java.util.Arrays;  
  
public class DES {  
  
 // Таблица начальной перестановки (Initial Permutation)  
 private static final int[] *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  
 };  
  
 // Таблица конечной перестановки (Final Permutation)  
 private static final int[] *FP* = {  
 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  
 };  
  
 // Таблица расширения (Expansion E)  
 private static final int[] *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  
 };  
  
 // S-box таблицы  
 private static final int[][][] *SBOX* = {  
 { // S1  
 {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}  
 },  
 { // S2  
 {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}  
 },  
 { // S3  
 {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}  
 },  
 { // S4  
 {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}  
 },  
 { // S5  
 {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}  
 },  
 { // S6  
 {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}  
 },  
 { // S7  
 {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}  
 },  
 { // S8  
 {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}  
 }  
 };  
  
 // Таблица перестановки P  
 private static final int[] *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  
 };  
  
 // Таблица перестановки ключа PC1  
 private static final int[] *PC1* = {  
 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,  
 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  
 };  
  
 // Таблица перестановки ключа PC2  
 private static final int[] *PC2* = {  
 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  
 };  
  
 // Количество сдвигов на каждом раунде  
 private static final int[] *SHIFT* = {  
 1, 1, 2, 2, 2, 2, 2, 2,  
 1, 2, 2, 2, 2, 2, 2, 1  
 };  
  
 private byte[] key; // Основной ключ  
  
 public DES(byte[] key) {  
 this.key = key;  
 }  
  
 public byte[] encrypt(byte[] plaintext) {  
 byte[] permutedBlock = initialPermutation(plaintext);  
 byte[] encryptedBlock = processBlock(permutedBlock, true);  
 return finalPermutation(encryptedBlock);  
 }  
  
 public byte[] decrypt(byte[] ciphertext) {  
 byte[] permutedBlock = initialPermutation(ciphertext);  
 byte[] decryptedBlock = processBlock(permutedBlock, false);  
 return finalPermutation(decryptedBlock);  
 }  
  
 private byte[] processBlock(byte[] block, boolean encrypt) {  
 // Разделение на левую и правую половины  
 byte[] left = Arrays.*copyOfRange*(block, 0, block.length / 2);  
 byte[] right = Arrays.*copyOfRange*(block, block.length / 2, block.length);  
  
 // Основной цикл Feistel  
 for (int i = 0; i < 16; i++) {  
 byte[] temp = right.clone();  
 right = xor(left, feistelFunction(right, getRoundKey(i, encrypt)));  
 left = temp;  
 }  
  
 // Конкатенация правой и левой частей  
 return concat(right, left);  
 }  
  
 private byte[] initialPermutation(byte[] block) {  
 return permute(block, *IP*);  
 }  
  
 private byte[] finalPermutation(byte[] block) {  
 return permute(block, *FP*);  
 }  
  
 private byte[] feistelFunction(byte[] right, byte[] roundKey) {  
 // Этап расширения E  
 byte[] expandedRight = permute(right, *E*);  
  
 // XOR с раундовым ключом  
 byte[] xored = xor(expandedRight, roundKey);  
  
 // S-box замены и перестановка P  
 byte[] substituted = sBoxSubstitution(xored);  
 return permute(substituted, *P*);  
 }  
  
 private byte[] sBoxSubstitution(byte[] input) {  
 byte[] output = new byte[4]; // Результат после всех S-боксов  
 for (int i = 0; i < 8; i++) {  
 byte[] sixBits = getBits(input, i \* 6, 6); // 6 бит на каждый S-бокс  
 int row = ((sixBits[0] << 1) | sixBits[5]); // 1 и 6 бит для строки  
 int col = ((sixBits[1] << 3) | (sixBits[2] << 2) | (sixBits[3] << 1) | sixBits[4]); // 2-5 биты для столбца  
 int sboxValue = *SBOX*[i][row][col];  
 setBits(output, i \* 4, 4, sboxValue); // Установка 4 бит результата  
 }  
 return output;  
 }  
  
 private byte[] getRoundKey(int round, boolean encrypt) {  
 // Генерация раундовых ключей на основе PC1, PC2 и сдвигов  
 return new byte[6]; // Пример, заменить реальной логикой  
 }  
  
 private byte[] xor(byte[] a, byte[] b) {  
 byte[] result = new byte[a.length];  
 for (int i = 0; i < a.length; i++) {  
 result[i] = (byte) (a[i] ^ b[i]);  
 }  
 return result;  
 }  
  
 private byte[] permute(byte[] input, int[] table) {  
 byte[] output = new byte[table.length / 8];  
 for (int i = 0; i < table.length; i++) {  
 int bitIndex = table[i] - 1;  
 int bytePos = bitIndex / 8;  
 int bitPos = bitIndex % 8;  
 int bit = (input[bytePos] >> (7 - bitPos)) & 1;  
 output[i / 8] |= bit << (7 - (i % 8));  
 }  
 return output;  
 }  
  
 private byte[] concat(byte[] a, byte[] b) {  
 byte[] result = new byte[a.length + b.length];  
 System.*arraycopy*(a, 0, result, 0, a.length);  
 System.*arraycopy*(b, 0, result, a.length, b.length);  
 return result;  
 }  
  
 private byte[] getBits(byte[] input, int start, int length) {  
 byte[] bits = new byte[length];  
 for (int i = 0; i < length; i++) {  
 int bytePos = (start + i) / 8;  
 int bitPos = (start + i) % 8;  
 bits[i] = (byte) ((input[bytePos] >> (7 - bitPos)) & 1);  
 }  
 return bits;  
 }  
  
 private void setBits(byte[] output, int start, int length, int value) {  
 for (int i = 0; i < length; i++) {  
 int bit = (value >> (length - 1 - i)) & 1;  
 int bytePos = (start + i) / 8;  
 int bitPos = (start + i) % 8;  
 output[bytePos] |= bit << (7 - bitPos);  
 }  
 }  
}

Application – запуск программы

package com.example.application;  
  
import com.vaadin.flow.component.page.AppShellConfigurator;  
import com.vaadin.flow.theme.Theme;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
@Theme(value = "my-app")  
public class Application implements AppShellConfigurator {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
  
}

Вывод: мною был реализован алгоритм шифрования данных DES в режиме CFB