МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт ИИ

Кафедра «Информационная безопасность» (БК №252)

Курсовая работа

по дисциплине: Методы программирования

тема: Алгоритм шифрования «DES» в режиме гаммирования с обратной связью по выходу

Студент группы ККСО-02-20: Шинкарев М. С.

Руководитель работы: Чуваев А. В.

Работа представлена к защите: «___» _____ 2022 г.

Допущена к защите: «___» _____ 2022 г.

Оценка: «_____»

Москва 2022

Оглавление

Введение

Быстро развивающиеся компьютерные информационные технологии вносят заметные изменения в нашу жизнь. Всё чаще понятие «информация» используется как обозначение специального товара, который можно приобрести, продать, обменять на что-то другое и т.п. При этом стоимость информации превосходит стоимость компьютерной системы, в которой она находится. Поэтому вполне естественно возникает потребность в защите информации от несанкционированного доступа, умышленного изменения, кражи, уничтожения и других преступных действий.

В последние десятилетия, когда человечество вступило в стадию информационного общества, криптография (наука о защите информации) стала использоваться очень широко, обслуживая, в первую очередь, потребности бизнеса. Причем имеются ввиду не только межбанковские расчеты по компьютерным сетям, или, скажем, биржи, в которых все расчеты проводят через интернет, но и многочисленные операции, в которых ежедневно участвуют миллионы, если не миллиарды «обычных» людей, а именно: расчеты по кредитным карточкам, перевод заработной платы в банк, заказ билетов через Интернет, покупки в Интернет-магазинах и т.д. Естественно, все эти операции, как и, скажем, разговоры по мобильным телефонам и электронная почта должны быть защищены от нечестных или просто чрезмерно любопытных людей и организаций.

На сегодняшний день благодаря повсеместному применению открытых сетей передачи данных, таких как Internet, и построенных на их основе сетей intranet и extranet, криптографические протоколы находят все более широкое применение для решения разнообразного круга задач и обеспечения постоянно расширяющихся услуг, предоставляемых пользователям таких сетей.

1. Теоретические сведения для реализации алгоритма DES

1.1 Сети Фейстеля

Сеть Фейстеля (конструкция Фейстеля) — один из методов построения блочных шифров. Сеть представляет собой определённую многократно повторяющуюся (итерированную) структуру, называющуюся ячейкой Фейстеля [6]. При переходе от одной ячейки к другой меняется ключ, причём выбор ключа зависит от конкретного алгоритма. Операции шифрования и дешифрования на каждом этапе очень просты, и при определённой доработке совпадают, требуя только обратного порядка используемых ключей. Шифрование при помощи данной конструкции легко реализуется как на программном уровне, так и на аппаратном, что обеспечивает широкие возможности применения. Большинство современных блочных шифров используют сеть Фейстеля в качестве основы.

1.1.1 Шифрование

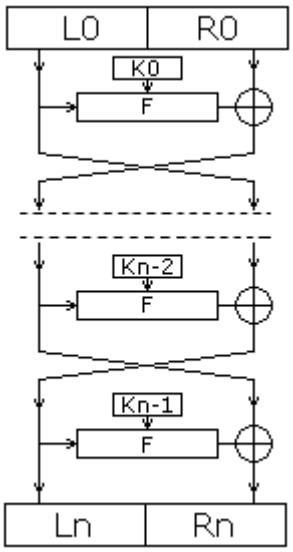

Рисунок 1.1 Шифрование

Рассмотрим случай, когда мы хотим зашифровать некоторую информацию, представленную в двоичном виде в компьютерной памяти

(например, файл) или электронике, как последовательность нулей и единиц.

Вся информация разбивается на блоки фиксированной длины. В случае, если длина входного блока меньше, чем размер, который шифруется заданным алгоритмом, то блок удлиняется каким-либо способом. Как правило длина блока является степенью двойки, например: 64 бита, 128 бит. Далее будем рассматривать операции, происходящие только с одним блоком, так как с другими в процессе шифрования выполняются те же самые операции.
Выбранный блок делится на два равных подблока — «левый» (L_0) и «правый» (R_0).
«Левый подблок» L_0 видоизменяется функцией $f(L_0, K_0)$ в зависимости от раундового ключа K_0 , после чего он складывается по модулю 2 с «правым подблоком» R_0 .
Результат сложения присваивается новому левому подблоку L_1 , который будет половиной входных данных для следующего раунда, а «левый подблок» L_0 присваивается без изменений новому правому подблоку R_1 (рисунок 1.1), который будет другой половиной.
После чего операция повторяется N-1 раз, при этом при переходе от одного этапа к другому меняются раундовые ключи (K_0 на K_1 и т. д.) по какому-либо математическому правилу, где N — количество раундов в заданном алгоритме.

1.1.2 Дешифрование

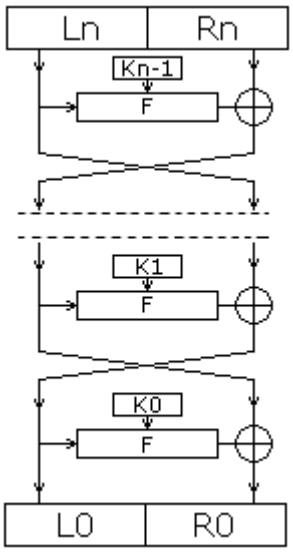

Рисунок 1.2 Дешифрование

Дешифровка информации происходит так же, как и шифрование, с тем лишь исключением, что ключи идут в обратном порядке, то есть не от первого к N-ному, а от N-го к первому [6].

1.2 Режимы использования

Для DES рекомендовано несколько режимов использования [2]:

1. ECB (Electronic Code Book) – режим «электронной кодовой книги» (простая замена);
2. CBC (Cipher Block Chaining) – режим сцепления блоков;
3. CFB (Cipher Feed Back) – режим обратной связи по шифротексту;
4. OFB (Output Feed Back) – режим обратной связи по выходу;
5. Counter Mode – режим счетчика.

2. Описание выбранного алгоритма и режима шифрования

2.1 Алгоритм шифрования DES (Data Encryption Standard)

DES (Data Encryption Standard) – симметричный алгоритм шифрования, разработанный совместными силами IBM, NBS и АНБ в 1977 году и утвержденный правительством США как официальный стандарт. DES шифрует информацию блоками по 64 бита с помощью 64-битного ключа шифрования.

2.1.1 Общая схема шифрования

Процесс шифрования заключается в начальной перестановке битов 64-битового блока, шестнадцати циклах шифрования и обратной перестановки битов (рисунок 2.1). Все перестановки и коды в таблицах подобраны разработчиками таким образом, чтобы максимально затруднить процесс дешифровки путем подбора ключа [3]. Структура алгоритма DES приведена на рисунке 2.2.

Рисунок 2.1 – Обобщенная схема шифрования

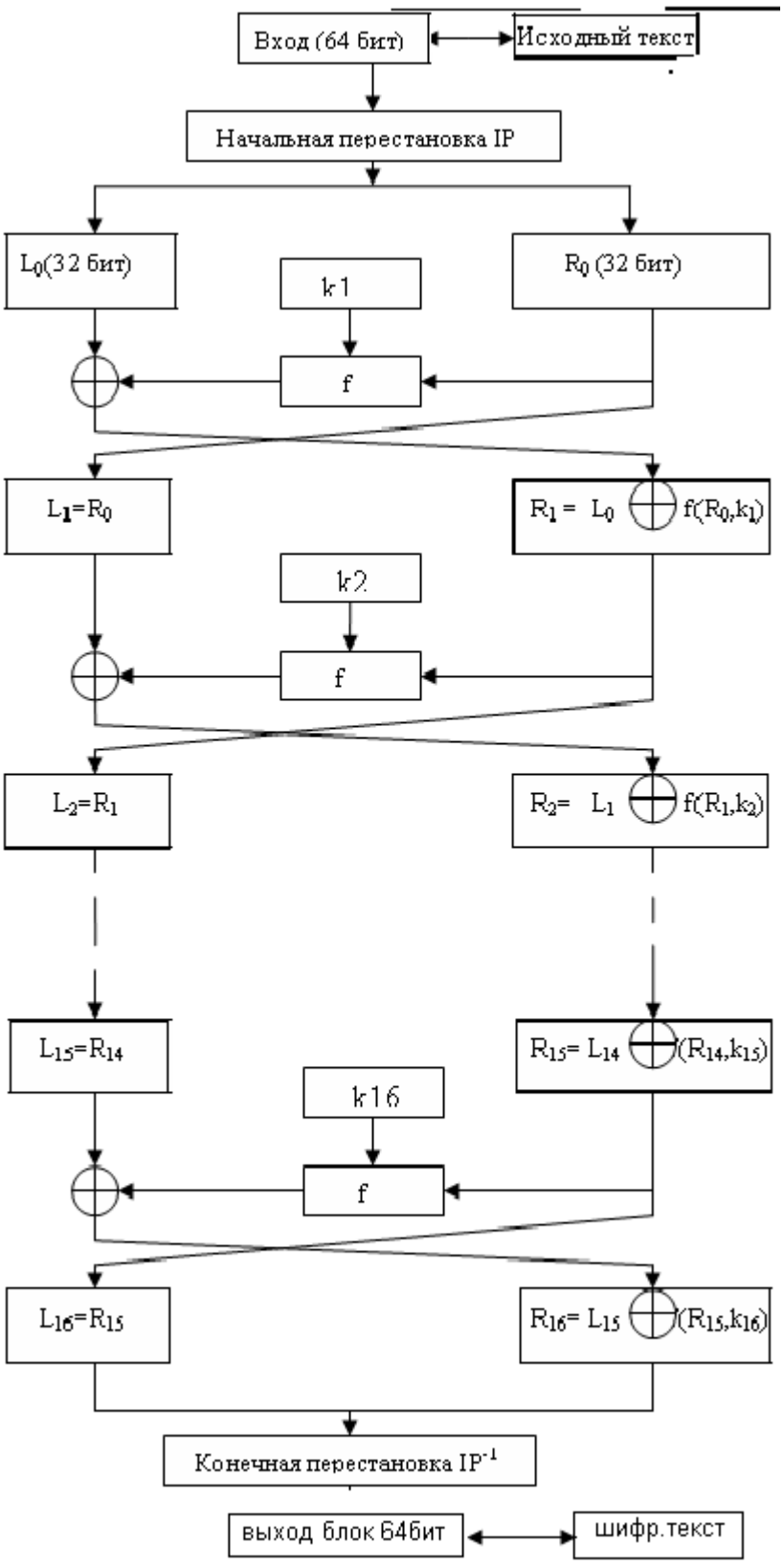

Рисунок 2.2 – Структура алгоритма шифрования DES

Исходный текст T (блок 64 бит) преобразуется с помощью начальной перестановки IP, которая определяется таблицей 2.1, следующим образом: бит 58 блока T становится битом 1, бит 50 – битом 2 и т.д.

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 2.1 – Начальная перестановка IP

Полученный после начальной перестановки 64-битовый блок IP(T), участвует в 16 циклах преобразования Фейстеля. Он разделяется на две части L_0 и R_0 по 32 бита каждая так, что IP(T) = L_0R_0.

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

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$$

В 16-циклах преобразования Фейстеля функция f играет роль шифрования. Функцию f рассмотрим в дальнейшем.

На 16-й итерации получаем последовательность L_{16} и R_{16}, которые конкатенируют в 64-битовую последовательность $L_{16}R_{16}$. Затем позиции битов этой последовательности переставляются в соответствии с перестановкой IP^{-1}, представленной на таблице 2.2

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 2.2 – Обратная перестановка IP^{-1}

2.1.2 Функция f

Аргументами функции f являются 32-битовая последовательность R_{i-1} и 48-битовый ключ k_i, который является результатом преобразования 56-битового исходного ключа шифра k [4]. Для вычисления функции f последовательно используются:

1. функция расширения E.
2. сложение по модулю 2 с ключом k_i
3. преобразование S, состоящее из 8 преобразований S-блоков S_1, S_2, …, S_8.
4. перестановка P.

Функция E расширяет 32-битовый вектор R_{i-1} до 48-битового вектора $E(R_{i-1})$ путём дублирования некоторых битов из R_{i-1}. Порядок битов вектора $E(R_{i-1})$ указан в таблице 2.3

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

Таблица 2.3 – Функция расширения E

Полученный после перестановки блок $E(R_{i-1})$ складывается по модулю 2 с ключами k_i и затем представляется в виде восьми последовательных блоков B1, B2, …, B8

$$E(R_{i-1}) \oplus k_i = B_1 B_2 \dots B_8$$

Каждый блок B_j является 6-битовым блоком. Далее каждый из блоков преобразуется в 4-битовый блок с помощью соответствующего узла замены S_j (рисунок 2.3) следующим образом: рассмотрим 6-битовый вход S-блока: b_1, b_2, b_3, b_4, b_5, b_6. Биты b_1 и b_6 объединяются, образую 2-битовое число от 0 до 3, соответствующее строке таблицы. Средние 4 бита, с b_2 по b_5, объединяются, образуя 4-битовое число от 0 до 15, соответствующее столбцу таблицы [5].

В результате этих преобразований получается восемь 4-битовых блоков, которые объединяются в единый 32-битовый блок. Этот блок вновь изменяется в соответствии с перестановкой P (таблица 2.4).

	Номер столбца																
Номер строки	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	S_1
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0	
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13	
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	S_2
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15	
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9	
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8	S_3
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7	
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12	
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15	S_4
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4	
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14	
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9	S_5
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14	
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3	
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11	S_6
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6	
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13	
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1	S_7
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2	
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12	
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7	S_8
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8	
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11	

Рисунок 2.3 – S-блоки

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 2.4 – Перестановка P

Результат последней операции и является выходным значением функции шифрования f.

2.1.3 Генерация подключей

Сначала 64-битовый ключ уменьшается до 56-битового ключа отбрасыванием каждого 8-го бита. Эти биты используются только для контроля четности, позволяя проверять правильность ключа [3]. После извлечения 56-битового ключа для каждого из 16 преобразований Фейстеля генерируется новый 48-битовый подключ. Эти подключи k_i, определяются следующим образом.

Сначала изначальный ключ преобразуется согласно перестановки PC1 (таблица 2.5).

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 2.5 – Перестановка PC1

Результат перестановки делится на две 28-битовые половины и в зависимости от номера итерации обе части циклически сдвигаются на 1 или 2 бита согласно таблице 2.6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Таблица 2.6 – Число битов сдвига ключа в зависимости от этапа

После сдвига выбираются 48 из 56 битов с помощью перестановки PC2 (таблица 2.7), которые и формируют очередной подключ.

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

Таблица 2.7 – Перестановка PC2

2.1.4 Алгоритм дешифрования

Процедуры шифрования и дешифрования полностью идентичны, за исключением порядка использования ключевых элементов k_i. При дешифровании данных все действия выполняются в обратном порядке. В 16 циклах дешифрования, в отличие от шифрования с помощью прямого преобразования сетью Фейстеля, здесь используется обратное преобразование сетью Фейстеля.

$$R_{i-1} = L_i$$
$$L_{i-1} = R_i \oplus f(L_i, k_i)$$

Структура алгоритма дешифрования указана на рисунке 2.4

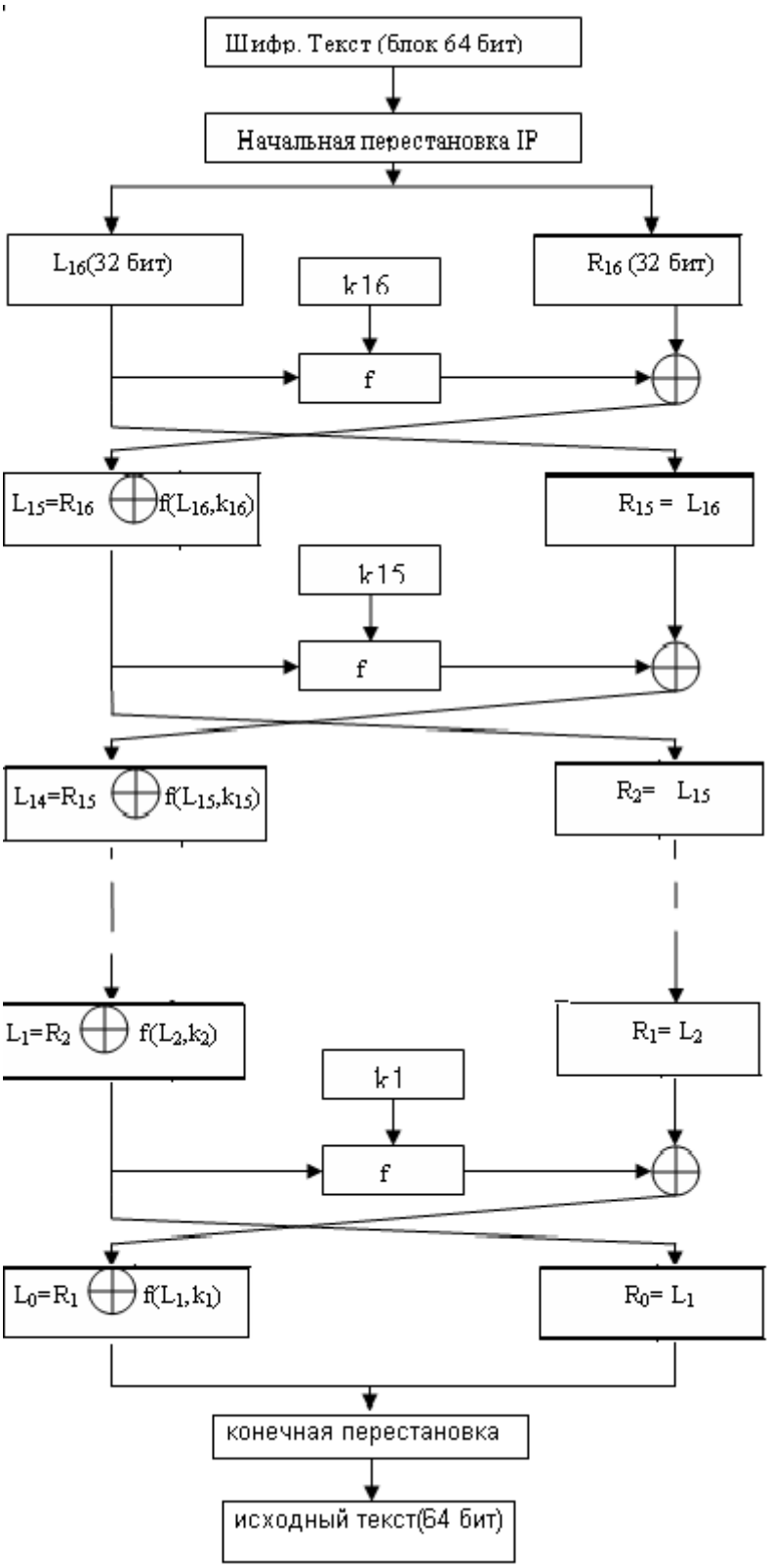

Рисунок 2.4 – Структура алгоритма дешифрования DES

2.2 Режим гаммирования с обратной связью по выходу (OFB)

OFB – один из вариантов использования симметричного блочного шифра. Особенностью режима является то, что в качестве входных данных для алгоритма блочного шифрования не используется само сообщение. Вместо этого блочный шифр используется для генерации псевдослучайного потока байтов, который с помощью операции XOR складывается с блоками открытого текста [2].

Схема шифрования в режиме OFB определяется следующим образом:

$K_0 = IV$

$K_i = E(K, K_{i-1})$ для $i = 1, \dots, n$

$C_i = K_i \oplus P_i$

Где, IV – вектор инициализации (синхропосылка), K_i – ключевой поток, K – ключ шифрования, E – функция шифрования блока (в нашем случае DES), n – количество блоково открытого текста в сообщении, C_i и P_i – блоки шифрованного и открытого текстов соответственно.

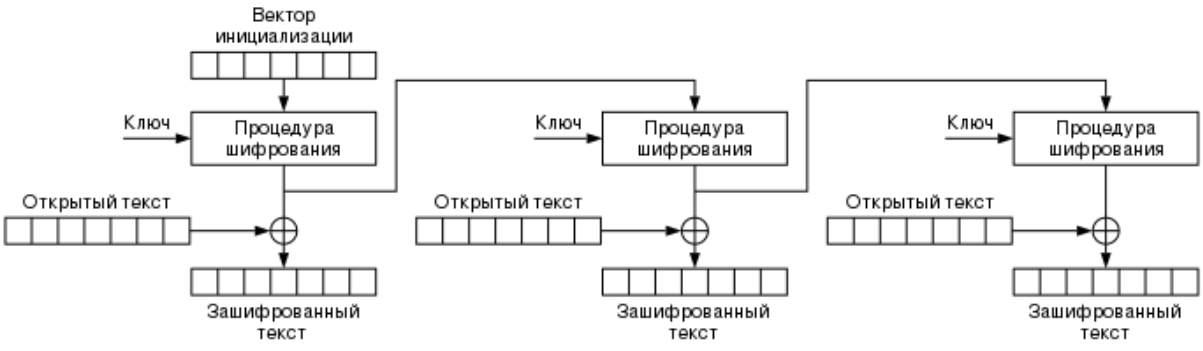

Рисунок 2.5 – Схема шифрования в режиме обратной связи по выходу

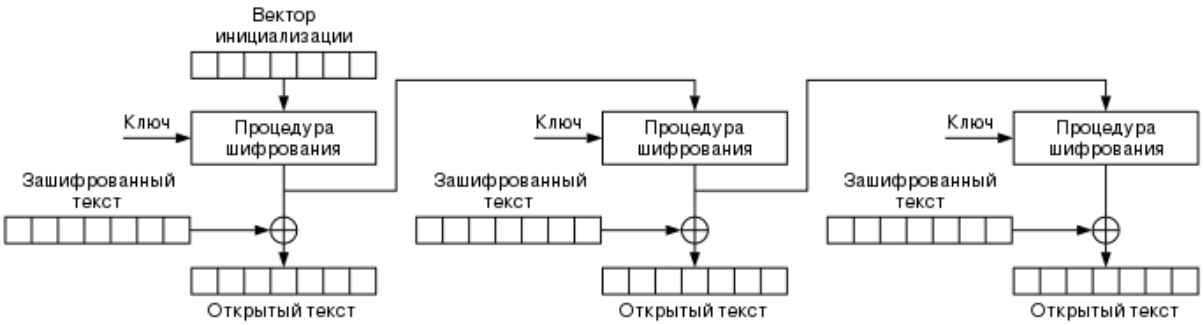

Рисунок 2.6 – Схема дешифрования в режиме обратной связи по выходу

Алгоритм дешифрования в режиме OFB полностью совпадает с алгоритмом шифрования. Функция дешифрования блочного алгоритма не используется в данном режиме, т.к. ключевой поток генерируется только функцией шифрования блока [5].

3. Описание программной реализации

Программа реализована на языке Java. Минимальная необходимая версия jdk = 17. В коде используются только стандартные библиотеки языка. В зависимости от введенных аргументов командной строки программа может работать в двух режимах: режиме тестирования и режиме шифрования.

Режим тестирования запускается при одном единственном аргументе – команде «test». После запуска этого режима происходит проверка режима работы OFB алгоритма DES на контрольных векторах из американских нормативных документов. Результатом работы является отчет о прохождении тестов.

Для запуска режима шифрования первым аргументов вводится пароль, вторым – путь до файла с синхропосылкой, третьим – путь/пути до файлов и/или папок, которые нужно шифровать. В результате работы программы будет произведена перезапись файлов их зашифрованной/дешифрованной версией.

3.1 Класс Main

Главный класс программы. Связывает все остальные классы программы. Организовывает поблочную перезапись (считывание и запись) в один файл.

3.2 Класс DES

Данный класс реализовывает алгоритм шифрования DES с режимом работы OFB. Также в классе реализован метод для автоматической проверки работоспособности класса на основе контрольных значений из американских официальных документов.

3.3 Класс Utils

Обрабатывает введенные аргументы, преобразовывает пароль в ключ при помощи хэш-функции «SHA-256», извлекает, либо генерирует синхропосылку в файле по указанному пути.

Заключение

В ходе работы были получены основные принципы блочного шифрования и их режимы работы. В результате получена работоспособная программа, реализующая алгоритм шифрования DES в режиме OFB (Output Feed Back) на языке программирования Java.

В данной программе поддерживаются следующие функции:

1. Проверка алгоритма по контрольным значениям;
2. Преобразование пароля в ключ;
3. Генерация случайной синхропосылки (IV);
4. Поддержка работы с набором и деревом файлов.

Список литературы

1. https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf
2. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf
3. Б.Шнайер — Прикладная криптография. Протоколы, алгоритмы и исходные тексты на языке С
4. https://ru.wikipedia.org/wiki/DES
5. Панасенко С.П. Алгоритмы шифрования. Специальный справочник. — СПб.: БХВПетербург, 2009. 506 с.: ил.
6. https://infopedia.su/19x4fd0.html
7. https://habr.com/ru/post/140404/
8. http://websites.umich.edu/~x509/ssleay/fip81/fip81.html#f4

Приложения

Класс DES

```
public class DES {

    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
    };

    private static final int[] IP1 = {
            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
    };

    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
    };

    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
    };

    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
    };

    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
    };
```

```
private static final int[][][] S_BOX = {
        {
                {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}
        }
};
private static final int[] shiftBits = {
        1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
};

private static String[] keys;
private static String OFBInputBlock;

private static String hexToBin(String input) {
    int n = input.length() * 4;

    input = Long.toBinaryString(Long.parseUnsignedLong(input, 16));
```

```
        while (input.length() < n)
            input = "0".concat(input);

        return input;
    }

    private static String binToHex(String input) {
        int n = input.length() / 4;

        input = Long.toHexString(Long.parseUnsignedLong(input, 2));

        while (input.length() < n)
            input = "0".concat(input);

        return input;
    }

    private static String permutation(int[] sequence, String input) {
        String output = "";
        input = hexToBin(input);

        for (int j : sequence)
            output = output.concat(String.valueOf(input.charAt(j - 1)));

        output = binToHex(output);

        return output;
    }


    private static String xor(String a, String b) {
        long t_a = Long.parseUnsignedLong(a, 16);
        long t_b = Long.parseUnsignedLong(b, 16);

        t_a = t_a ^ t_b;
        a = Long.toHexString(t_a);

        while (a.length() < b.length())
            a = "0".concat(a);

        return a;
    }

    private static String leftCircularShift(String input, int numBits) {
        int n = input.length() * 4;
        int[] perm = new int[n];

        for (int i = 0; i < n - 1; i++)
            perm[i] = (i + 2);

        perm[n - 1] = 1;

        while (numBits-- > 0)
            input = permutation(perm, input);

        return input;
    }

    public static void setKeys(String key) {
        keys = new String[16];

        key = permutation(PC1, key);
```

```
        for (int i = 0; i < 16; i++) {
            key = leftCircularShift(key.substring(0, 7), shiftBits[i])
                    + leftCircularShift(key.substring(7, 14), shiftBits[i]);

            keys[i] = permutation(PC2, key);
        }
    }

    private static String sBox(String input) {
        String output = "";
        input = hexToBin(input);

        for (int i = 0; i < 48; i += 6) {
            String temp = input.substring(i, i + 6);

            int num = i / 6;
            int row = Integer.parseInt(temp.charAt(0) + "" + temp.charAt(5),
2);
            int col = Integer.parseInt(temp.substring(1, 5), 2);

            output =
output.concat(Integer.toHexString(S_BOX[num][row][col]));
        }

        return output;
    }

    private static String round(String input, String key) {
        // fk
        String left = input.substring(0, 8);
        String temp = input.substring(8, 16);
        String right = temp;

        temp = permutation(E, temp);
        temp = xor(temp, key);
        temp = sBox(temp);
        temp = permutation(P, temp);
        left = xor(left, temp);

        return right + left;
    }

    private static String encrypt(String plainText) {
        int i;

        plainText = permutation(IP, plainText);

        for (i = 0; i < 16; i++) {
            plainText = round(plainText, keys[i]);
        }

        plainText = plainText.substring(8, 16) + plainText.substring(0, 8);

        plainText = permutation(IP1, plainText);

        return plainText;
    }

    private static String decrypt(String cipherText) {
        int i;

        cipherText = permutation(IP, cipherText);
```

```
        for (i = 15; i > -1; i--) {
            cipherText = round(cipherText, keys[i]);
        }

        cipherText = cipherText.substring(8, 16) + cipherText.substring(0,
8);
        cipherText = permutation(IP1, cipherText);

        return cipherText;
    }

    public static void setInitVector(String initVector) {
        OFBInputBlock = initVector;
    }

    public static String encryptDecryptOFB(String text) {
        OFBInputBlock = encrypt(OFBInputBlock);

        return xor(text, OFBInputBlock);
    }

    public static void runTests() {

        int failedCounter = 0;

        String keyForOFB = "0123456789abcdef";
        String initVectorForOFB = "1234567890abcdef";
        String[][] testsForOFB = {
                {"5765207468652070", "ea03351dc6e26e55"},
                {"656f706c65206f66", "38f81a3c22a63779"},
                {"2074686520556e69", "7b7641a66463fa8a"},
                {"7465642053746174", "0c9d1ead3ed113d7"},
                {"65732c20696e206f", "0608a565602f23c4"},
                {"7264657220746f20", "21c8836e82c5f07b"}
        };

        System.out.println("\nTests for OFB mode:");
        setKeys(keyForOFB);
        setInitVector(initVectorForOFB);
        for (int i = 0; i < testsForOFB.length; i++) {
            String text = testsForOFB[i][0];

            text = encryptDecryptOFB(text);
            if (!text.equals(testsForOFB[i][1])) {
                System.out.println("\tTest " + (i + 1) + " failed!"
                        + "\n\t\tKey:         " + keyForOFB
                        + "\n\t\tIV:          " + initVectorForOFB
                        + "\n\t\tClear:       " + testsForOFB[i][0]
                        + "\n\t\tCipher:      " + testsForOFB[i][1]
                        + "\n\t\tEnc result: " + text);
                failedCounter++;

                continue;
            }

            System.out.println("\tTest " + (i + 1) + " passed!");
        }

        System.out.println();
        if (failedCounter == 0) {
            System.out.println("All tests passed!");
        } else {
            System.out.println(failedCounter + " tests failed!");
```

```
        }
    }
}
```

Класс Utils

```
import java.io.File;
import java.nio.file.Files;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.HexFormat;

public class Utils {

    private static String password;
    private static File initVectorFile;
    private static ArrayList<File> files;

    public static boolean processInput(String[] args) throws Exception {
        if (!(args != null && (args.length == 1 && "test".equals(args[0]) ||
args.length >= 3))) {
            throw new Exception("Wrong arguments!");
        }

        if ("test".equals(args[0])) {
            return true;
        } else {
            password = args[0];
            initVectorFile = new File(args[1]);
            files = new ArrayList<>();

            for (int i = 2; i < args.length; i++) {
                File file = new File(args[i]);
                if (!file.isDirectory()) {
                    files.add(file);
                } else {
                    addFilesFromDirectory(file, files);
                }
            }
        }

        return false;
    }

    private static void addFilesFromDirectory(File directory, ArrayList<File>
files) {
        File[] directoryFiles = directory.listFiles();

        if (directoryFiles != null) {
            for (File directoryFile : directoryFiles) {
                if (!directoryFile.isDirectory()) {
                    files.add(directoryFile);
                } else {
                    addFilesFromDirectory(directoryFile, files);
                }
            }
        }
    }

    public static String passwordToKey() throws Exception {
        String key = HexFormat.of().formatHex(
```

```
                MessageDigest.getInstance("SHA-
256").digest(password.getBytes()));

        return key.substring(48);
    }

    public static String getInitVector() throws Exception {
        if (initVectorFile.length() == 8L)
            return HexFormat.of().formatHex(
                    Files.readAllBytes(initVectorFile.toPath()));
        else {
            byte[] initVector = new byte[8];

            SecureRandom secureRandom = new SecureRandom();
            secureRandom.nextBytes(initVector);

            Files.write(initVectorFile.toPath(), initVector);

            return HexFormat.of().formatHex(initVector);
        }
    }

    public static ArrayList<File> getFiles() {
        return files;
    }
}
```

Класс Main

```
import java.io.File;
import java.io.RandomAccessFile;
import java.util.HexFormat;

class Main {

    public static void main(String[] args) throws Exception {
        if (Utils.processInput(args)) {
            DES.runTests();
        } else {
            DES.setKeys(Utils.passwordToKey());

            for (File file : Utils.getFiles()) {
                DES.setInitVector(Utils.getInitVector());

                RandomAccessFile rwFile = new RandomAccessFile(file, "rw");
                byte[] buffer = new byte[8];

                int read = rwFile.read(buffer);
                while (read != -1) {
                    rwFile.seek(rwFile.getFilePointer() - read);

                    String hexString =
DES.encryptDecryptOFB(HexFormat.of().formatHex(buffer));
                    buffer = HexFormat.of().parseHex(hexString);
                    for (int j = 0; j < read; j++) {
                        rwFile.writeByte(buffer[j]);
                    }

                    read = rwFile.read(buffer);
                }

                rwFile.close();
            }}}}
```