

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Пермский национальный исследовательский политехнический университет»
Электротехнический факультет
Кафедра «Информационные технологии и автоматизированные системы»

Дисциплина: «Защита информации»

Профиль: «Автоматизированные системы обработки информации и
управления»

Семестр 5

ОТЧЕТ

по лабораторной работе №4

Тема: «Блочное шифрование»

Выполнил: студент группы РИС-19-16

Миннахметов Э.Ю. _____

Проверил: доцент кафедры ИТАС

Шереметьев В. Г. _____

Дата _____

Пермь, 2021

ЦЕЛЬ РАБОТЫ

Получить практические навыки по созданию и применению блочных шифров.

ЗАДАНИЕ

Вариант №14. Реализовать шифрование текстового файла, методом блочного шифрования, используя блоки длиной 16 бит, ключ длиной 32 бит, реализуя в алгоритме шифрования методику DES.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Характерной особенностью блочных криптоалгоритмов является тот факт, что в ходе своей работы они производят преобразование блока входной информации фиксированной длины и получают результирующий блок того же объема, но недоступный для прочтения сторонним лицам, не владеющим ключом. Таким образом, схему работы блочного шифра можно описать функциями $Z = \text{EnCrypt}(X, \text{Key})$ и $X = \text{DeCrypt}(Z, \text{Key})$

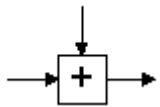
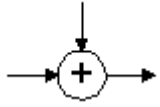
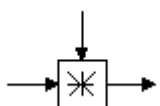
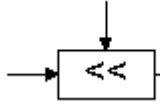
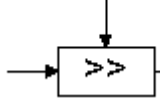
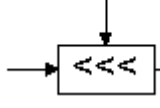
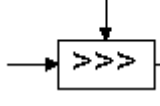
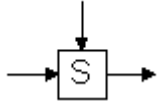
Ключ Key является параметром блочного криптоалгоритма и представляет собой некоторый блок двоичной информации фиксированного размера. Исходный (X) и зашифрованный (Z) блоки данных также имеют фиксированную разрядность, равную между собой, но необязательно равную длине ключа.

На функцию стойкого блочного шифра $Z = \text{EnCrypt}(X, \text{Key})$ накладываются следующие условия:

- Функция EnCrypt должна быть обратимой.
- Не должно существовать иных методов прочтения сообщения X по известному блоку Z , кроме как полным перебором ключей Key .
- Не должно существовать иных методов определения каким ключом Key было произведено преобразование известного сообщения X в сообщение Z , кроме как полным перебором ключей.

Все действия, производимые над данными блочным криптоалгоритмом, основаны на том факте, что преобразуемый блок может быть представлен в виде целого неотрицательного числа из диапазона, соответствующего его разрядности. Так, например, 32-битный блок данных можно интерпретировать как число из диапазона 0..4'294'967'295. Кроме того, блок, разрядность которого обычно является «степенью двойки», можно трактовать как несколько независимых неотрицательных чисел из меньшего диапазона (рассмотренный выше 32-битный блок можно также представить в виде 2 независимых чисел из диапазона 0..65535 или в виде 4 независимых чисел из диапазона 0..255).

Над этими числами блочным криптоалгоритмом и производятся по определенной схеме следующие действия (слева даны условные обозначения этих операций на графических схемах алгоритмов) :

Биективные математические функции		
	Сложение	$X' = X + V$
	Исключающее ИЛИ	$X' = X \text{ XOR } V$
	Умножение по модулю $2^N + 1$	$X' = (X * V) \bmod (2^N + 1)$
	Умножение по модулю 2^N	$X' = (X * V) \bmod (2^N)$
Битовые сдвиги		
	Арифметический сдвиг влево	$X' = X \text{ SHL } V$
	Арифметический сдвиг вправо	$X' = X \text{ SHR } V$
	Циклический сдвиг влево	$X' = X \text{ ROL } V$
	Циклический сдвиг вправо	$X' = X \text{ ROR } V$
Табличные подстановки		
	S-box (англ. substitute)	$X' = \text{Table}[X, V]$

В качестве параметра V для любого из этих преобразований может использоваться:

фиксированное число (например, $X' = X + 125$)

число, получаемое из ключа (например, $X' = X + F(\text{Key})$)

число, получаемое из независимой части блока (например, $X_2' = X_2 + F(X_1)$)

Последний вариант используется в схеме, названной по имени ее создателя сетью Фейштеля (нем. Feistel).

Сеть Фейштеля

Суть метода - смешивания текущей части шифруемого блока с результатом некоторой функции, вычисленной от другой независимой части того же блока. Эта методика получила широкое распространение, поскольку обеспечивает выполнение требования о многократном использовании ключа и материала исходного блока информации.

Классическая сеть Фейштеля имеет следующую структуру:

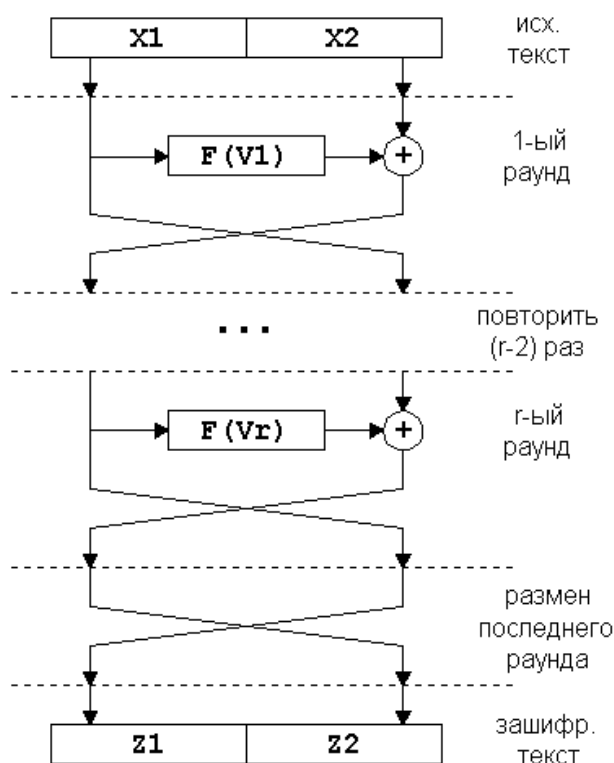


Рис.1.

Независимые потоки информации, порожденные из исходного блока, называются ветвями сети. В классической схеме их две. Величины V_i именуется параметрами сети, обычно это функции от материала ключа. Функция F называется образующей. Действие, состоящее из однократного вычисления образующей функции и последующего наложения ее результата на другую ветвь с обменом их местами, называется циклом или раундом (англ. round) сети Фейштеля. Оптимальное число раундов K – от 8 до 32. Важно то, что увеличение количества раундов значительно увеличивает криптоскойстость любого блочного шифра к криптоанализу. Возможно, эта особенность и повлияла на столь активное распространение сети Фейштеля – ведь при обнаружении, скажем, какого-либо слабого места в алгоритме, почти всегда достаточно увеличить количество раундов на 4-8, не переписывая сам алгоритм. Часто количество раундов не фиксируется разработчиками алгоритма, а лишь указываются разумные пределы (обязательно нижний, и не всегда – верхний) этого параметра.

Сеть Фейштеля обладает тем свойством, что даже если в качестве образующей функции F будет использовано необратимое преобразование, то и в этом случае вся цепочка будет восстанавливаема. Это происходит вследствие того, что для обратного преобразования сети Фейштеля не нужно вычислять функцию F^{-1} .

Более того, как нетрудно заметить, сеть Фейштеля симметрична. Использование операции XOR, обратимой своим же повтором, и инверсия последнего обмена ветвей делают возможным раскодирование блока той же сетью Фейштеля, но с инверсным порядком параметров V_i . Заметим, что для обратимости сети Фейштеля не имеет значения является ли число раундов четным или нечетным числом. В большинстве реализаций схемы, в которых оба вышеперечисленные условия (операция XOR и уничтожение последнего обмена) сохранены, прямое и обратное преобразования производятся одной и той же процедурой, которой в качестве параметра передается вектор величин V_i либо в исходном, либо в инверсном порядке.

С незначительными доработками сеть Фейштеля можно сделать и абсолютно симметричной, то есть выполняющей функции шифрования и дешифрования одним и тем же набором операций. Математическим языком это записывается как "Функция EnCrypt тождественно равна функции DeCrypt". Если мы рассмотрим граф состояний криптоалгоритма, на котором точками отмечены блоки входной и выходной информации, то при каком-то фиксированном ключе для классической сети Фейштеля мы будем иметь картину, изображенную на рис.2а, а во втором случае каждая пара точек получит уникальную связь, как изображено на рис. 2б. Модификация сети Фейштеля, обладающая подобными свойствами приведена на рисунке 3. Как видим, основная ее хитрость в повторном использовании данных ключа в обратном порядке во второй половине цикла. Необходимо заметить, однако, что именно из-за этой недостаточно исследованной специфики такой схемы (то есть потенциальной возможности ослабления зашифрованного текста обратными преобразованиями) ее используют в криптоалгоритмах с большой осторожностью.

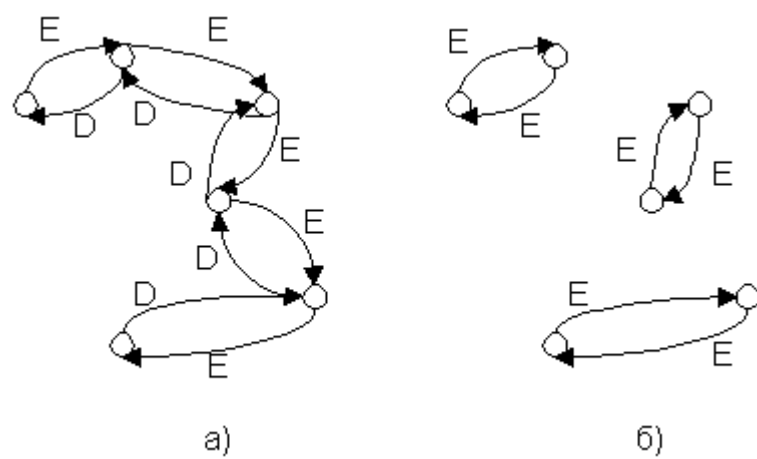


Рис.2.

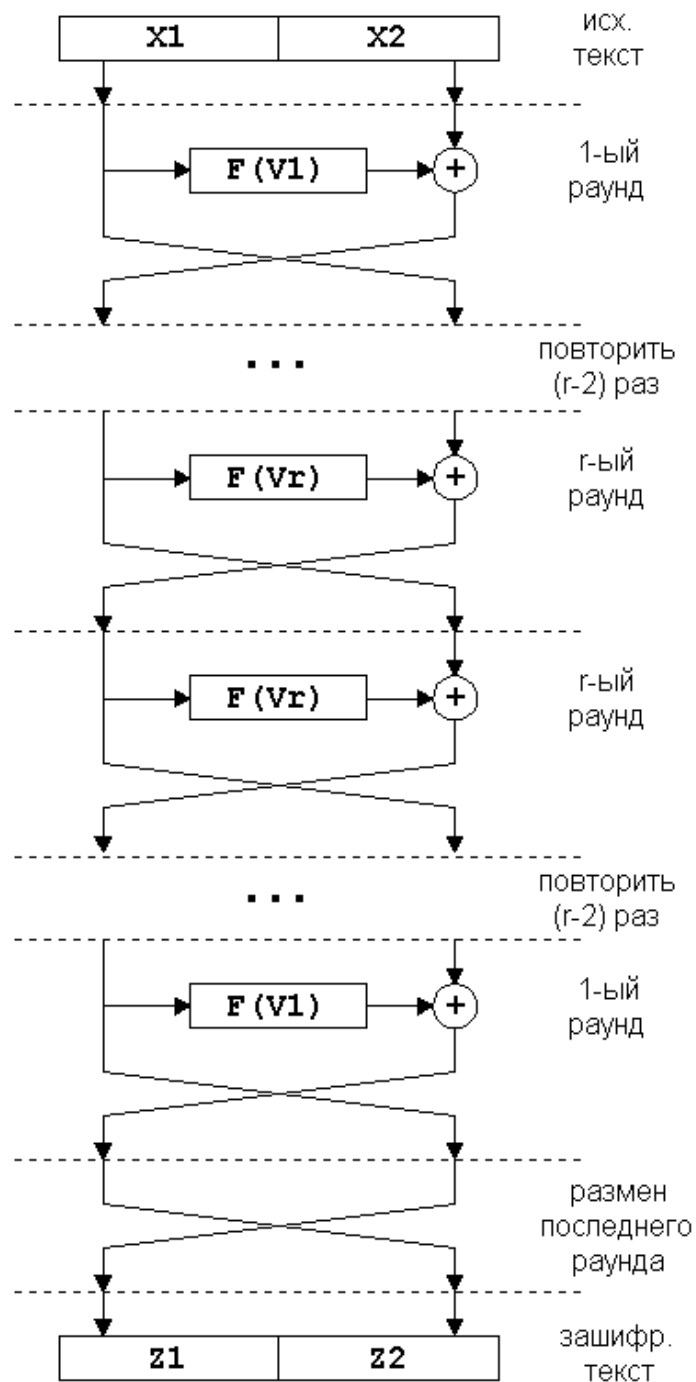


Рис.3.

А вот модификацию сети Фейштеля для большего числа ветвей применяют гораздо чаще. Это в первую очередь связано с тем, что при больших размерах кодируемых блоков (128 и более бит) становится неудобно работать с математическими функциями по модулю 64 и выше. Как известно, основные единицы информации обрабатываемые процессорами на сегодняшний день – это байт и двойное машинное слово 32 бита. Поэтому все чаще и чаще в блочных криптоалгоритмах встречается сеть Фейштеля с 4-мя ветвями. Самый простой принцип ее модификации изображен на рисунке 4а. Для более быстрого перемешивания информации между ветвями (а это основная проблема сети

Фейштеля с большим количеством ветвей) применяются две модифицированные схемы, называемые «type-2» и «type-3». Они изображены на рисунках 4б и 4в.

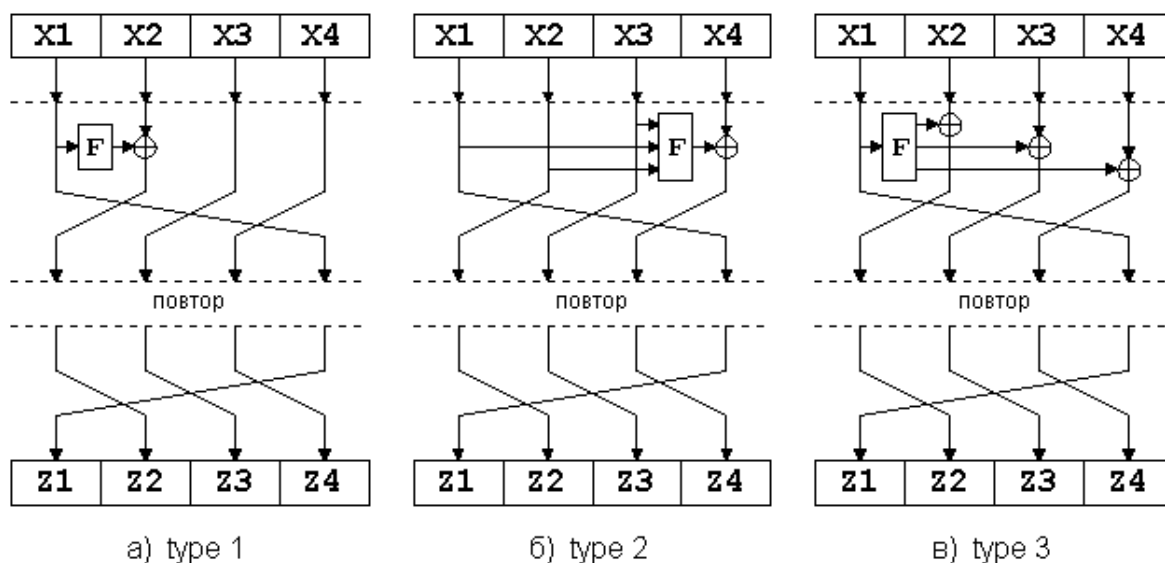


Рис.4.

Сеть Фейштеля надежно зарекомендовала себя как криптостойкая схема произведения преобразований, и ее можно найти практически в любом современном блочном шифре. Незначительные модификации касаются обычно дополнительных начальных и конечных преобразований (англоязычный термин – whitening) над шифруемым блоком. Подобные преобразования, выполняемые обычно также либо «исключающим ИЛИ» или сложением имеют целью повысить начальную рандомизацию входного текста. Таким образом, криптостойкость блочного шифра, использующего сеть Фейштеля, определяется на 95% функцией F и правилом вычисления V_i из ключа. Эти функции и являются объектом все новых и новых исследований специалистов в области криптографии.

Пример блочного шифра - Блочный шифр DES

Стандарт шифрования DES (Data Encryption Standard) был разработан в 1970-х годах, он базируется на алгоритме DEA.

Исходные идеи алгоритма шифрования данных DEA (data encryption algorithm) были предложены компанией IBM еще в 1960-х годах и базировались на идеях, описанных Клодом Шенноном в 1940-х годах. Первоначально эта методика шифрования называлась lucifer (разработчик Хорст Фейштель, название dea она получила лишь в 1976 году. Lucifer был первым блочным алгоритмом шифрования, он использовал блоки размером 128 бит и 128-битовый ключ. По существу этот алгоритм являлся прототипом DEA. В 1986 в Японии (NIT) разработан алгоритм FEAL(Fast data Encipherment

ALgorithm), предназначенный для использования в факсах, модемах и телефонах (длина ключа до 128 бит). Существует и ряд других разработок.

DEA оперирует с блоками данных размером 64 бита и использует ключ длиной 56 бит. Такая длина ключа соответствует 10^{17} комбинаций, что обеспечивало до недавнего времени достаточный уровень безопасности. В дальнейшем можно ожидать расширения ключа до 64 бит (например, LOKI) или вообще замены DES другим стандартом, например Советский шифр ГОСТ – 28147-89, алгоритм работы которого очень схож с DES использует так же 64-битные блоки, но 256-битный ключ.

Входной блок данных, состоящий из 64 бит, преобразуется в выходной блок идентичной длины. Ключ шифрования должен быть известен, как отправляющей так и принимающей сторонам. В алгоритме широко используются перестановки битов текста.

Вводится функция f , которая работает с 32-разрядными словами исходного текста (A) и использует в качестве параметра 48-разрядный ключ (J). Схеме работы функции f показана на рис. 5. Сначала 32 входные разряда расширяются до 48, при этом некоторые разряды повторяются. Схема этого расширения показана ниже (номера соответствуют номерам бит исходного 32-разрядного кода).

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

Для полученного 48-разрядного кода и ключа выполняется операция исключающее ИЛИ (XOR). Результирующий 48-разрядный код преобразуется в 32-разрядный с помощью S-матриц. На выходе S-матриц осуществляется перестановка согласно схеме показанной ниже (числа представляют собой порядковые номера бит).

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

Преобразование начинается с перестановки бит (**IP – Initial Permutation**) в 64-разрядном блоке исходных данных. 58-ой бит становится первым, 50-ый – вторым и т.д. Схема перестановки битов показана ниже.

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

Полученный блок делится на две 32-разрядные части L_0 и R_0 . Далее 16 раз повторяются следующие 4 процедуры:

Преобразование ключа с учетом номера итерации i (перестановки разрядов с удалением 8 бит, в результате получается 48-разрядный ключ)

Пусть $A=L_i$, а J – преобразованный ключ. С помощью функции $f(A,J)$ генерируется 32-разрядный выходной код.

Выполняется операция XOR для $R_i f(A,J)$, результат обозначается R_{i+1} .

Выполняется операция присвоения $L_{i+1}=R_i$.

Инверсная перестановка разрядов предполагает следующее размещение 64 бит зашифрованных данных (первым битом становится 40-ой, вторым 8-ой и т.д.).

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

S-матрицы представляют собой таблицы содержащие 4-ряда и 16 столбцов. Матрица $S(1)$ представлена ниже (эта матрица, также как и те, что приведены в ссылке 2, являются рекомендуемыми).

No. 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
 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

Исходный 48-разрядный код делится на 8 групп по 6 разрядов. Первый и последний разряд в группе используется в качестве адреса строки, а средние 4 разряда – в качестве адреса столбца. В результате каждые 6 бит кода преобразуются в 4 бита, а весь 48-разрядный код в 32-разрядный (для этого нужно 8 S-матриц). Существуют разработки, позволяющие выполнять шифрование в рамках стандарта DES аппаратным образом, что обеспечивает довольно высокое быстродействие.

Преобразования ключей K_n ($n=1,...,16$; $K_n = KS(n, key)$, где n – номер итерации) осуществляются согласно алгоритму, показанному на рис. 7.

Для описания алгоритма вычисления ключей K_n (функция KS) достаточно определить структуру «Выбора 1» и «Выбора 2», а также задать схему сдвигов влево. «Выбор 1» и «Выбор 2» представляют собой перестановки битов ключа. При необходимости биты 8, 16, ..., 64 могут использоваться для контроля четности.

Для вычисления очередного значения ключа таблица делится на две части C_0 и D_0 . В C_0 войдут биты 57, 49, 41, ..., 44 и 36, а в D_0 – 63, 55, 47, ..., 12 и 4. Так как схема сдвигов задана C_1, D_1 ; C_n, D_n и так далее могут быть легко получены из C_0 и D_0 . Так, например, C_3 и D_3 получаются из C_2 и D_2 циклическим сдвигом влево на 2 разряда

РС-1 (Выбор 1)	РС-2 (Выбор 2)
57 49 41 33 25 17 9	14 17 11 24 1 5
1 58 50 42 34 26 18	3 28 15 6 21 10
10 2 59 51 43 35 27	23 19 12 4 26 8
19 11 3 60 52 44 36	16 7 27 20 13 2
63 55 47 39 31 23 15	41 52 31 37 47 55
7 62 54 46 38 30 22	30 40 51 45 33 48
14 6 61 53 45 37 29	44 49 39 56 34 53
21 13 5 28 20 12 4	46 42 50 36 29 32

Номер итерации	Число сдвигов влево
----------------	---------------------

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

Пример блочного шифра - Блочный шифр TEA.

TEA (Tiny Encryption Algorithm) - один из самых простых в реализации, разработанный в Кэмбридже в 1985 году. Параметры шифра: длина блока - 64 бита, длина ключа - 128 бит. Оптимизирован под 32 битные процессоры. Испытан временем и является довольно криптостойким.

В алгоритме использована сеть Фейштеля с двумя ветвями в 32 бита каждая. Образующая функция F обратима. Сеть Фейштеля несимметрична из-за использования в качестве операции наложения не исключающего «ИЛИ», а арифметического сложения.

Недостатком алгоритма является некоторая медлительность, вызванная необходимостью повторять цикл Фейштеля 32 раза (это необходимо для тщательного «перемешивания данных» из-за отсутствия табличных подстановок).

Отличительной чертой криптоалгоритма TEA является его размер. Простота операций, отсутствие табличных подстановок и оптимизация под 32-разрядную архитектуру процессоров позволяет реализовать его на языке ASSEMBLER в предельно малом объеме кода. Недостатком алгоритма является некоторая медлительность, вызванная необходимостью повторять цикл Фейштеля 32 раза (это необходимо для тщательного «перемешивания данных» из-за отсутствия табличных подстановок).

ХОД РАБОТЫ

Было написано 2 приложения:

- REST-сервис на языке Kotlin для генерации случайных чисел с заданной разрядностью, использовался фреймворк Spring (исходный код в Приложении А);
- десктопное приложение на языке C++ с использованием фреймворка Qt (исходный код в Приложении Б).

На рисунке 1 представлен вывод REST-сервиса по запросу ключа шифрования в браузер.

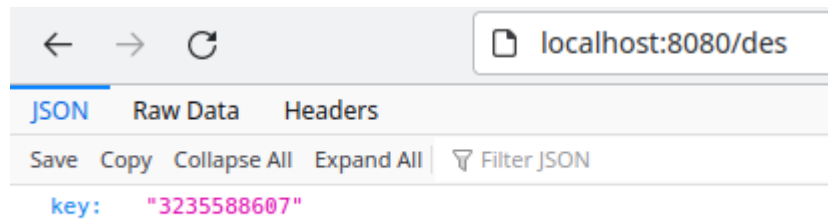


Рисунок 1 – Вывод результата запроса к серверу в браузере

На рисунке 2 представлен графический интерфейс десктопного приложения.

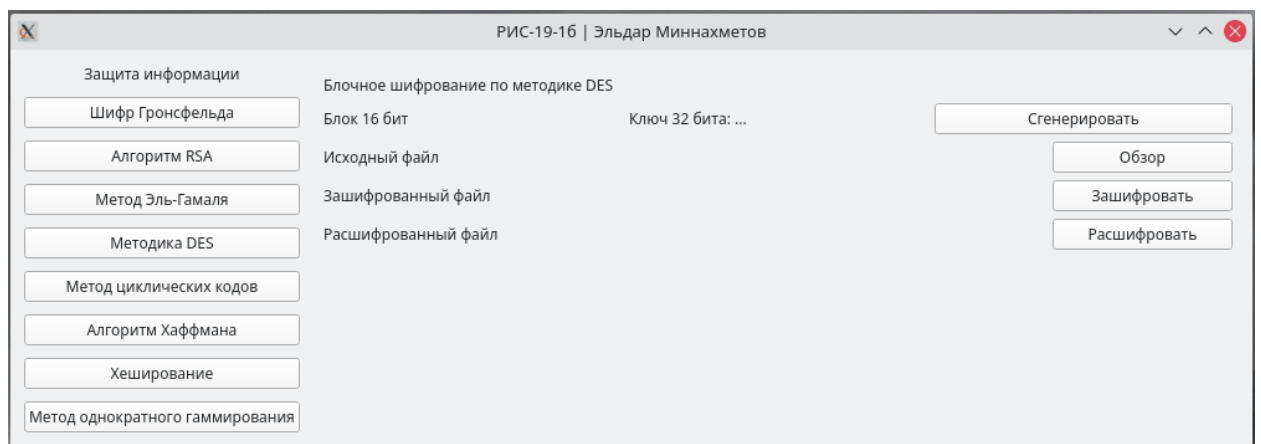


Рисунок 2 – Графический пользовательский интерфейс

После нажатия на кнопку «Сгенерировать» десктопное приложение отправляет запрос ключа шифрования. Полученный ключ вводится в соответствующее поле на форме. Результат генерации представлен на рисунке 3.

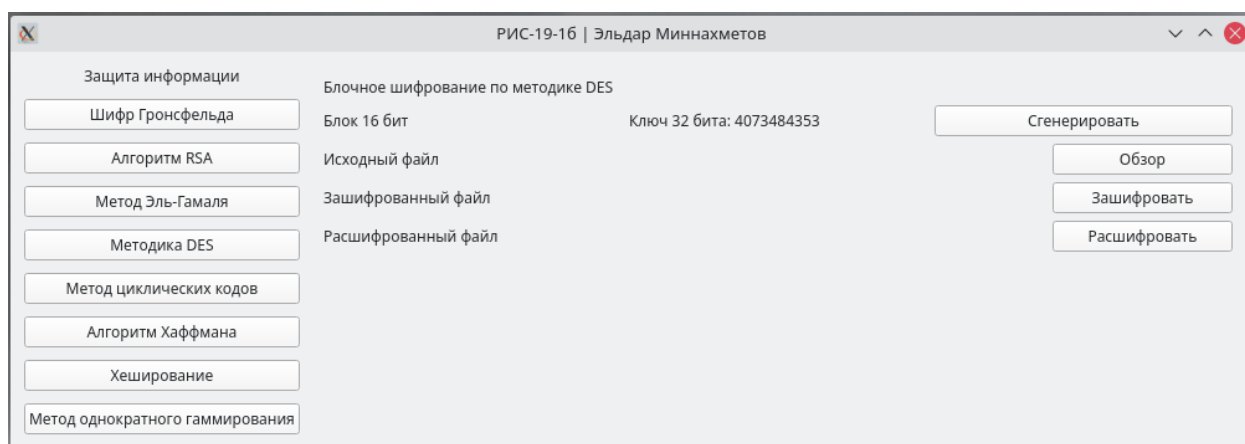


Рисунок 3 – Генерация ключа

Далее необходимо выбрать файл с исходным текстом, предварительно нажав на кнопку «Обзор», как показано на рисунке 4.

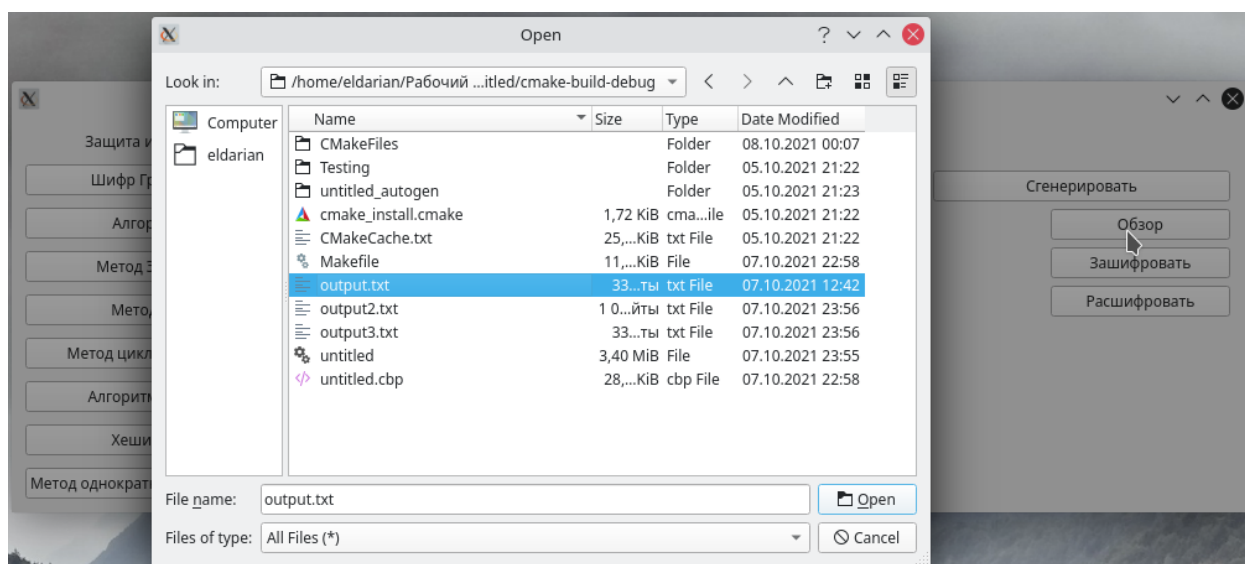


Рисунок 4 – Выбор файла с исходным текстом

А после, выбрать, если существует (перезаписать), или создать файл для записи зашифрованного текста. Таким же образом поступить и с файлом расшифрованного текста. Результат данных действий представлен на рисунке 5.

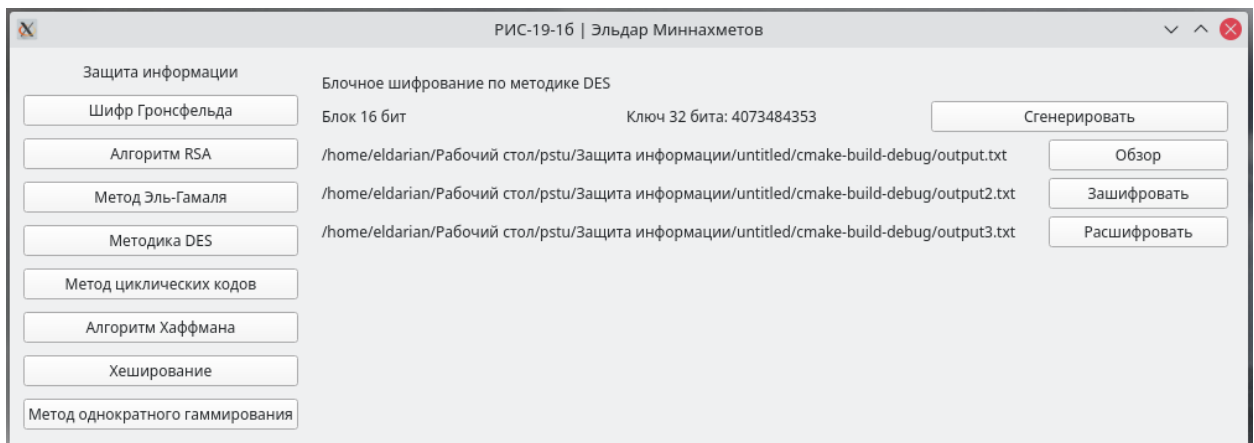


Рисунок 5 – Выбор остальных файлов для записи
зашифрованного и расшифрованного текстов

В процессе выбора файлов также выполнялись и соответствующие действия над ними, т.е. зашифровывание и расшифровывания. Теперь же необходимо посмотреть итог выполнения действий. На рисунке 6 представлен исходный текстовый файл.

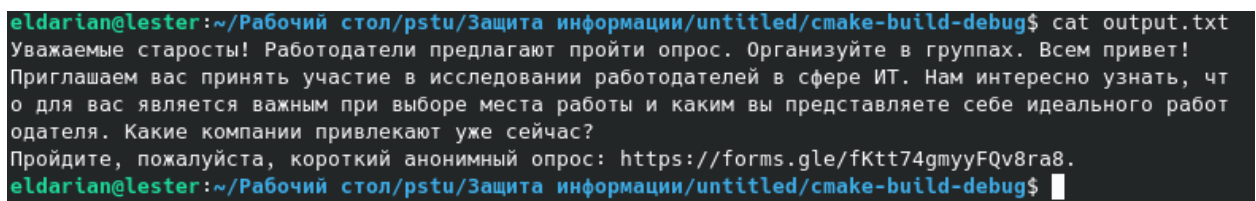


Рисунок 6 – Исходный текстовый файл

Содержимое зашифрованного текстового файла представлено на рисунке 7.

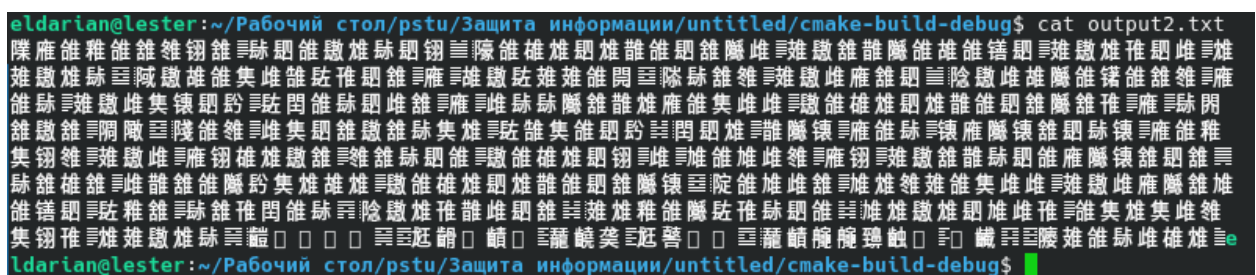


Рисунок 7 – Зашифрованный текстовый файл

Содержимое расшифрованного текстового файла представлено на рисунке 8.

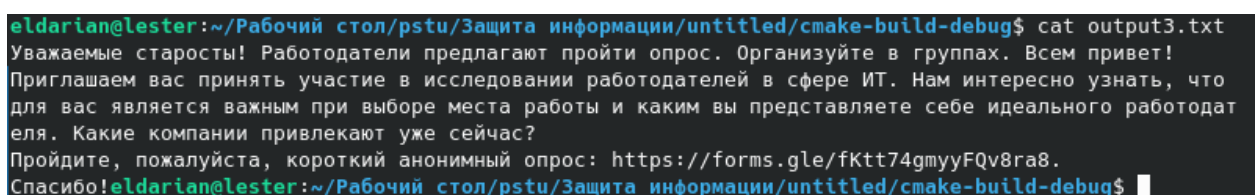


Рисунок 8 – Расшифрованный текстовый файл

ПРИЛОЖЕНИЕ А

Листинг файла BackController.kt

```
package org.eldarian.back

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController
import java.math.BigInteger
import java.util.*

@SpringBootApplication
class BackApplication

fun main(args: Array<String>) {
    runApplication<BackApplication>(*args)
}

@RestController
class LController {
    @GetMapping("/des")
    fun des(): DesResponse {
        return DesResponse(BigInteger.probablePrime(32, Random()))
    }
}

data class DesResponse (val key: String) {
    constructor(key: BigInteger) : this(key.toString())
}
```


ПРИЛОЖЕНИЕ Б

Листинг файла DesTask.h

```
#pragma once

#include "Task.h"
#include "Loader.h"

class QLabel;
class QLineEdit;
class QPushButton;
class QVBoxLayout;
class QHBoxLayout;
class QJsonObject;

class DesClient;

class DesTask: public QObject, public Task {
Q_OBJECT

private:
    DesClient *client = nullptr;

    QVBoxLayout *lytMain;
    QHBoxLayout *lytInput;
    QHBoxLayout *lytSource;
    QHBoxLayout *lytCrypt;
    QHBoxLayout *lytDecrypt;

    QLabel *lblName;
    QLabel *lblBlock;
    QLabel *lblKey;
    QLabel *lblSource;
    QLabel *lblCrypt;
    QLabel *lblDecrypt;

    QPushButton *btnGenerate;
    QPushButton *btnSource;
    QPushButton *btnCrypt;
    QPushButton *btnDecrypt;

public:
    DesTask(): Task("Методика Des") {}

    void initWidget(QWidget *wgt) override;
    void setKey(const QString& key);

public slots:
    void generate();
    void source();
    void crypt();
```

```

    void decrypt();

private:
    void crypt(QLabel *source, QLabel *target, bool d);

};

class DesLoader : public LoadTask {
private:
    DesTask* task;

public:
    explicit DesLoader(DesTask* task) : task(task) {}
    QString query() override { return "des"; }
    void done(QJsonObject& json) override;

};

class DesClient {
private:
    long long key;

public:
    explicit DesClient(long long key) : key(key) {}

    QString crypt(QString text, bool d) const;

};

```

Листинг файла DesTask.cpp

```
#include "DesTask.h"

#include <QLabel>
#include <QLineEdit>
#include <QJsonObject>
#include <QHBoxLayout>
#include <QPushButton>
#include <QFileDialog>
#include <QMessageBox>

#include "Des.h"

void DesTask::initWidget(QWidget *wgt) {
    lytMain = new QVBoxLayout;
    lytInput = new QHBoxLayout;
    lytSource = new QHBoxLayout;
    lytCrypt = new QHBoxLayout;
    lytDecrypt = new QHBoxLayout;

    lblName = new QLabel("Блочное шифрование по методике Des");
    lblBlock = new QLabel("Блок 16 бит");
    lblKey = new QLabel("Ключ 32 бита: ...");
    lblSource = new QLabel("Исходный файл");
    lblCrypt = new QLabel("Зашифрованный файл");
    lblDecrypt = new QLabel("Расшифрованный файл");

    btnGenerate = new QPushButton("Сгенерировать");
    btnSource = new QPushButton("Обзор");
    btnCrypt = new QPushButton("Зашифровать");
    btnDecrypt = new QPushButton("Расшифровать");

    wgt->setLayout(lytMain);
    lytMain->addWidget(lblName);
    lytMain->addLayout(lytInput);
    lytInput->addWidget(lblBlock);
    lytInput->addWidget(lblKey);
    lytInput->addWidget(btnGenerate);
    lytMain->addWidget(lblSource);
    lytMain->addLayout(lytSource);
    lytSource->addWidget(lblSource, 4);
    lytSource->addWidget(btnSource, 1);
    lytMain->addLayout(lytCrypt);
    lytCrypt->addWidget(lblCrypt, 4);
    lytCrypt->addWidget(btnCrypt, 1);
    lytMain->addLayout(lytDecrypt);
    lytDecrypt->addWidget(lblDecrypt, 4);
    lytDecrypt->addWidget(btnDecrypt, 1);

    lytMain->setAlignment(Qt::Alignment::enum_type::AlignTop);

    connect(btnGenerate, SIGNAL(released()), this, SLOT(generate()));
```

```

connect(btnSource, SIGNAL(released()), this, SLOT(source()));
connect(btnCrypt, SIGNAL(released()), this, SLOT(crypt()));
connect(btnDecrypt, SIGNAL(released()), this, SLOT(decrypt()));
}

void DesTask::setKey(const QString& key) {
    if (client) {
        delete client;
    }
    client = new DesClient(key.toLongLong());
    lblKey->setText("Ключ 32 бита: " + key);
}

void DesTask::generate() {
    (new DesLoader(this))->run();
}

void DesTask::source() {
    lblSource->setText(QFileDialog::getOpenFileName());
}

void DesTask::crypt() {
    crypt(lblSource, lblCrypt, false);
}

void DesTask::decrypt() {
    crypt(lblCrypt, lblDecrypt, true);
}

void DesTask::crypt(QLabel *source, QLabel *target, bool d) {
    QFile flSource(source->text());
    QFile flTarget(QFileDialog::getSaveFileName());
    if(!flSource.open(QIODevice::ReadOnly) || !flTarget.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(nullptr, "Предупреждение", "Исходный файл не выбран или не существует");
        return;
    }
    target->setText(flTarget.fileName());
    QString sourceStr(flSource.readAll());
    flTarget.write(client->crypt(sourceStr, d).toUtf8());
    flSource.close();
    flTarget.close();
}

void DesLoader::done(QJsonObject& json) {
    task->setKey(json["key"].toString());
}

QString DesClient::crypt(QString text, bool d) const {
    int n = text.length();
    QString r;
    std::string in;
    for(int i = 0; i < n; ++i) {

```

```
    ushort t = text[i].unicode();
    in += char(t / 256);
    in += char(t % 256);
}
std::string out = myEncrypt(in, key, d);
for(int i = 0; i < 2 * n; i += 2) {
    r += QChar(int(out[i]) * 256 + int(out[i + 1]));
}
return r;
}
```

Листинг файла Des.h

```
#pragma once

#include <stdint>

#define BUFF_SIZE 1024
#define LSHIFT_28BIT(x, L) (((x) << (L)) | ((x) >> (-(L) & 27))) & (((uint64_t)1 << 32) - 1))

static const uint8_t Sbox[8][4][16] = {
    { // 0
        {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},
    },
    { // 1
        {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},
    },
    { // 2
        {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},
    },
    { // 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},
    },
    { // 4
        {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},
    },
    { // 5
        {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},
    },
    { // 6
        {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},
    },
    { // 7
        {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},
    },
};

static const uint8_t IP[64] = {
    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,
};

static const uint8_t FP[64] = {
    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,
};

static const uint8_t K1P[28] = {
    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,
};

static const uint8_t K2P[28] = {
    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,
};

static const uint8_t CP[48] = {
    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,
};

static const uint8_t EP[48] = {
    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,
};

static const uint8_t P[32] = {
    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,
};

size_t DES(uint8_t * to, uint8_t mode, uint8_t * keys8b, uint8_t * from, size_t length);

void key_expansion(uint64_t key64b, uint64_t * keys48b);

```

```

void key_permutation_56bits_to_28bits(uint64_t block56b, uint32_t * block32b_1, uint32_t * block32b_2);
void key_expansion_to_48bits(uint32_t block28b_1, uint32_t block28b_2, uint64_t * keys48b);
uint64_t key_contraction_permutation(uint64_t block56b);

```

```

void feistel_cipher(uint8_t mode, uint32_t * N1, uint32_t * N2, uint64_t * keys48b);
void round_feistel_cipher(uint32_t * N1, uint32_t * N2, uint64_t key48b);
uint32_t func_F(uint32_t block32b, uint64_t key48b);
uint64_t expansion_permutation(uint32_t block32b);
uint32_t substitutions(uint64_t block48b);
void substitution_6bits_to_4bits(uint8_t * blocks6b, uint8_t * blocks4b);
uint32_t permutation(uint32_t block32b);

```

```

uint8_t extreme_bits(uint8_t block6b);
uint8_t middle_bits(uint8_t block6b);

```

```

uint64_t initial_permutation(uint64_t block64b);
uint64_t final_permutation(uint64_t block64b);

```

```

void split_64bits_to_32bits(uint64_t block64b, uint32_t * block32b_1, uint32_t * block32b_2);
void split_64bits_to_8bits(uint64_t block64b, uint8_t * blocks8b);
void split_48bits_to_6bits(uint64_t block48b, uint8_t * blocks6b);

```

```

uint64_t join_32bits_to_64bits(uint32_t block32b_1, uint32_t block32b_2);
uint64_t join_28bits_to_56bits(uint32_t block28b_1, uint32_t block28b_2);
uint64_t join_8bits_to_64bits(uint8_t * blocks8b);
uint32_t join_4bits_to_32bits(uint8_t * blocks8b);

```

```

static inline void swap(uint32_t * N1, uint32_t * N2);

```

```

std::string myEncrypt(std::string text, uint key, bool d) {
    uint8_t encrypted[BUFF_SIZE];
    uint8_t buffer[BUFF_SIZE] = {0};
    uint8_t keys8b[8] = {0};

```

```

    for(int i = 0; i < 4; ++i) {
        keys8b[i] = ((uint8_t*)&key)[i];
    }

```

```

    size_t length = text.length();
    for(int i = 0; i < length; ++i) {
        buffer[i] = text[i];
    }

```

```

    length = DES(encrypted, d ? 'D' : 'E', keys8b, buffer, length);

```

```

    std::string r;
    for(int i = 0; i < length; ++i) {
        r += char(encrypted[i]);
    }
    return r;
}

```



```

size_t DES(uint8_t * to, uint8_t mode, uint8_t * keys8b, uint8_t * from, size_t length) {
    length = length % 8 == 0 ? length : length + (8 - (length % 8));

    uint64_t keys48b[16] = {0};
    uint32_t N1, N2;

    key_expansion(
        join_8bits_to_64bits(keys8b),
        keys48b
    );

    for (size_t i = 0; i < length; i += 8) {
        split_64bits_to_32bits(
            initial_permutation(
                join_8bits_to_64bits(from + i)
            ),
            &N1, &N2
        );
        feistel_cipher(mode, &N1, &N2, keys48b);
        split_64bits_to_8bits(
            final_permutation(
                join_32bits_to_64bits(N1, N2)
            ),
            (to + i)
        );
    }
}

return length;
}

void feistel_cipher(uint8_t mode, uint32_t * N1, uint32_t * N2, uint64_t * keys48b) {
    switch(mode) {
        case 'E': case 'e': {
            for (int8_t round = 0; round < 16; ++round) {
                round_feistel_cipher(N1, N2, keys48b[round]);
            }
            swap(N1, N2);
            break;
        }
        case 'D': case 'd': {
            for (int8_t round = 15; round >= 0; --round) {
                round_feistel_cipher(N1, N2, keys48b[round]);
            }
            swap(N1, N2);
            break;
        }
    }
}

void round_feistel_cipher(uint32_t * N1, uint32_t * N2, uint64_t key48b) {
    uint32_t temp = *N2;
    *N2 = func_F(*N2, key48b) ^ *N1;
    *N1 = temp;
}

```

```

}

uint32_t func_F(uint32_t block32b, uint64_t key48b) {
    uint64_t block48b = expansion_permutation(block32b);
    block48b ^= key48b;
    block32b = substitutions(block48b);
    return permutation(block32b);
}

uint64_t expansion_permutation(uint32_t block32b) {
    uint64_t block48b = 0;
    for (uint8_t i = 0; i < 48; ++i) {
        block48b |= (uint64_t)((block32b >> (32 - EP[i])) & 0x01) << (63 - i);
    }
    return block48b;
}

uint32_t substitutions(uint64_t block48b) {
    uint8_t blocks4b[4], blocks6b[8] = {0};
    split_48bits_to_6bits(block48b, blocks6b);
    substitution_6bits_to_4bits(blocks6b, blocks4b);
    return join_4bits_to_32bits(blocks4b);
}

void substitution_6bits_to_4bits(uint8_t * blocks6b, uint8_t * blocks4b) {
    uint8_t block2b, block4b;

    for (uint8_t i = 0, j = 0; i < 8; i += 2, ++j) {
        block2b = extreme_bits(blocks6b[i]);
        block4b = middle_bits(blocks6b[i]);
        blocks4b[j] = Sbox[i][block2b][block4b];

        block2b = extreme_bits(blocks6b[i+1]);
        block4b = middle_bits(blocks6b[i+1]);
        blocks4b[j] = (blocks4b[j] << 4) | Sbox[i+1][block2b][block4b];
    }
}

uint8_t extreme_bits(uint8_t block6b) {
    return ((block6b >> 6) & 0x2) | ((block6b >> 2) & 0x1);
}

uint8_t middle_bits(uint8_t block6b) {
    return (block6b >> 3) & 0xF;
}

uint32_t permutation(uint32_t block32b) {
    uint32_t new_block32b = 0;
    for (uint8_t i = 0; i < 32; ++i) {
        new_block32b |= ((block32b >> (32 - P[i])) & 0x01) << (31 - i);
    }
    return new_block32b;
}

```

```

}

uint64_t initial_permutation(uint64_t block64b) {
    uint64_t new_block64b = 0;
    for (uint8_t i = 0; i < 64; ++i) {
        new_block64b |= ((block64b >> (64 - IP[i])) & 0x01) << (63 - i);
    }
    return new_block64b;
}

uint64_t final_permutation(uint64_t block64b) {
    uint64_t new_block64b = 0;
    for (uint8_t i = 0; i < 64; ++i) {
        new_block64b |= ((block64b >> (64 - FP[i])) & 0x01) << (63 - i);
    }
    return new_block64b;
}

void key_expansion(uint64_t key64b, uint64_t* keys48b) {
    uint32_t K1 = 0, K2 = 0;
    key_permutation_56bits_to_28bits(key64b, &K1, &K2);
    key_expansion_to_48bits(K1, K2, keys48b);
}

void key_permutation_56bits_to_28bits(uint64_t block56b, uint32_t* block28b_1, uint32_t* block28b_2) {
    for (uint8_t i = 0; i < 28; ++i) {
        *block28b_1 |= ((block56b >> (64 - K1P[i])) & 0x01) << (31 - i);
        *block28b_2 |= ((block56b >> (64 - K2P[i])) & 0x01) << (31 - i);
    }
}

void key_expansion_to_48bits(uint32_t block28b_1, uint32_t block28b_2, uint64_t* keys48b) {
    uint64_t block56b;
    uint8_t n;

    for (uint8_t i = 0; i < 16; ++i) {
        switch(i) {
            case 0: case 1: case 8: case 15: n = 1; break;
            default: n = 2; break;
        }

        block28b_1 = LSHIFT_28BIT(block28b_1, n);
        block28b_2 = LSHIFT_28BIT(block28b_2, n);
        block56b = join_28bits_to_56bits(block28b_1, block28b_2);
        keys48b[i] = key_contraction_permutation(block56b);
    }
}

uint64_t key_contraction_permutation(uint64_t block56b) {
    uint64_t block48b = 0;
    for (uint8_t i = 0; i < 48; ++i) {
        block48b |= ((block56b >> (64 - CP[i])) & 0x01) << (63 - i);
    }
}

```

```

    return block48b;
}

void split_64bits_to_32bits(uint64_t block64b, uint32_t * block32b_1, uint32_t * block32b_2) {
    *block32b_1 = (uint32_t)(block64b >> 32);
    *block32b_2 = (uint32_t)(block64b);
}

void split_64bits_to_8bits(uint64_t block64b, uint8_t * blocks8b) {
    for (size_t i = 0; i < 8; ++i) {
        blocks8b[i] = (uint8_t)(block64b >> ((7 - i) * 8));
    }
}

void split_48bits_to_6bits(uint64_t block48b, uint8_t * blocks6b) {
    for (uint8_t i = 0; i < 8; ++i) {
        blocks6b[i] = (block48b >> (58 - (i * 6))) << 2;
    }
}

uint64_t join_32bits_to_64bits(uint32_t block32b_1, uint32_t block32b_2) {
    uint64_t block64b;
    block64b = (uint64_t)block32b_1;
    block64b = (uint64_t)(block64b << 32) | block32b_2;
    return block64b;
}

uint64_t join_28bits_to_56bits(uint32_t block28b_1, uint32_t block28b_2) {
    uint64_t block56b;
    block56b = (block28b_1 >> 4);
    block56b = ((block56b << 32) | block28b_2) << 4;
    return block56b;
}

uint64_t join_8bits_to_64bits(uint8_t * blocks8b) {
    uint64_t block64b;
    for (uint8_t *p = blocks8b; p < blocks8b + 8; ++p) {
        block64b = (block64b << 8) | *p;
    }
    return block64b;
}

uint32_t join_4bits_to_32bits(uint8_t * blocks4b) {
    uint32_t block32b;
    for (uint8_t *p = blocks4b; p < blocks4b + 4; ++p) {
        block32b = (block32b << 8) | *p;
    }
    return block32b;
}

static inline void swap(uint32_t * N1, uint32_t * N2) {
    uint32_t temp = *N1;
    *N1 = *N2;

```

```
*N2 = temp;  
}
```