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»

Направление 09.03.01 – «Информатика и вычислительная техника»

Дисциплина: «Защита информации»

Профиль: «Автоматизированные системы обработки информации и управления»

Семестр 7

ОТЧЕТ

по лабораторной работе №4

Тема: «Блочные шифры»

Выполнил: студент группы АСУ-14-1б

Калмыков В. А. \_\_\_\_\_\_\_\_\_\_\_

Проверил: доцент кафедры ИТАС

Шереметьев В. Г. \_\_\_\_\_\_\_\_\_\_\_

Дата \_\_\_\_\_\_

Пермь, 2017

**ЦЕЛЬ РАБОТЫ**

Получить практические навыки по созданию и применению блочных шифров.

**ЗАДАНИЕ**

**Вариант №6**. Реализовать шифрование текстового файла, методом блочного шиф-рования, используя блоки длиной 64 бит, ключ длиной 32 бит.

**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=EnCrypt(X,Key)* и *X=DeCrypt(Z,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=EnCrypt(X,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 | | |
| 2221-1 | Сложение | X'=X+V |
| 2221-2 | Исключающее ИЛИ | X'=X XOR V |
|  | Умножение по модулю 2N+1 | X'=(X\*V) mod (2N+1) |
| 2221-3 | Умножение по модулю 2N | X'=(X\*V) mod (2N) |
| Битовые сдвиги | | |
| 2221-4 | Арифметический сдвиг влево | X'=X SHL V |
| 2221-5 | Арифметический сдвиг вправо | X'=X SHR V |
| 2221-6 | Циклический сдвиг влево | X'=X ROL V |
| 2221-7 | Циклический сдвиг вправо | X'=X ROR V |
| Табличные подстановки | | |
| 2221-8 | S-box (англ. substitute) | X'=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(Key)*)

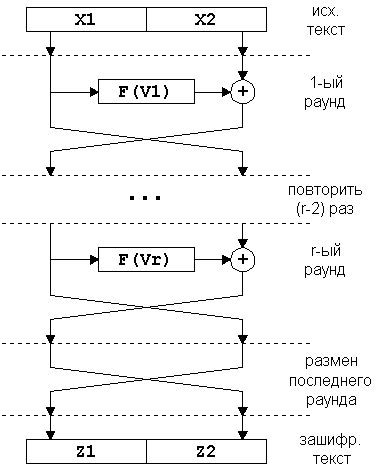
число, получаемое из независимой части блока (например, *X2'=X2+F(X1)*)

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:

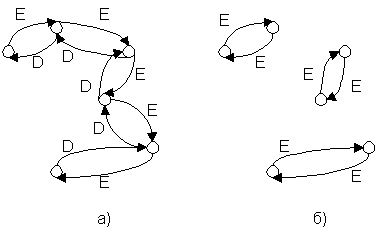
  
**Рис.1.**

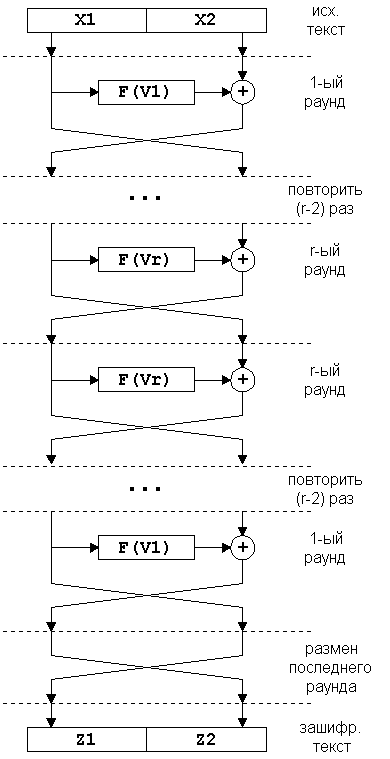
Независимые потоки информации, порожденные из исходного блока, называются ветвями сети. В классической схеме их две. Величины Vi именуются параметрами сети, обычно это функции от материала ключа. Функция F называется образующей. Действие, состоящее из однократного вычисления образующей функции и последующего 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 восстановима. Это происходит вследствие того, что для обратного преобразования сети Фейштеля не нужно вычислять функцию F-1.

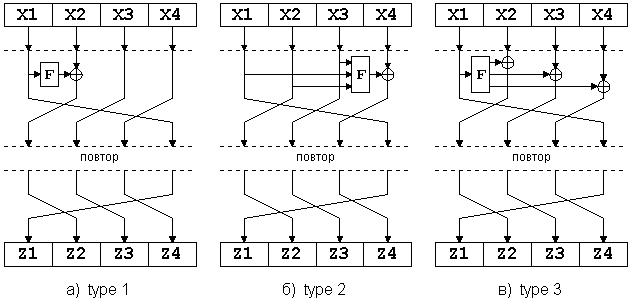
Более того, как нетрудно заметить, сеть Фейштеля симметрична. Использование операции XOR, обратимой своим же повтором, и инверсия последнего обмена ветвей делают возможным раскодирование блока той же сетью Фейштеля, но с инверсным порядком параметров Vi. Заметим, что для обратимости сети Фейштеля не имеет значение 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 Vi 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 оконечных 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 Vi 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 1017 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 DEC 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 (А) 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 L0 и R0. 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=Li, а J – преобразованный ключ. С помощью функции f(A,J) генерируется 32-разрядный выходной код.  
Выполняется операция XOR для Ri f(A,J), результат обозначается Ri+1.  
Выполняется операция присвоения Li+1=Ri.

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 Kn (n=1,…,16; Kn = KS(n,key), где n – номер итерации) осуществляются согласно алгоритму, показанному на рис. 7.

Для описания алгоритма вычисления ключей Kn (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 С0 и D0. В С0 войдут биты 57, 49, 41,…, 44 и 36, а в D0 – 63, 55, 47,…, 12 и 4. Так как схема сдвигов задана C1,D1; Cn, Dn и так далее могут быть легко получены из C0 и D0. Так, например, C3 и D3 получаются из C2 и D2 циклическим сдвигом влево на 2 разряда

|  |  |
| --- | --- |
| **PC-1 (Выбор 1)** | **PC-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**

На рисунке 1 представлена главная форма программы.

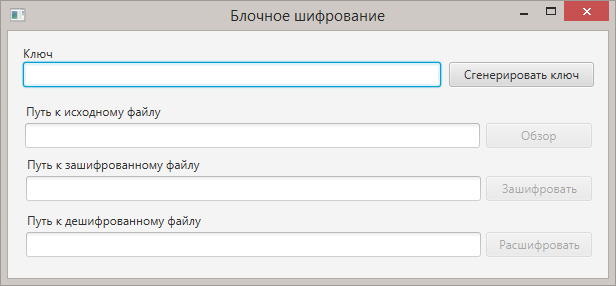


Рисунок 1 – Главная форма программы.

Пример работы программы представлен на рисунке 2.

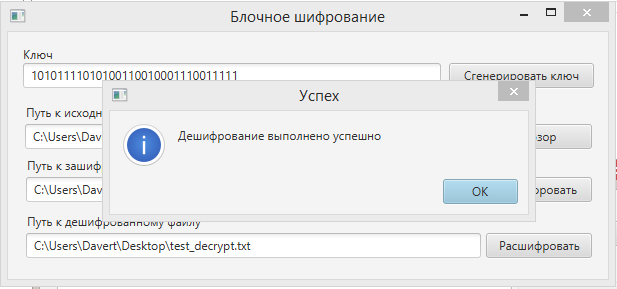


Рисунок 2 – Пример работы программы.

Содержимое исходного текстового файла представлено на рисунке 3.

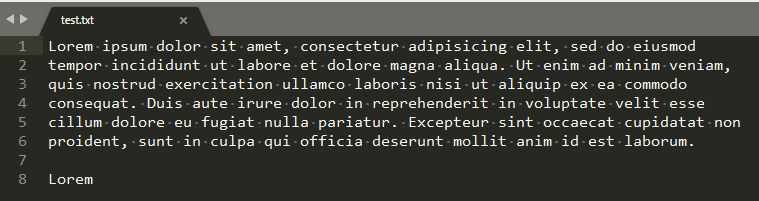


Рисунок 3 – Исходный текстовый файл.

Содержимое зашифрованного текстового файла представлено на рисунке 4.

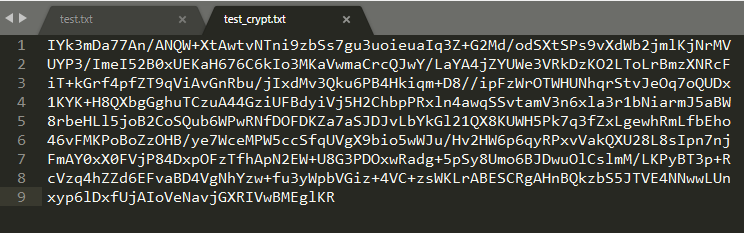


Рисунок 4 – Зашифрованный текстовый файл.

Содержимое расшифрованного текстового файла представлено на рисунке 5.

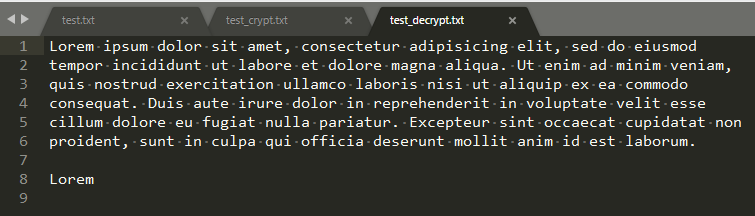


Рисунок 5 – Расшифрованный текстовый файл.

**ПРИЛОЖЕНИЕ А**

**Листинг класса BlockCipher.java**

package Models;

import java.io.UnsupportedEncodingException;

import java.math.BigInteger;

import java.util.Random;

public class BlockCipher {

private static final int BLOCK\_SIZE = 64;

private static final int CHAR\_SIZE = 16;

private static final int KEY\_SIZE = 32;

private String key;

private String decryptKey;

private int rounds = 16;

private int shiftKey = 2;

public BlockCipher() {

Random rnd = new Random();

BigInteger bi = BigInteger.probablePrime(BlockCipher.KEY\_SIZE, rnd);

this.key = bi.toString(2);

}

public String getKey() {

return key;

}

public String padStringToLength(String str) {

while((str.length() \* BlockCipher.CHAR\_SIZE) % BlockCipher.BLOCK\_SIZE != 0) {

str += "#";

}

return str;

}

public String getBinaryString(String str) throws UnsupportedEncodingException {

String result = "";

byte[] bytes = str.getBytes("UTF-8");

for(byte b : bytes) {

String elem = Integer.toBinaryString(b);

while(elem.length() < BlockCipher.CHAR\_SIZE) {

elem = "0" + elem;

}

result += elem;

}

return result;

}

public String[] divideBinaryStringToBlocks(String binaryStr) {

String[] bStr = new String[binaryStr.length() / BlockCipher.BLOCK\_SIZE];

int lengthOfBlock = binaryStr.length() / bStr.length;

for(int i = 0; i < bStr.length; i++) {

bStr[i] = binaryStr.substring(i \* lengthOfBlock, (i + 1) \* lengthOfBlock);

}

return bStr;

}

public String[] encrypt(String[] encryptStrBlocks) throws UnsupportedEncodingException {

String bKey = this.getBinaryString(this.key);

for(int i = 0; i < this.rounds; i++) {

for(int j = 0; j < encryptStrBlocks.length; j++) {

encryptStrBlocks[j] = this.encode(encryptStrBlocks[j], bKey);

}

bKey = this.keyForNextRound(bKey);

}

this.decryptKey = this.keyForPrevRound(bKey);

String[] result = new String[encryptStrBlocks.length];

for(int i = 0; i < encryptStrBlocks.length; i++) {

String encryptStrBlock = encryptStrBlocks[i];

result[i] = encryptStrBlock;

}

return result;

}

private String keyForNextRound(String key) {

for(int i = 0; i < this.shiftKey; i++) {

key = key.substring(key.length() - 1, key.length()) + key;

key = key.substring(0, key.length() - 1);

}

return key;

}

private String keyForPrevRound(String key) {

for(int i = 0; i < this.shiftKey; i++) {

key += key.charAt(0);

key = key.substring(1, key.length());

}

return key;

}

private String encode(String input, String key) {

String L = input.substring(0, input.length() / 2);

String R = input.substring(input.length() / 2, input.length());

return this.XOR(this.f(L, key), R) + L;

}

private String f(String s1, String s2) {

return this.XOR(s1, s2);

}

private String XOR(String s1, String s2) {

String result = "";

for(int i = 0; i < s1.length(); i++) {

boolean a = s1.substring(i, i + 1).equals("1");

boolean b = s2.substring(i, i + 1).equals("1");

result += a ^ b ? "1" : "0";

}

return result;

}

public String getStringFromBinaryBlocks(String[] input) {

String result = "";

for(int i = 0; i < input.length; i++) {

int a = 0;

int degree = input[i].length() - 1;

for(char ch : input[i].toCharArray()) {

a += (int) ch \* (int) Math.pow(2, degree--);

}

result += String.valueOf((char) a);

}

return result;

}

public String[] decrypt(String[] decryptStrBlocks) throws UnsupportedEncodingException {

String bKey = this.getBinaryString(this.decryptKey);

for(int i = 0; i < this.rounds; i++) {

for(int j = 0; j < decryptStrBlocks.length; j++) {

decryptStrBlocks[j] = this.dencode(decryptStrBlocks[j], bKey);

}

bKey = this.keyForPrevRound(bKey);

}

String[] result = new String[decryptStrBlocks.length];

for(int i = 0; i < decryptStrBlocks.length; i++) {

String encryptStrBlock = decryptStrBlocks[i];

result[i] = encryptStrBlock;

}

return result;

}

private String dencode(String input, String key) {

String L = input.substring(0, input.length() / 2);

String R = input.substring(input.length() / 2, input.length());

return R + this.XOR(L, this.f(R, key));

}

}

**ПРИЛОЖЕНИЕ Б**

**Листинг класса SampleController.java**

package Forms;

import Models.BlockCipher;

import Models.ShadowCipher;

import javafx.fxml.FXML;

import javafx.scene.control.Alert;

import javafx.scene.control.Button;

import javafx.scene.control.TextField;

import javafx.stage.FileChooser;

import java.io.\*;

public class SampleController {

@FXML

private TextField txtKey;

@FXML

private TextField txtEncryptKey;

@FXML

private TextField txtPathToFile;

@FXML

private TextField txtPathToEncryptFile;

@FXML

private TextField txtPathToDecryptFile;

@FXML

private Button btnKeygen;

@FXML

private Button btnOpenFile;

@FXML

private Button btnEncrypt;

@FXML

private Button btnDecrypt;

private BlockCipher bc;

private ShadowCipher sc;

private String pathToFile;

private String orgText;

private String[] bBlocks;

private String result;

@FXML

private void initialize() throws Exception {

this.sc = new ShadowCipher();

}

@FXML

private void handleBtnKeygenClick() {

this.bc = new BlockCipher();

this.txtKey.setText(bc.getKey());

this.btnOpenFile.setDisable(false);

}

@FXML

private void handleBtnOpenFileClick() {

FileChooser fc = new FileChooser();

fc.setTitle("Открытие текстового файла");

fc.getExtensionFilters().add(new FileChooser.ExtensionFilter("Текстовые файлы", "\*.txt"));

File selectedFile = fc.showOpenDialog(null);

if (selectedFile != null) {

this.pathToFile = selectedFile.getAbsolutePath();

this.txtPathToFile.setText(this.pathToFile);

this.btnEncrypt.setDisable(false);

}

}

@FXML

private void handleBtnEncryptClick() {

try(BufferedReader br = new BufferedReader(new FileReader(this.pathToFile))) {

this.orgText = "";

String line;

while((line = br.readLine()) != null) {

this.orgText += line + "<br>";

}

br.close();

String textForEncrypt = this.bc.padStringToLength(this.orgText);

textForEncrypt = this.bc.getBinaryString(textForEncrypt);

this.bBlocks = this.bc.divideBinaryStringToBlocks(textForEncrypt);

this.result = this.sc.encrypt(this.orgText);

String path = this.pathToFile.replaceFirst("[.][^.]+$", "") + "\_crypt.txt";

BufferedWriter bw = new BufferedWriter(new FileWriter(path));

bw.write(this.result);

bw.close();

this.txtPathToEncryptFile.setText(path);

this.btnDecrypt.setDisable(false);

this.showMessage("Успех", "", "Шифрование выполнено успешно", Alert.AlertType.INFORMATION);

} catch(Exception e) {

this.showMessage("Ошибка", "", e.getMessage(), Alert.AlertType.ERROR);

}

}

@FXML

private void handleBtnDecryptClick() {

String dir = this.pathToFile.replaceFirst("[.][^.]+$", "");

String path = dir + "\_crypt.txt";

try(BufferedReader br = new BufferedReader(new FileReader(path))) {

String textFromFile = "";

String line;

while((line = br.readLine()) != null) {

textFromFile += line;

}

br.close();

String textForDecrypt = this.bc.getBinaryString(textFromFile);

String[] textBlocks = this.bc.divideBinaryStringToBlocks(textForDecrypt);

this.result = this.sc.decrypt(textFromFile);

String result = this.bc.getStringFromBinaryBlocks(this.bBlocks);

this.result = this.result.replace("<br>", "\r\n").replace("#", "");

path = dir + "\_decrypt.txt";

BufferedWriter bw = new BufferedWriter(new FileWriter(path));

bw.write(this.result);

bw.close();

this.txtPathToDecryptFile.setText(path);

this.showMessage("Успех", "", "Дешифрование выполнено успешно", Alert.AlertType.INFORMATION);

} catch(Exception e) {

this.showMessage("Ошибка", "", e.getMessage(), Alert.AlertType.ERROR);

}

}

private void showMessage(String title, String headerText, String message, Alert.AlertType type) {

Alert alert = new Alert(type);

alert.setTitle(title);

alert.setHeaderText(headerText);

alert.setContentText(message);

alert.showAndWait();

}

}

**ПРИЛОЖЕНИЕ В**

**Листинг класса Main.java**

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void start(Stage primaryStage) throws Exception{

Parent root = FXMLLoader.load(getClass().getResource("/Forms/sample.fxml"));

primaryStage.setTitle("Блочное шифрование");

primaryStage.setScene(new Scene(root));

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}