МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Математический факультет

Кафедра компьютерной безопасности и прикладной алгебры

**КУРСОВАЯ РАБОТА**

**Реализация и сравнение алгоритмов симметричного шифрования на примере DES и LOKI97**

Выполнил студент:

**Гутров Роман Михайлович**

группы МК-201

очной формы обучения

специальности

10.05.01 Компьютерная безопасность

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

(подпись)

«\_\_\_» июля 2022 г.

Научный руководитель:

Ручай Алексей Николаевич

Должность: зав. кафедрой

Ученая степень: канд.физ.-мат. наук

Ученое звание: доцент

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

(подпись)

«\_\_\_» июля 2022 г.

Научный руководитель:

Панасенко Дмитрий Игоревич

Должность: ст. преподаватель

Ученая степень: нет

Ученое звание: нет

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

(подпись)

«\_\_\_» июля 2022 г.

**Оглавление**

[**Введение** 3](#_Toc108203240)

[**1. Базовые определения** 4](#_Toc108203241)

[**2. Применение симметричного шифрования** 5](#_Toc108203242)

[**3.Математическая модель симметрического шифрования** 5](#_Toc108203243)

[**4.Сеть Фейстеля** 5](#_Toc108203244)

[**5.Алгоритм симметричного шифрования DES** 6](#_Toc108203245)

[**5.1 Схема шифрования алгоритма** 6](#_Toc108203246)

[**5.2 Начальная и конечная перестановка** 7](#_Toc108203247)

[**5.3 Раунды** 7](#_Toc108203248)

[**5.4 Основная функция** 8](#_Toc108203249)

[**5.5 Генерация ключей для раундов** 10](#_Toc108203250)

[**5.6 Схема расшифрования** 11](#_Toc108203251)

[**6. Алгоритм симметричного шифрования LOKI97** 11](#_Toc108203252)

[**6.1 Схема шифрования алгоритма** 11](#_Toc108203253)

[**6.2 Основная функция .** 12](#_Toc108203254)

[**6.2.1 KP-перестановка.** 12](#_Toc108203255)

[**6.2.2 -перестановка** 12](#_Toc108203256)

[**6.2.3 -boxes** 13](#_Toc108203257)

[**6.2.4 P – перестановка** 13](#_Toc108203258)

[**6.2.5 Конечные S-боксы.** 13](#_Toc108203259)

[**6.3 Процедура расширения ключа** 13](#_Toc108203260)

[**6.4 Схема расшифрования** 14](#_Toc108203261)

[**7. Сравнение и тестирование работы алгоритмов DES и LOKI97** 15](#_Toc108203262)

[**Заключение** 15](#_Toc108203263)

[**Список литературы** 16](#_Toc108203264)

[**Приложение** 17](#_Toc108203265)

# **Введение**

Информация уже давно стала одним из самых важных и ценных ресурсов. Поэтому проблема ее защиты встает очень остро. С целью разрешить эту проблему появилась такая наука, как криптография.

В начале двадцатого века криптографией занимались в основном госструктуры, с целью передачи зашифрованных сообщений. Сегодня же шифрование используется повсеместно: шифрование паролей, сообщений, документов, аудио и видео информации, цифровые подписи. Шифрование позволяет скрыть информацию от посторонних и сохранить ее от злоумышленников.

Сегодня, алгоритмы шифрования делятся на симметричные и ассиметричные. Основное различие – симметричные алгоритмы используют одни и те же ключи для шифрования и дешифрования, в то время как ассиметричные используют разные.

Симметричное шифрование позволяет работать с большим объемом данных, т.к. быстрее работают, поскольку не требуют формирования новых ключей для дешифрования.

Целью данной работы является реализация и сравнение двух алгоритмов симметричного шифрования: DES и LOKI97.

**Основная часть**

# **1. Базовые определения**

Шифр – семейство обратимых отображений множества последовательностей блоков текстов (сообщений) открытых в множество последовательностей блоков текстов (сообщении) шифрованных и обратно, задаваемых функцией шифрования. Каждое из отображений определяется некоторым параметром, называемым ключом, и описывается некоторым алгоритмом шифрования, реализующим один из режимов шифрования.

Ключ – изменяемый элемент (параметр), каждому значению которого однозначно соответствует одно из отображений, реализуемых криптосистемой. Все возможные значения ключа составляют множество ключевое криптосистемы.

Симметричное шифрование – способ шифрования данных, при котором один и тот же ключ используется и для кодирования, и для восстановления информации.

Ассиметричное шифрование – метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом.

Алгоритмы шифрования – алгоритм зашифрования, реализующий при каждом фиксированном значении ключа одно обратимое отображение множества блоков текста открытого, имеющих фиксированную длину. Представляет собой алгоритм простой замены блоков текста фиксированной длины.

Сеть Фейстеля – способ построения цикла (раунда) шифрования в алгоритмах шифрования итеративных (блочных) на основе регистра сдвига нелинейного длины 2 с функцией обратной связи, зависящей от ключа циклового (раундового).

Раунд – один шаг шифрования, в ходе которого одна или несколько частей шифруемого блока данных подвергается модификации путём применения круговой функции.

# **2. Применение симметричного шифрования**

Симметричное шифрование потребляет относительно мало системных ресурсов, по сравнению с другими методами шифрования. Благодаря этому оно применяется в основном для быстрого шифрования больших объемов данных.

Симметричное шифрование, например, применяется в мессенджерах с целью защиты передаваемых сообщений, а сервисы для видеосвязи используют его для защиты аудио и видео потоков. Так же симметричное шифрование имеет широкое применение в банковском деле и приложениях с карточными транзакциями. Этот вид шифрования позволяет обеспечить высокую скорость шифрования и высокую защиту данных.

Более того, AES, преемник DES – идеальный алгоритм для беспроводных сетей, которые используют протокол WPA2.

Но несмотря на все преимущества симметричное шифрование не может использоваться для формирования цифровых подписей и сертификатов, потому что секретный ключ при использовании этого метода должен быть известен всем, что противоречит идее цифровой подписи.

# **3.Сеть Фейстеля**

Сеть Фейстеля – один из методов построения блочных шифров. Сеть состоит из ячеек, называемых ячейками Фейстеля. На вход каждой ячейки поступают данные и ключ. На выходе каждой ячейки получают изменённые данные и изменённый ключ. Все ячейки однотипны, и говорят, что сеть представляет собой определённую многократно повторяющуюся (итерированную) структуру. Ключ выбирается в зависимости от алгоритма шифрования/расшифрования и меняется при переходе от одной ячейки к другой. При шифровании и расшифровании выполняются одни и те же операции; отличается только порядок ключей. Ввиду простоты операций сеть Фейстеля легко реализовать как программно, так и аппаратно.

# **4.Алгоритм симметричного шифрования DES**

Стандарт шифрования данных DES (Data Encryption Standard) – блочный шифр с симметричными ключами, разработан Национальным Институтом Стандартов и Технологии.

## **4.1 Схема шифрования алгоритма**

Алгоритм принимает 64-битовый открытый текст и порождает 64-битовый зашифрованный текст и наоборот. В обоих случаях для шифрования 64-битового блока используется один и тот же 56-битовый ключ. Процесс шифрования состоит из двух перестановок, которые называют начальной и финальной перестановками, и 16 раундов Фейстеля. Каждый раунд использует различные сгенерированные 48-битовые ключи. На рисунке 1 изображена схема алгоритма.

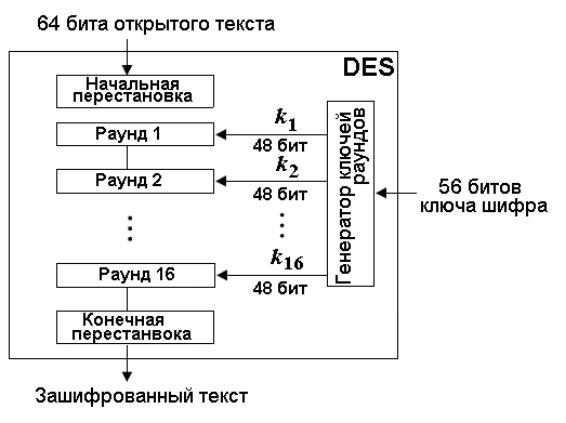


Рисунок 1. Схема алгоритма DES

## **4.2 Начальная и конечная перестановка**

На вход начальной и конечной перестановки поступает 64 бита, которые затем переставляются в соответствии с заданными таблицами (Таблица 1).



Таблица 1. IP-перестановки

Эти перестановки взаимно обратны. Таким образом на выходе получаем блок размером 64 бит, который состоит из тех же битов, что и входной блок. Другими словами, 58-й бит на входе начальной перестановки переходит в 1-ую позицию на выходе из нее. А финальная перестановка 1-ый входной бит переведет в 58-ую позицию на выходе. Обе перестановки не имеют никакого значения в крипто стойкости алгоритма.

## **4.3 Раунды**

DES использует 16 раундов. Каждый раунд DES применяет шифр Фейстеля. (Рисунок 2)

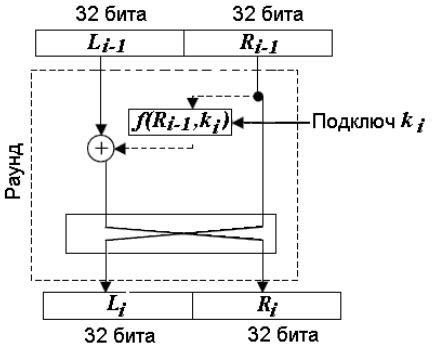


Рисунок 2. Схема раунда DES

Раунд принимает полублоки от предыдущего раунда (или начального) и создает полублоки для входа в следующий раунд (или конечный блок перестановки). Все необратимые элементы сосредоточены в функции .

## **4.4 Основная функция**

Функция с помощью 48-битового ключа раунда зашифровывает 32 правых бит , чтобы получить на выходе 32-битовый результат. Функция содержит 4 составляющие (Рисунок 3): операция XOR, P-бокс расширения, группу S-боксов и прямой бокс.

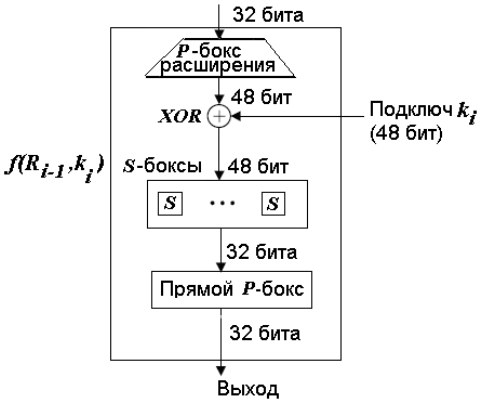


Рисунок 3. Схема основной функции.

P-бокс расширяет 32-битовый блок до 48 битов, чтобы согласовать его размер с размером ключа раунда. Блок делится на 8 секций по 4 бита, каждая секция затем расширяется до 6 бит. Отношения между входом и выходом могут быть как определены математически, так и заданы таблицей (Таблица 2).



Таблица 2. P-бокс расширения.

После расширения блока, используется операция XOR над блоком и ключом раунда

Затем, после суммирования с ключом, блок из 48 бит делится на 8 последовательных 6-битовых векторов , каждый из которых заменяется на 4-битовый вектор с помощью S-боксов (Приложение 1). Чтобы найти в S-боксе шифрообозначения вектора, надо:

1. из 1-го и последнего битов вектора образовать двоичное число, перевести его в десятичную систему. Это будет номер строки m (m = 0,1,2,3);
2. из 2-го, 3-го, 4-го и 5-го бита вектора образовать двоичное число, перевести его в десятичную систему. Это будет номер столбца l (l = 0,...,15);
3. Число, стоящее в S -боксе на пересечении m-ой строки и l-го столбца, будет шифрообозначение
4. Шифрообозначение перевести в двоичную систему. Это число будет новым 6-битным значением.

Таким образом, после S-боксов мы получаем 8 4-битовых векторов , которые объединяют в 32-битовый блок. Далее биты блока переставляются в прямом P-боксе согласно таблице (Таблица 3).



Таблица 3. Прямой P-бокс

После 16-го раунда DES правый и левый блоки уже не меняются местами, а объединяются в блок и подвергаются финальной перестановке .

## **4.5 Генерация ключей для раундов**

DES создает 16 раундовых ключей по 48 битов из ключа шифра на 56 бит. Ключевое расписание состоит из этапов:

1. Перестановка сжатия 64-битового ключа до 56 бит для удаления битов проверки. Из исходного ключа удаляют биты 8,16,24,32,40,48,56,64 и переставляют остальные биты согласно таблице (Таблица 4).



Таблица 4. Удаление проверочных бит из исходного ключа

1. После перестановки 56 битов, ключ делится на два блока по 28 бита каждый. Далее для генерации раундовых ключей из блоков с помощью операции циклического сдвига влево на 1-2 бита строятся блоки . В раундах 1,2,9,16 смещение – на 1 бит, в других – на 2 бита. После определения блоков биты этих блоков объединяются в 56-битовый ключ.
2. Перестановка сжатия (Таблица 5) изменяет 56 битов на 48 битов, которые образуют раундовый ключ.



Таблица 5. Перестановка сжатия

## **4.6 Схема расшифрования**

Для расшифрования применяются те же действия, что и для шифрования, за исключением раундовых ключей – они используются в обратном порядке. Т.е. для 1-го раунда используется 16-ый раундовый ключ.

# **5. Алгоритм симметричного шифрования LOKI97**

Блочный алгоритм с симметричными ключами. lOKI97 шифрует данные блоками по 128 битов и использует ключи размером 128, 192 и 256 битов.

## **5.1 Схема шифрования алгоритма**

Алгоритм базируется на сети Фейстеля: данные шифруемого блока разбиваются на два субблока по 64 бита, над одним из которых выполняются следующие операции:

1. Сложение с первым фрагментом ключа раунда по модулю .
2. Наложение функции , выполняющей нелинейное преобразование данных и использующей в качестве аргумента второй аргумент ключа раунда.
3. Сложение по модулю с третьим фрагментом ключа раунда.

После выполнения перечисленных действий обработанный субблок накладывается на другой применением операции XOR, после чего субблоки меняются местами (за исключением последнего раунда). Схема алгоритма приведена на рисунке 1.1:

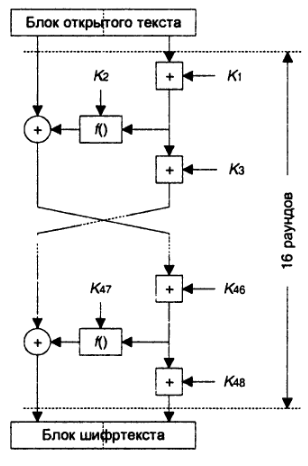


Рисунок 1.1. Схема шифрования алгоритма LOKI97

## **5.2 Основная функция .**

Основная функция алгоритма LOKI97. Используется в раундах сети Фейстеля и для расширения ключа до 256 бит. Сама функция состоит из KP-перестановки, E-перестановки, -перестановок, P-перестановки и конечной -перестановки.

### **5.2.1 KP-перестановка.**

KP (Keyed Permutation) – простая перестановка входных данных на основе младших 32 битов используемого функцией фрагмента ключа раунда , где i – номер раунда. Шаги выполнения перестановки:

1. Входные данные разбиваются на два субблока по 32 бита.
2. Для каждого единичного бита фрагмента ключа выполняется перестановка соответствующих ему битов субблоков между собой. Т.е. если нулевой бит ключа имеет значение 1, то нулевой бит первого субблока входных данных принимает значение нулевого бита второго субблока и наоборот.

### **5.2.2 -перестановка**

E (Expansion) – генерация 96-битной последовательности на основе 64-битного результата операции KP. Последовательность формируется на основе следующих битов входа: [4-0,63-56,58-48,52-40,42-32,34-24,28-16,18-8,12-0].

### **5.2.3 -boxes**

(Substitution boxes) – таблицы замен. обрабатывает 13-битные фрагменты 96-битной последовательности, полученной операцией , – 11-битные фрагменты. Замена выполняется согласно следующим формулам (в формулах указаны шестнадцатеричные константы):

### **5.2.4 P – перестановка**

P-перестановка – побитовая перестановка входной последовательности согласно таблице 1.1:

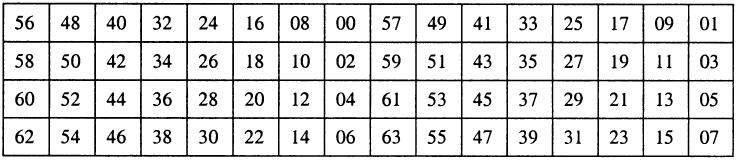


Таблица 1.1. P-перестановка

### **5.2.5 Конечные S-боксы.**

64-битный результат перестановки P и 32-битная старшая часть фрагмента ключа раунда повторно обрабатываются таблицей замен. Входная 64-битная последовательность поровну (по 8 битов) распределяется между таблицами, а из фрагмента ключа раунда берутся недостающие биты до необходимых 13 или 11 битов. После формирования S-таблиц снова выполняется преобразование согласно формулам.

После всех преобразований субблоки меняются местами (за исключением последнего раунда) и объединяются в один 128-битовый блок.

## **5.3 Процедура расширения ключа**

Процедура расширения ключа позволяет на основе ключа шифрования получить 48 фрагментов ключей раундов , по три из которых используются в каждом раунде алгоритма.

В начале процедуры выполняется инициализация исходной 256-битной ключевой последовательности, которая, в зависимости от длины ключа шифрования, выполняется одним из следующих способов:

1. 256-битный ключ используется в неизменном виде
2. 192-битный ключ дополняется результатом применения описанной выше функции , которая в качестве обрабатываемых данных (первый аргумент) использует первые 64 бита ключа шифрования, а в качестве подключа (второй аргумент) – следующие 64 бита ключа шифрования.
3. 128-битный ключ дополняется результатом применения функции с обратным относительно 192-битного ключа порядком аргументов, а затем дополняется аналогичным 192-битному ключу образом.

Дальнейшая обработка исходной ключевой последовательности основана на сети Фейстеля (Рисунок 1.2), после каждого раунда которой получается фрагмент ключа раунда алгоритма.

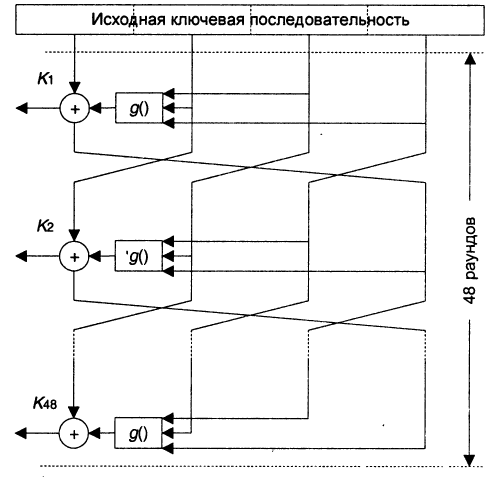


Рисунок 1.2. Схема получения фрагментов ключа раунда

В каждом раунде три 64-битных фрагмента текущего состояния ключевой последовательности обрабатываются функцией и накладываются операцией XOR на оставшийся 64-битный фрагмент. Функция описывается следующим образом: , где:

i-номер текущего раунда процедуры расширения ключа;

∆ - константа, равная: ∆ = .

## **5.4 Схема расшифрования**

Для расшифрования применяются те же действия, что и для шифрования, за исключением раундовых ключей – они используются в обратном порядке. Т.е. для 1-го раунда используется 16-ый раундовый ключ.

# **6. Сравнение и тестирование работы алгоритмов DES и LOKI97**

Результат работы алгоритмов представлен в таблице 3.1

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Размер файла | Время шифрования | Время дешифрования |
| DES | 330 Кб | 1.517 c | 1.491 c |
| 1 Мб | 4.446 c | 4.399 c |
| 100 Мб | 470.122 c | 479.730 |
| 1000 Мб | 4676.847 с | 5390.203 с |
| LOKI97 | 330 Кб | 0.901 c | 890 c |
| 1 Мб | 2.629 c | 2.626 c |
| 100 Мб | 268.864 с | 268.928 с |
| 1000 Мб | 2774.584 c | 2760.802 c |

Таблица 3.1. Сравнение алгоритмов DES и LOKI97

# **Заключение**

В ходе работ мы изучили, реализовали и сравнили алгоритмы симметричного шифрования DES и LOKI97. Исходя из результатов работ алгоритмов наибольшую эффективность показывает алгоритм LOKI97. Это связано с тем, что алгоритм использует блок размером 128 бит, против 64 бит у DES. Так же, LOKI97 использует ключ большего размера: 256 бит, что делает его более безопасным и криптостойким в сравнении с DES. Все поставленные задачи были выполнены.

# **Список литературы**

[1] Сушко С.А. Практическая криптология [Текст]: учебник для вузов. – 116

с.

[2] Панасенко С. П. Алгоритмы шифрования [Текст]: специальный справочник. – СПб.: БХВ-Петербург, 2009. – 576 с.

[3] DES [Электронный ресурс]: Википедия. Свободная энциклопедия. - Режим доступа: <https://ru.wikipedia.org/wiki/DES>

# **Приложение**

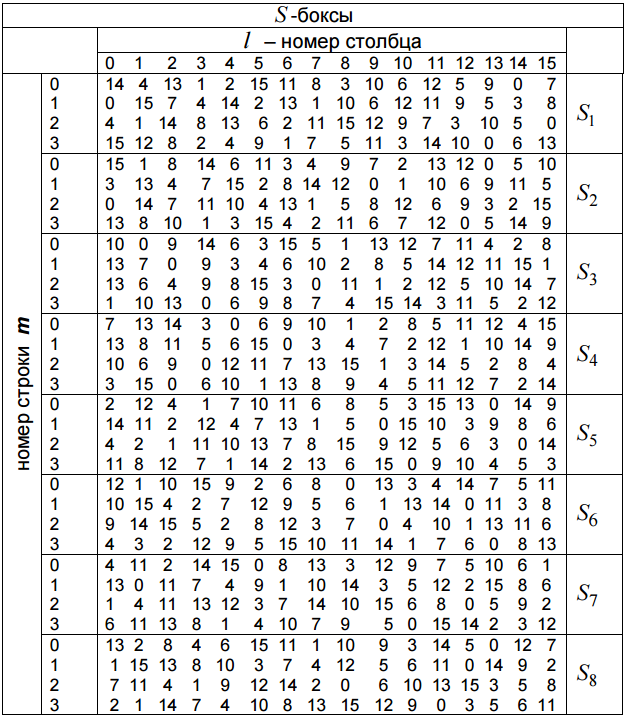


Таблица S – перестановок

Код DES

Main.cpp

#include <iostream>

#include "DES.h"

int main()

{

DES des;

des.encrypt();

des.decrypt();

cout << "\nEncryption time: " << des.exec\_time\_enc.count() << " millisec | File size: " << des.sizeEncFile << " KByte\n";

cout << "\nDecryption time: " << des.exec\_time\_dec.count() << " millisec | File size: " << des.sizeDecFile << " KByte\n";

system("pause");

return 0;

}

DES.cpp

#include "DES.h"

#include "chrono"

#define ENCR true

#define DECR false

void DES::init\_file()

{

ifstream in;

path = "data.pdf";

int choise = 1;

//открытие файла для чтения иходного текста и файла для записи шифротекста

cout << "1 - Use default path (C://Users/gutro/.../data.pdf) \n2 - Enter path\n=>";

cin >> choise;

if (choise == 1)

in.open(path, std::ios::binary);

else {

cout << "\nEnter path data: ";

cin >> path;

in.open(path, std::ios::binary);

}

if (!in.is\_open())

{

cout << "\nFile cannot open!" << endl;

system("pause");

exit(-10);

}

in.close();

sizeSourceFile = getSizeFile(path);

cout << "\nEnter key(hex format): ";

string hex\_str\_key, bin\_str\_key;

try {

cin >> hex\_str\_key;

bin\_str\_key = convert\_string(hex\_str\_key);

key = bitset<64>(string(bin\_str\_key));

}

catch (std::exception& ex)

{

cout << "\nINPUT ERROR: " << ex.what() << endl;

system("pause");

exit(-15);

}

}

void DES::IP\_first()

{

bitset<64> tmp = data;

int IP[64] =

{

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,

56, 48, 40, 32, 24, 16, 8, 0,

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6

};

for (int i = 0; i < 64; i++)

{

data.set(i, tmp.test(IP[i]));

}

}

void DES::IP\_second()

{

bitset<64> tmp = data;

int FP[64] =

{

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,

32, 0, 40, 8, 48, 16, 56, 24

};

for (int i = 0; i < 64; i++)

{

data.set(i, tmp.test(FP[i]));

}

}

void DES::encrypt()

{

init\_file();

auto start\_time = chrono::steady\_clock::now();

Enc\_filename = path;

Enc\_filename.insert(0, "ENC\_");

ifstream in(path, std::ios::binary);

ofstream output(Enc\_filename, std::ios::binary);

if (!key\_flag)

{

key\_flag = true;

key\_extension();

}

//получать и шифровать блоки пока не закончится исходный файл

while (!in.eof())

{

in.read((char\*)&data, sizeof(bitset<64>));

for (int i=0; i<16; i++)

round(i, ENCR);

output.write((char\*)&data, sizeof(bitset<64>));

}

sizeEncFile = getSizeFile(Enc\_filename);

auto end\_time = chrono::steady\_clock::now();

exec\_time\_enc = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time);

}

bitset<32> DES::block\_convertion(bitset<32> BLOCK, bitset<48> R\_key)

{

bitset<48> TMPblock48;

bitset<32> TMPblock32[2];

bitset<6> Sblock6[8];

bitset<4>Sblock4[8];

int P[32] = {

15, 6, 19, 20,

28, 11, 27, 16,

0, 14, 22, 25,

4, 17, 30, 9,

1, 7, 23, 13,

31, 26, 2, 8,

18, 12, 29, 5,

21, 10, 3, 24

};

TMPblock48 = EP(BLOCK);

// XOR операция над раундовым ключом и подблоком

TMPblock48 ^= R\_key;

//разбиение 48-битного блока на 8 S-блоков по 6 бит

for (int it = 0; it < 48; it++)

{

if (it < 6) Sblock6[0].set(it, TMPblock48.test(it));

if (it >= 6 && it < 12) Sblock6[1].set(it - 6, TMPblock48.test(it));

if (it >= 12 && it < 18) Sblock6[2].set(it - 12, TMPblock48.test(it));

if (it >= 18 && it < 24) Sblock6[3].set(it - 18, TMPblock48.test(it));

if (it >= 24 && it < 30) Sblock6[4].set(it - 24, TMPblock48.test(it));

if (it >= 30 && it < 36) Sblock6[5].set(it - 30, TMPblock48.test(it));

if (it >= 36 && it < 42) Sblock6[6].set(it - 36, TMPblock48.test(it));

if (it >= 42 && it < 48) Sblock6[7].set(it - 42, TMPblock48.test(it));

}

apply\_Sbox(&Sblock6[0], &Sblock4[0]);

//объединение восьми 4-битовых блоков в 32-битный блок

int tmpB\_it = 0;

for (int it = 0; it < 8; it++)

{

for (int \_it = 0; \_it < 4; \_it++)

{

TMPblock32[0].set(tmpB\_it, Sblock6[it].test(\_it));

tmpB\_it++;

}

}

//перестановка после S-боксов

TMPblock32[1] = TMPblock32[0];

for (int it = 0; it < 32; it++)

{

TMPblock32[0].set(it, TMPblock32[1].test(P[it]));

}

return TMPblock32[0];

}

void DES::round(int j, bool flag)

{

bitset<32> temp;

if (flag == ENCR)

{

IP\_first();

//разделение 64-битового блока

for (int i = 0; i < 32; i++)

{

leftBlock[i] = data[i];

rightBlock[i] = data[i + 32];

}

leftBlock ^= block\_convertion(rightBlock, RoundKey[j]);

for (int i = 0; i < 32; i++)

{

data[i] = rightBlock[i];

data[i+32] = leftBlock[i];

}

IP\_second();

return;

}

if (flag == DECR)

{

IP\_first();

for (int i = 0; i < 32; i++)

{

leftBlock[i] = data[i];

rightBlock[i] = data[i + 32];

}

rightBlock ^= block\_convertion(leftBlock, RoundKey[15-j]);

for (int i = 0; i < 32; i++)

{

data[i] = rightBlock[i];

data[i + 32] = leftBlock[i];

}

IP\_second();

}

}

void DES::apply\_Sbox(bitset<6>\* Sblock6, bitset<4>\* Sblock4)

{

int swap;

int S\_BOX[8][4][16] = {

{

{ 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 }

}

};

int position[2] = { 0,0 }, tmp = 0;

//получение перестановкой 4-битовых S-блоков

for (int it = 0; it < 8; it++)

{

// определить число из таблицы

if (Sblock6[it][0] == 0 && Sblock6[it][5] == 0) position[0] = 0; //номер строки

else if (Sblock6[it][0] == 0 && Sblock6[it][5] == 1) position[0] = 1; //номер строки

else if (Sblock6[it][0] == 1 && Sblock6[it][5] == 0) position[0] = 2; //номер строки

else if (Sblock6[it][0] == 1 && Sblock6[it][5] == 1) position[0] = 3; //номер строки

tmp = 0;

for (int \_it = 4; \_it > 0; \_it--) //номер столбца в двоичной СС

{

tmp = tmp \* 10 + Sblock6[it][\_it];

}

position[1] = conv\_to\_dec(tmp); //номер столбца в 10 СС

Sblock4[it] = S\_BOX[it][position[0]][position[1]]; //задаем значение 4-битового битсета

}

}

void DES::decrypt()

{

auto start\_time = chrono::steady\_clock::now();

path.insert(0, "DEC\_");

ifstream in(Enc\_filename, std::ios::binary);

ofstream output(path, std::ios::binary);

//получать и шифровать блоки пока не закончится исходный файл

while (!in.eof())

{

in.read((char\*)&data, sizeof(bitset<64>));

for (int i = 0; i < 16; i++)

round(i, DECR);

output.write((char\*)&data, sizeof(bitset<64>));

}

sizeDecFile = getSizeFile(path);

auto end\_time = chrono::steady\_clock::now();

exec\_time\_dec = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time);

}

bitset<48> DES::EP(bitset<32> &block)

{

int Pbox[] = {

31, 0, 1, 2, 3, 4,

3, 4, 5, 6, 7, 8,

7, 8, 9, 10, 11, 12,

11, 12, 13, 14, 15, 16,

15, 16, 17, 18, 19, 20,

19, 20, 21, 22, 23, 24,

23, 24, 25, 26, 27, 28,

27, 28, 29, 30, 31, 0

};

bitset<48> EPblock;

for (int i = 0; i < 48; i++)

{

EPblock[i] = block[Pbox[i]];

}

return EPblock;

}

void DES::key\_extension()

{

bitset<28> tmpKey1, tmpKey2;

bitset<56> tmpKEY56;

int P1[] = {

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

},

P2[] = {

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

},

RP[] = {

13, 16, 10, 23, 0, 4, 2, 27, 14, 5, 20, 9,

22, 18, 11, 3, 25, 7, 15, 6, 26, 19, 12, 1,

40, 51, 30, 37, 46, 54, 29, 39, 50, 44, 32, 47,

43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28, 31

};

for (int i = 0; i < 28; i++) //Получение промежуточного 56-битового ключа в виде двух ключей по 28 бит

{

tmpKey1[i] = key[i];

tmpKey2[i] = key[i+28];

}

for (int i = 0; i < 16; i++) //Получение 16 ключей раундов

{

if (i == 0 | i == 1 | i == 8 | i == 15) //сдвиг каждого подключа

{

tmpKey1 <<= 1;

tmpKey2 <<= 1;

}

else {

tmpKey1 <<= 2;

tmpKey2 <<= 2;

}

for (int j = 0; j < 28; j++) //объединение двух подключей в один 56-битный ключ

{

tmpKEY56[j] = tmpKey1[j];

tmpKEY56[28+j] = tmpKey2[j];

}

for (int j = 0; j < 48; j++) //конечная перестановка для получения раундового ключа

{

RoundKey[i].set(j, tmpKEY56.test(RP[j]));

}

}

}

int DES::conv\_to\_dec(int count)

{

switch (count)

{

case 0:

return 0;

case 1:

return 1;

case 10:

return 2;

case 11:

return 3;

case 100:

return 4;

case 101:

return 5;

case 110:

return 6;

case 111:

return 7;

case 1000:

return 8;

case 1001:

return 9;

case 1010:

return 10;

case 1011:

return 11;

case 1100:

return 12;

case 1101:

return 13;

case 1110:

return 14;

case 1111:

return 15;

default:

cout << "\n\t ERROR IN CONV\_TO\_DEC FUNCTION!\n";

system("pause");

exit(100);

}

}

int64\_t DES::getSizeFile(string path\_)

{

fstream file(path\_);

file.seekg(0, file.end);

streamsize size = file.tellg();

file.seekg(0, std::ios::beg);

file.close();

return size / 1024 + 1;

}

string DES::convert\_string(string& hex)

{

string bin;

for (int i = 0; i != hex.length(); i++)

{

bin += hex\_char\_to\_bin(hex[i]);

}

if (bin.size() > 64)

cout << "\nWARNING! The size of the entered key exceeds 64 bits. The key size will be trimmed to a possible 64!\n";

return bin;

}

const char\* DES::hex\_char\_to\_bin(char ch)

{

ch = toupper(ch);

if ((ch < 48 || ch > 57) && (ch < 65 || ch > 90))

{

cout << "\nINPUT ERROR: the symbol is not included in the hexadecimal system!\n";

system("pause");

exit(10);

}

switch (toupper(ch))

{

case '0': return "0000";

case '1': return "0001";

case '2': return "0010";

case '3': return "0011";

case '4': return "0100";

case '5': return "0101";

case '6': return "0110";

case '7': return "0111";

case '8': return "1000";

case '9': return "1001";

case 'A': return "1010";

case 'B': return "1011";

case 'C': return "1100";

case 'D': return "1101";

case 'E': return "1110";

case 'F': return "1111";

}

}

DES.h

#pragma once

#include <fstream>

#include <iostream>

#include <string>

#include <bitset>

#include <vector>

#include <windows.h>

#include <chrono>

using namespace std;

class DES

{

public:

void encrypt();

void decrypt();

chrono::milliseconds exec\_time\_dec, exec\_time\_enc;

int64\_t sizeEncFile = 0, sizeDecFile = 0, sizeSourceFile = 0;

private:

int64\_t fileSize = 0;

bitset<64> data;

bitset<64> key;

bitset<48> RoundKey[16];

bitset<32> leftBlock, rightBlock;

string path, Enc\_filename;

void IP\_first();

void IP\_second();

bitset<48> EP(bitset<32>& block);

void key\_extension();

int conv\_to\_dec(int tmp);

void init\_file();

void round(int j, bool flag);

void apply\_Sbox(bitset<6>\* Sblock6, bitset<4>\* Sblock4);

bitset<32> block\_convertion(bitset<32> BLOCK, bitset<48> R\_key);

int64\_t getSizeFile(string path\_);

string convert\_string(string& hex);

const char\* hex\_char\_to\_bin(char ch);

bool key\_flag = false;

};

Код LOKI97

Main.c

#include <iostream>

#include "LOKI.h"

int main()

{

LOKI loki;

loki.encrypt();

loki.decrypt();

cout << "\nEncryption time: " << loki.exec\_time\_enc.count() << " millisec | File size: " << loki.sizeEncFile << " KByte\n";

cout << "\nDecryption time: " << loki.exec\_time\_dec.count() << " millisec | File size: " << loki.sizeDecFile << " KByte\n";

system("pause");

return 0;

}

LOKI.cpp

#include "LOKI.h"

void LOKI::init\_file()

{

ifstream in;

path = "data.pdf";

int choise = 1;

//открытие файла для чтения иходного текста и файла для записи шифротекста

cout << "1 - Use default path (C://Users/.../data.pdf) \n2 - Enter path\n=>";

cin >> choise;

if (choise == 1)

in.open(path, std::ios::binary);

else {

cout << "\nEnter path data: ";

cin >> path;

in.open(path, std::ios::binary);

}

if (!in.is\_open())

{

cout << "\nFile cannot open!" << endl;

system("pause");

exit(-10);

}

in.close();

sizeSourceFile = getSizeFile(path);

cout << "\nEnter key(hex format): ";

string hex\_str\_key, bin\_str\_key;

try {

cin >> hex\_str\_key;

bin\_str\_key = convert\_string(hex\_str\_key);

key = bitset<256>(string(bin\_str\_key));

}

catch (std::exception &ex)

{

cout << "\nINPUT ERROR: " << ex.what() << endl;

system("pause");

exit(-15);

}

}

void LOKI::encrypt()

{

init\_file();

auto start\_time = chrono::steady\_clock::now();

Enc\_filename = path;

Enc\_filename.insert(0, "ENC\_");

ifstream in(path, std::ios::binary);

ofstream output(Enc\_filename, std::ios::binary);

if (!key\_flag)

{

key\_flag = true;

key\_extension();

}

//получать и шифровать блоки пока не закончится исходный файл

while (!in.eof())

{

in.read((char\*)&data, sizeof(BLOCK\_128));

for (int i = 0; i < 16; i++)

round(i, ENCR);

output.write((char\*)&data, sizeof(BLOCK\_128));

}

sizeEncFile = getSizeFile(Enc\_filename);

auto end\_time = chrono::steady\_clock::now();

exec\_time\_enc = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time);

}

void LOKI::decrypt()

{

auto start\_time = chrono::steady\_clock::now();

path.insert(0, "DEC\_");

ifstream in(Enc\_filename, std::ios::binary);

ofstream output(path, std::ios::binary);

if (!key\_flag)

{

key\_flag = true;

key\_extension();

}

//получать и шифровать блоки пока не закончится исходный файл

while (!in.eof())

{

in.read((char\*)&data, sizeof(BLOCK\_128));

for (int i = 0; i < 16; i++)

round(i, DECR);

output.write((char\*)&data, sizeof(BLOCK\_128));

}

sizeDecFile = getSizeFile(path);

auto end\_time = chrono::steady\_clock::now();

exec\_time\_dec = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time);

}

void LOKI::round(int j, bool flag)

{

//разделение блока на правый и левый подблок

for (int i = 0; i < 64; i++)

{

leftBlock[i] = data[i];

rightBlock[i] = data[i + 64];

}

if (flag == ENCR)

{

rightBlock ^= RoundKey[j \* 3];

leftBlock ^= F(rightBlock, RoundKey[(j \* 3) + 1]);

rightBlock ^= RoundKey[(j \* 3) + 2];

}

if (flag == DECR)

{

leftBlock ^= RoundKey[47 - (j \* 3)];

rightBlock ^= F(leftBlock, RoundKey[47 - ((j \* 3) + 1)]);

leftBlock ^= RoundKey[47 - ((j \* 3) + 2)];

}

//объединение правого и левого блока в один блок

for (int i = 0; i < 64; i++)

{

data[i] = rightBlock[i];

data[i + 64] = leftBlock[i];

}

}

void LOKI::key\_extension()

{

BLOCK\_64 key\_subBlock[4], temp;

check\_key();

//разделяем ключ на 4 блока по 64 бит

for (int i = 0; i < 64; i++)

{

key\_subBlock[0][i] = key[i];

key\_subBlock[1][i] = key[i + 64];

key\_subBlock[2][i] = key[i + 128];

key\_subBlock[3][i] = key[i + 192];

}

for (int i = 0; i < 48; i++)

{

temp = 0x9E3779B97F4A7C15 \* i;

temp ^= key\_subBlock[1] ^ key\_subBlock[2];

key\_subBlock[0] ^= F(temp, key\_subBlock[3]);

if (i != 47)

{

temp = key\_subBlock[0];

key\_subBlock[0] = key\_subBlock[1];

key\_subBlock[1] = key\_subBlock[2];

key\_subBlock[2] = key\_subBlock[3];

key\_subBlock[3] = temp;

}

RoundKey[i] = key\_subBlock[0];

}

}

BLOCK\_64 LOKI::F(BLOCK\_64 block1, BLOCK\_64 block2)

{

BLOCK\_96 ep\_block;

BLOCK\_32 subKey1, subKey2;

for (int i = 0; i < 32; i++)

{

subKey1[i] = block2[i];

subKey2[i] = block2[i + 32];

}

block1 = KP(block1, subKey2);

ep\_block = E(block1);

block1 = apply\_Sboxes\_layer1(ep\_block);

block1 = P(block1);

block1 = apply\_Sboxes\_layer2(block1, subKey1);

return block1;

}

//key permutation перестановка в функции F

BLOCK\_64 LOKI::KP(BLOCK\_64 block, BLOCK\_32 key)

{

BLOCK\_32 subBlock1, subBlock2;

int temp;

for (int i = 0; i < 32; i++)

{

subBlock1[i] = block[i];

subBlock2[i] = block[i + 32];

}

//если бит ключа == 1, свапаем бит в этой позиции в подблоках данных

for (int i = 0; i < 32; i++)

{

if (key.test(i) == true)

{

temp = subBlock1[i];

subBlock1[i] = subBlock2[i];

subBlock2[i] = temp;

}

}

for (int i = 0; i < 32; i++)

{

block[i] = subBlock1[i];

block[i + 32] = subBlock2[i];

}

return block;

}

//expanding перестановка в F функции

BLOCK\_96 LOKI::E(BLOCK\_64 block)

{

BLOCK\_96 result;

int it = 0;

for (int i = 0; i < 9; i++)

{

switch (i)

{

case 0:

for (int j = 4; j >= 0; j--)

{

result[it] = block[i];

it++;

}

continue;

case 1:

for (int j = 63; j >= 56; j--)

{

result[it] = block[i];

it++;

}

continue;

case 2:

for (int j = 58; j >= 48; j--)

{

result[it] = block[i];

it++;

}

continue;

case 3:

for (int j = 52; j >= 40; j--)

{

result[it] = block[i];

it++;

}

continue;

case 4:

for (int j = 42; j >= 32; j--)

{

result[it] = block[i];

it++;

}

continue;

case 5:

for (int j = 34; j >= 24; j--)

{

result[it] = block[i];

it++;

}

continue;

case 6:

for (int j = 28; j >= 16; j--)

{

result[it] = block[i];

it++;

}

continue;

case 7:

for (int j = 18; j >= 8; j--)

{

result[it] = block[i];

it++;

}

continue;

case 8:

for (int j = 12; j >= 0; j--)

{

result[it] = block[i];

it++;

}

continue;

default:

exit(-100);

}

}

return result;

}

//рименение S-боксов 1 слоя

BLOCK\_64 LOKI::apply\_Sboxes\_layer1(BLOCK\_96 ep\_block)

{

BLOCK\_64 result;

bitset<13> S13;

bitset<11> S11;

bitset<8> S8[8];

WORD word;

int it = 0;

//разбиение и применение S-боксов

for (int i = 0; i < 8; i++)

{

switch (i)

{

case 0:

for (int i = 0; i < 13; i++)

S13[i] = ep\_block[i];

word = S13.to\_ulong();

S8[0] = apply\_S1(word);

continue;

case 1:

it = 0;

for (int i = 13; i < 24; i++)

{

S11[it] = ep\_block[i];

it++;

}

word = S11.to\_ulong();

S8[1] = apply\_S2(word);

continue;

case 2:

it = 0;

for (int i = 24; i < 37; i++)

{

S13[it] = ep\_block[i];

it++;

}

word = S13.to\_ulong();

S8[2] = apply\_S1(word);

continue;

case 3:

it = 0;

for (int i = 37; i < 48; i++)

{

S11[it] = ep\_block[i];

it++;

}

word = S11.to\_ulong();

S8[1] = apply\_S2(word);

continue;

case 4:

it = 0;

for (int i = 48; i < 59; i++)

{

S11[it] = ep\_block[i];

it++;

}

word = S11.to\_ulong();

S8[1] = apply\_S2(word);

continue;

case 5:

it = 0;

for (int i = 59; i < 72; i++)

{

S13[it] = ep\_block[i];

it++;

}

word = S13.to\_ulong();

S8[5] = apply\_S1(word);

continue;

case 6:

it = 0;

for (int i = 72; i < 83; i++)

{

S11[it] = ep\_block[i];

it++;

}

word = S11.to\_ulong();

S8[1] = apply\_S2(word);

continue;

case 7:

it = 0;

for (int i = 83; i < 96; i++)

{

S13[it] = ep\_block[i];

it++;

}

word = S13.to\_ulong();

S8[7] = apply\_S1(word);

continue;

}

}

//объединение 8-битных S-box в один блок 64 бита

for (int i = 0; i < 8; i++)

{

result[i] = S8[0][i];

result[i+8] = S8[1][i];

result[i+16] = S8[2][i];

result[i+24] = S8[3][i];

result[i+32] = S8[4][i];

result[i+40] = S8[5][i];

result[i+48] = S8[6][i];

result[i+56] = S8[7][i];

}

return result;

}

WORD LOKI::my\_pow(WORD word, int n)

{

for (int i = 0; i < n; i++)

{

word = word \* word;

}

return word;

}

//перестановка после первого слоя S-боксов

BLOCK\_64 LOKI::P(BLOCK\_64 block)

{

int Perm[] = { 56, 48, 40, 32, 24, 16, 8, 0, 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, 28, 20, 12, 4, 61, 53, 45, 37, 29, 21, 13, 5,

62, 54, 46, 38, 30, 22, 14, 6, 63, 55, 47, 39, 31, 23, 15, 7 };

BLOCK\_64 temp = block;

for (int i = 0; i < 64; i++)

block[i] = temp[Perm[i]];

return block;

}

//рименение S-боксов 2 слоя

BLOCK\_64 LOKI::apply\_Sboxes\_layer2(BLOCK\_64 block, BLOCK\_32 key)

{

BLOCK\_64 result;

bitset<3> S3[4];

bitset<5> S5[4];

bitset<13> S13;

bitset<11> S11;

bitset<8> S8[8];

WORD word;

int it = 0;

//разделяем блок на 8 восьмибитных подблоков

for (int i = 0; i < 8; i++)

{

S8[0][i] = block[i];

S8[1][i] = block[i + 8];

S8[2][i] = block[i + 16];

S8[3][i] = block[i + 24];

S8[4][i] = block[i + 32];

S8[5][i] = block[i + 40];

S8[6][i] = block[i + 48];

S8[7][i] = block[i + 56];

}

for (int i = 0; i < 8; i++)

{

switch (i)

{

case 0:

it = 0;

for (int j = 0; j < 3; j++)

{

S11[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S11[it] = S8[0][i];

it++;

}

word = S11.to\_ulong();

S8[0] = apply\_S1(word);

continue;

case 1:

it = 0;

for (int j = 3; j < 6; j++)

{

S11[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S11[it] = S8[1][i];

it++;

}

word = S11.to\_ulong();

S8[1] = apply\_S1(word);

continue;

case 2:

it = 0;

for (int j = 6; j < 11; j++)

{

S13[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S13[it] = S8[2][i];

it++;

}

word = S13.to\_ulong();

S8[2] = apply\_S2(word);

continue;

case 3:

it = 0;

for (int j = 11; j < 16; j++)

{

S13[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S13[it] = S8[3][i];

it++;

}

word = S13.to\_ulong();

S8[3] = apply\_S2(word);

continue;

case 4:

it = 0;

for (int j = 16; j < 19; j++)

{

S11[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S11[it] = S8[4][i];

it++;

}

word = S11.to\_ulong();

S8[4] = apply\_S1(word);

continue;

case 5:

it = 0;

for (int j = 19; j < 22; j++)

{

S11[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S11[it] = S8[5][i];

it++;

}

word = S11.to\_ulong();

S8[5] = apply\_S1(word);

continue;

case 6:

it = 0;

for (int j = 22; j < 27; j++)

{

S13[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S13[it] = S8[6][i];

it++;

}

word = S13.to\_ulong();

S8[6] = apply\_S2(word);

continue;

case 7:

it = 0;

for (int j = 27; j < 32; j++)

{

S13[it] = key[j];

it++;

}

for (int i = 0; i < 8; i++)

{

S13[it] = S8[7][i];

it++;

}

word = S13.to\_ulong();

S8[7] = apply\_S2(word);

continue;

}

}

//объединение 8-битных S-box в один блок 64 бита

for (int i = 0; i < 8; i++)

{

result[i] = S8[0][i];

result[i + 8] = S8[1][i];

result[i + 16] = S8[2][i];

result[i + 24] = S8[3][i];

result[i + 32] = S8[4][i];

result[i + 40] = S8[5][i];

result[i + 48] = S8[6][i];

result[i + 56] = S8[7][i];

}

return result;

}

BLOCK\_8 LOKI::apply\_S1(WORD word)

{

BLOCK\_8 result;

word ^= 0x1FFF;

word = my\_pow(word, 3);

result = word % 0x2911;

result &= 0xFF;

return result;

}

BLOCK\_8 LOKI::apply\_S2(WORD word)

{

BLOCK\_8 result;

word ^= 0x7FF;

word = my\_pow(word, 3);

result = word % 0xAA7;

result &= 0xFF;

return result;

}

int64\_t LOKI::getSizeFile(string path\_)

{

fstream file(path\_);

file.seekg(0, file.end);

streamsize size = file.tellg();

file.seekg(0, std::ios::beg);

file.close();

return size/1024 + 1;

}

void LOKI::check\_key()

{

int pos = -1;

BLOCK\_64 temp[4];

for (int i = 255; i >= 0; i--)

{

if (key.test(i) == 1)

{

pos = i;

break;

}

}

if (pos > 127 && pos < 192)

{

for (int i = 0; i < 64; i++)

{

temp[0][i] = key[i];

temp[1][i] = key[i + 64];

temp[2][i] = key[i + 128];

}

temp[3] = F(temp[0], temp[1]);

for (int i = 0; i < 64; i++)

{

key[i] = temp[0][i];

key[i + 64] = temp[1][i];

key[i + 128] = temp[2][i];

key[i + 192] = temp[3][i];

}

return;

}

else if (pos < 128)

{

for (int i = 0; i < 64; i++)

{

temp[0][i] = key[i];

temp[1][i] = key[i + 64];

}

temp[2] = F(temp[1], temp[0]);

temp[3] = F(temp[0], temp[1]);

for (int i = 0; i < 64; i++)

{

key[i] = temp[0][i];

key[i + 64] = temp[1][i];

key[i + 128] = temp[2][i];

key[i + 192] = temp[3][i];

}

return;

}

else

return;

}

string LOKI::convert\_string(string &hex)

{

string bin;

for (int i = 0; i != hex.length(); i++)

{

bin += hex\_char\_to\_bin(hex[i]);

}

if (bin.size() > 256)

cout << "\nWARNING! The size of the entered key exceeds 256 bits. The key size will be trimmed to a possible 256!\n";

return bin;

}

const char\* LOKI::hex\_char\_to\_bin(char ch)

{

ch = toupper(ch);

if ((ch < 48 || ch > 57) && (ch < 65 || ch > 90))

{

cout << "\nINPUT ERROR: the symbol is not included in the hexadecimal system!\n";

system("pause");

exit(10);

}

switch (toupper(ch))

{

case '0': return "0000";

case '1': return "0001";

case '2': return "0010";

case '3': return "0011";

case '4': return "0100";

case '5': return "0101";

case '6': return "0110";

case '7': return "0111";

case '8': return "1000";

case '9': return "1001";

case 'A': return "1010";

case 'B': return "1011";

case 'C': return "1100";

case 'D': return "1101";

case 'E': return "1110";

case 'F': return "1111";

}

}

LOKI.h

#pragma once

#include <iostream>

#include <fstream>

#include <bitset>

#include <string>

#include <stdint.h>

#include <chrono>

#define BLOCK\_256 bitset<256>

#define BLOCK\_128 bitset<128>

#define BLOCK\_96 bitset<96>

#define BLOCK\_64 bitset<64>

#define BLOCK\_32 bitset<32>

#define BLOCK\_8 bitset<8>

#define ENCR true

#define DECR false

typedef int64\_t QWORD;

typedef int16\_t WORD;

using namespace std;

class LOKI

{

public:

void encrypt();

void decrypt();

chrono::milliseconds exec\_time\_dec, exec\_time\_enc;

int64\_t sizeEncFile = 0, sizeDecFile = 0, sizeSourceFile = 0;

private:

BLOCK\_128 data;

BLOCK\_256 key;

BLOCK\_64 RoundKey[48],

leftBlock, rightBlock;

string path, Enc\_filename;

void init\_file();

void round(int j, bool flag);

void key\_extension();

void check\_key();

BLOCK\_64 KP(BLOCK\_64 block, BLOCK\_32 key);

BLOCK\_64 F(BLOCK\_64 block1, BLOCK\_64 block2);

BLOCK\_96 E(BLOCK\_64);

BLOCK\_64 apply\_Sboxes\_layer1(BLOCK\_96 ep\_block);

BLOCK\_64 apply\_Sboxes\_layer2(BLOCK\_64 block, BLOCK\_32 key);

BLOCK\_8 apply\_S1(WORD word);

BLOCK\_8 apply\_S2(WORD word);

BLOCK\_64 P(BLOCK\_64 block);

WORD my\_pow(WORD word, int n);

int64\_t getSizeFile(string path);

string convert\_string(string &hex);

const char\* hex\_char\_to\_bin(char ch);

bool key\_flag = false;

};