Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №1

По дисциплине «Методы защиты информации»

По теме «Симметричная криптография. Двойной и тройной DES»

Выполнил:

Студент гр. 653501

Никитинская А. С.

Проверил:

Артемьев В. С.

Минск 2019

Содержание

[1. Постановка задачи 3](#_Toc20490052)

[2. Краткие теоретические сведения 4](#_Toc20490053)

[3. Блок-схемы алгоритма DES 10](#_Toc20490057)

[Вывод 11](#_Toc20490058)

[Приложение 1. Исходный код программы 12](#_Toc20490059)

[Приложение 2. Скриншот работы алгоритма 19](#_Toc20490060)

## 1. Постановка задачи

1) Изучить теоретические сведения.

2) Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритмов двойной и тройной DES.

## Краткие теоретические сведения

2.1 Алгоритм шифрования DES

# 2.1.1 Основные сведения

Одной из наиболее известных криптографических систем с закрытым ключом является DES – Data Encryption Standard. Эта система первой получила статус государственного стандарта в области шифрования данных. Она разработана специалистами фирмы IBM и вступила в действие в США 1977 году. Алгоритм DES по-прежнему широко применяется и заслуживает внимания при изучении блочных шифров с закрытым ключом.

Стандарт DES построен на комбинированном использовании перестановки, замены и гаммирования. Шифруемые данные должны быть представлены в двоичном виде.

*DES* является классической *сетью Фейстеля* с двумя ветвями. Данные шифруются 64-битными блоками, используя 56-битный ключ. Алгоритм преобразует за несколько *раундов* 64-битный вход в 64-битный выход. Длина ключа равна 56 битам. Процесс шифрования состоит из четырех этапов. На первом из них выполняется начальная перестановка (*IP*) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей. Следующий этап состоит из 16 *раундов* одной и той же функции, которая использует операции сдвига и подстановки. На третьем этапе левая и правая половины выхода последней (16-й) итерации меняются местами. Наконец, на четвертом этапе выполняется перестановка IP-1 результата, полученного на третьем этапе. Перестановка IP-1 инверсна начальной перестановке.

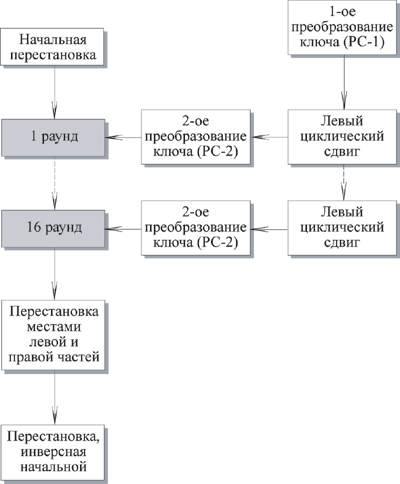


Рисунок 1 Общая схема DES

# 2.1.2 Шифрование

Начальная перестановка

Начальная перестановка и ее инверсия определяются стандартной таблицей. Если М- это произвольные 64 бита, то X = IP (M)-переставленные 64 бита. Если применить обратную функцию перестановки Y = IP-1 (X) = IP-1 (IP(M)), то получится первоначальная последовательность бит.

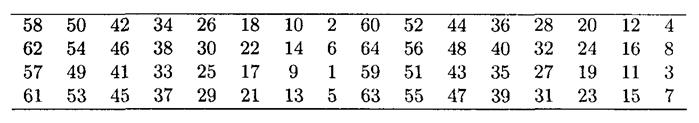


Рисунок 2 - DES. Начальная перестановка

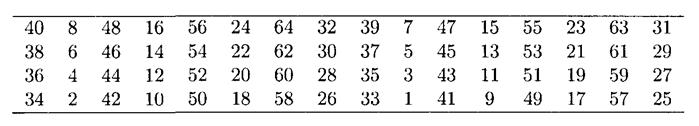


Рисунок 3 - DES. Заключительная перестановка

Последовательность преобразований отдельного раунда

Теперь рассмотрим последовательность преобразований, используемую в каждом *раунде*.

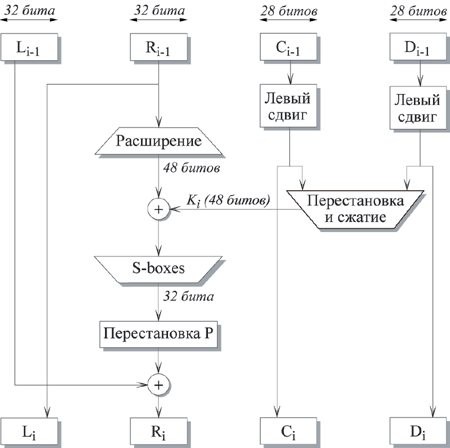


Рисунок 4 - I-ый раунд DES

64-битный входной блок проходит через 16 *раундов*, при этом на каждой итерации получается промежуточное 64-битное значение. Левая и правая части каждого промежуточного значения трактуются как отдельные 32-битные значения, обозначенные *L* и *R*. Каждую итерацию можно описать следующим образом:

Li = Ri-1

Ri = Li-1 F(Ri-1, Ki)

Где обозначает операцию XOR.

Таким образом, выход левой половины *Li* равен входу правой половины *Ri-1*. Выход правой половины *Ri*является результатом применения операции XOR к *Li-1* и функции *F*, зависящей от *Ri-1* и *Ki*.

Рассмотрим функцию *F* более подробно.

*Ri*, которое подается на вход функции *F*, имеет длину 32 бита. Вначале Ri расширяется до 48 бит, используя таблицу, которая определяет перестановку плюс расширение на 16 бит. Расширение происходит следующим образом. 32 бита разбиваются на группы по 4 бита и затем расширяются до 6 бит, присоединяя крайние биты из двух соседних групп. Например, если часть входного сообщения

. . . efgh ijkl mnop . . .

то в результате расширения получается сообщение

. . . defghi hijklm lmnopq . . .

После этого для полученного 48-битного значения выполняется операция XOR с 48-битным *подключом Ki*. Затем полученное 48-битное значение подается на вход функции подстановки, результатом которой является 32-битное значение.

Подстановка состоит из восьми *S-boxes,* каждый из которых на входе получает 6 бит, а на выходе создает 4 бита. Эти преобразования определяются специальными таблицами. Первый и последний биты входного значения *S-box* определяют номер строки в таблице, средние 4 бита определяют номер столбца. Пересечение строки и столбца определяет 4-битный выход. Например, если входом является 011011, то номер строки равен 01 (строка 1) и номер столбца равен 1101 (столбец 13). Значение в строке 1 и столбце 13 равно 5, т.е. выходом является 0101.

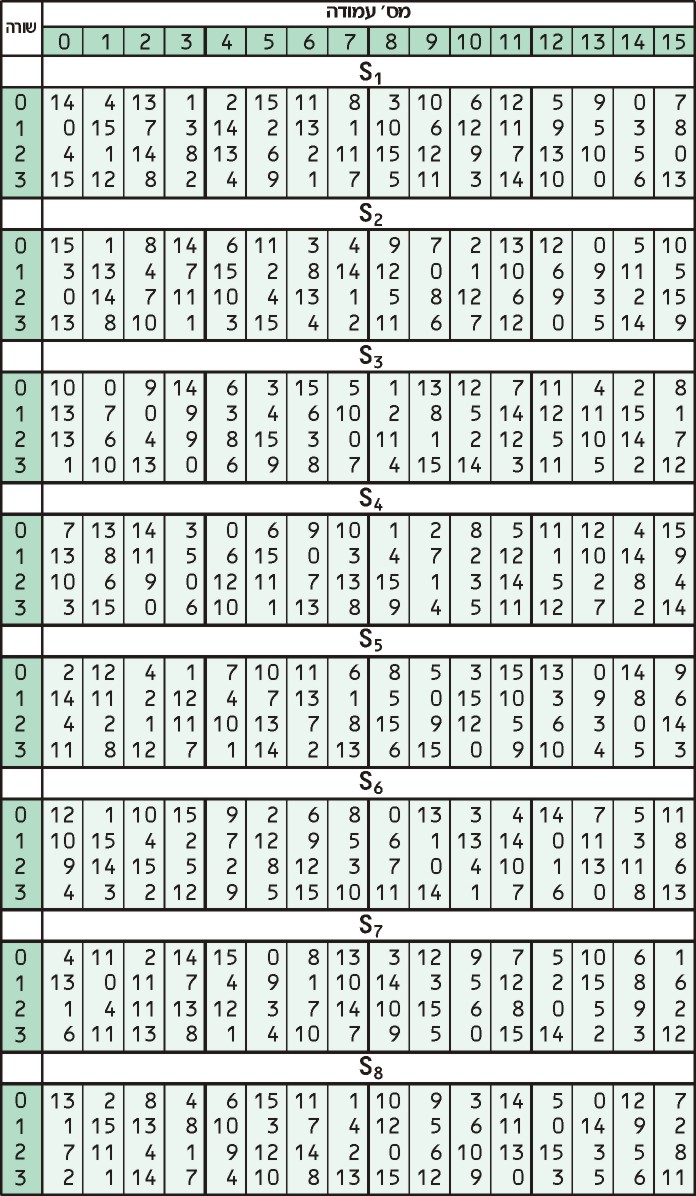


Рисунок 5 - S-boxes

Далее полученное 32-битное значение обрабатывается с помощью перестановки *Р*, целью которой является максимальное переупорядочивание бит, чтобы в следующем *раунде* шифрования с большой вероятностью каждый бит обрабатывался другим *S-box*.

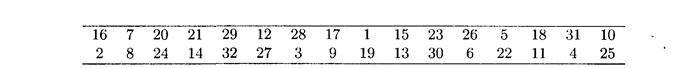


Рисунок 6 - Перестановка с помощью Р-блоков

Создание подключей

Ключ для отдельного *раунда Ki*состоит из 48 бит. Ключи *Ki* получаются по следующему алгоритму. Для 56-битного ключа, используемого на входе алгоритма (если используется 64-битный ключ, то, как видно из рис. 5 убираются биты 64, 56, 48, 40, 32, 16, 8) , вначале выполняется перестановка в соответствии с таблицей Permuted Choice 1 (РС-1).

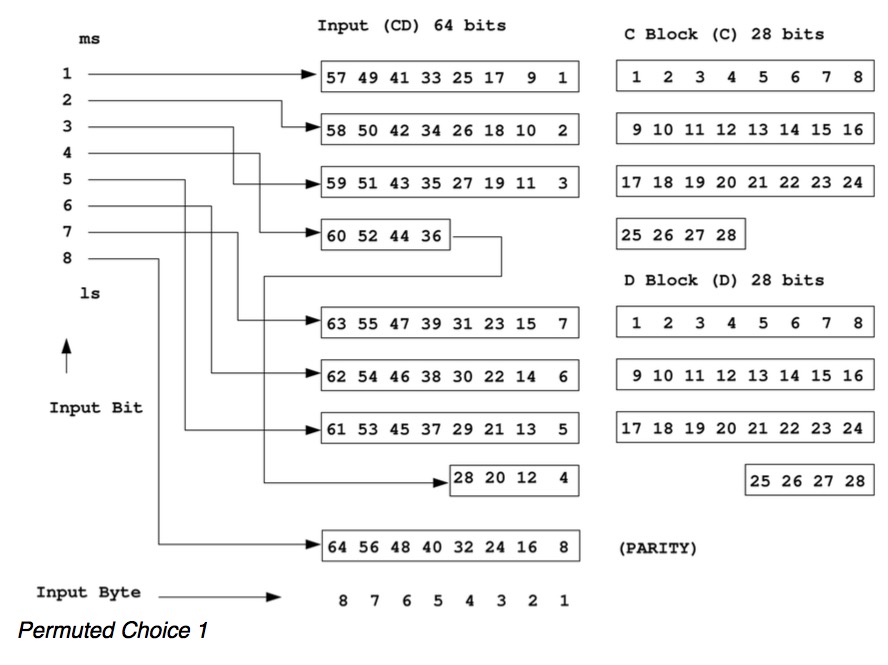


Рисунок 7 - Схема Permuted Choice

Полученный 56-битный ключ разделяется на две 28-битные части, обозначаемые как C0 и D0 соответственно. На каждом *раунде Ci* и *Di*независимо циклически сдвигаются влево на 1 или 2 бита, в зависимости от номера *цикла*.

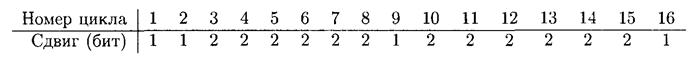


Рисунок 8 - Сдвиг ключа в зависимости от номера цикла

Полученные значения являются входом следующего *раунда*. Они также представляют собой вход в Permuted Choice 2 (РС-2), который создает 48-битное выходное значение, являющееся входом функции *F*(*Ri-1*, *Ki*).

# 2.1.3 Дешифрование

Процесс дешифрования аналогичен процессу шифрования. На входе алгоритма используется зашифрованный текст, но ключи *Ki* используются в обратной последовательности. *K16* используется на первом *раунде*, *K1* используется на последнем *раунде*.

2.1.4 Двойной DES

Простейший способ увеличить длину ключа состоит в повторном применении DES с двумя разными ключами. Используя незашифрованное сообщение P) 64-битного исходного текста и два ключа K1 и K2, зашифрованное сообщение С можно получить следующим образом:

C = Ek2 [Ek1 [P]]

При дешифровании два ключа применяются в обратном порядке:

P = Dk1 [Dk2 [C]]

В этом случае длина ключа равна 56 \* 2 = 112 бит.

2.1.5 Тройной DES

Очевидное противодействие атаке "встреча посередине" состоит в использовании третьей стадии шифрования с тремя различными ключами. Это поднимает стоимость лобовой атаки до 2168, которая на сегодняшний день считается выше практических возможностей. Но при этом длина ключа равна 56 \* 3 = 168 бит, что иногда бывает громоздко. В качестве альтернативы предлагается метод тройного шифрования, использующий только два ключа. В этом случае выполняется последовательность зашифрование-расшифрование-зашифрование (IP) 64-битного исходного текстаEDE).

C = EK1 [DK2 [EK1 [P) 64-битного исходного текста]]]

Не имеет большого значения, что используется на второй стадии:

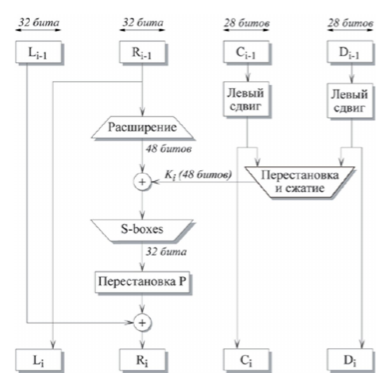
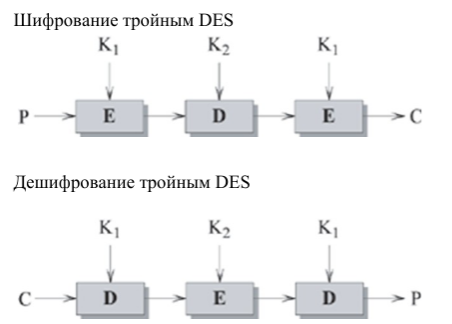
шифрование или дешифрование. В случае использования дешифрования существует только то преимущество, что можно тройной DES свести к обычному одиночному DES, используя K1 = K2:

C = EK1 [DK1 [EK1 [P]]] = EK1 [P]

Известных криптографических атак на тройной DES не существует.

Цена подбора ключа в тройном DES равна 2112

## 3. Блок-схемы алгоритма DES



## Вывод

В ходе написания лабораторной работы были изучены алгоритмы шифрования и дешифрования DES, а также написаны их программные реализации. Были получены навыки усложнения и увеличения криптостойкости алгоритма DES.

## Приложение 1. Исходный код программы

// Initial permut matrix for the datas

const vector<int> PI = {

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

};

// Initial permut made on the key

const vector<int> CP\_1 = {

57, 49, 41, 33, 25, 17, 9,

1, 58, 50, 42, 34, 26, 18,

10, 2, 59, 51, 43, 35, 27,

19, 11, 3, 60, 52, 44, 36,

63, 55, 47, 39, 31, 23, 15,

7, 62, 54, 46, 38, 30, 22,

14, 6, 61, 53, 45, 37, 29,

21, 13, 5, 28, 20, 12, 4

};

// Permut applied on shifted key to get Ki+1

const vector<int> CP\_2 = {

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

};

// Expand matrix to get a 48bits matrix of datas to apply the xor with Ki

const vector<int> E = {

32, 1, 2, 3, 4, 5,

4, 5, 6, 7, 8, 9,

8, 9, 10, 11, 12, 13,

12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21,

20, 21, 22, 23, 24, 25,

24, 25, 26, 27, 28, 29,

28, 29, 30, 31, 32, 1

};

// SBOX

const vector<vector<vector<int>>> S\_BOX = {

{

{14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},

{0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},

{4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},

{15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}

},

{

{15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},

{3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},

{0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},

{13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9},

},

{

{10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},

{13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},

{13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},

{1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12},

},

{

{7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},

{13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},

{10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},

{3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14},

},

{

{2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},

{14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},

{4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},

{11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3},

},

{

{12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},

{10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},

{9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},

{4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13},

},

{

{4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},

{13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},

{1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},

{6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12},

},

{

{13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},

{1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},

{7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},

{2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11},

}

};

// Permut made after each SBox substitution for each round

const vector<int> P = {

16, 7, 20, 21, 29, 12, 28, 17,

1, 15, 23, 26, 5, 18, 31, 10,

2, 8, 24, 14, 32, 27, 3, 9,

19, 13, 30, 6, 22, 11, 4, 25

};

// Final permut for datas after the 16 rounds

const vector<int> PI\_1 = {

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

};

// Matrix that determine the shift for each round of keys

const vector<int> SHIFT = {

1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1

};

const int DES\_ENCRYPT = 1;

const int DES\_DECRYPT = 0;

const int KEY\_SIZE = 8;

class Des {

private:

string password;

string text;

vector<vector<int>> keys;

static string add\_padding(string text) {

auto pad\_len = KEY\_SIZE - (text.size() % 8);

auto pad\_char = static\_cast<char>(pad\_len);

string result = text;

for (auto i = 0; i < pad\_len; i++) {

result += pad\_char;

}

return result;

}

static string remove\_padding(string text) {

auto pad\_len = static\_cast<int>(text[text.size() - 1]);

string result = "";

for (auto i = 0; i < text.size() - pad\_len; i++) {

result += text[i];

}

return result;

}

static vector<int> permut(vector<int> block, vector<int> table) {

auto result = vector<int>();

for (auto i = 0; i < table.size(); i++) {

auto item = table[i];

result.push\_back(block[item - 1]);

}

return result;

}

static vector<vector<int>> shift(vector<int> g, vector<int> d, int n) {

auto result = vector<vector<int>>();

result.push\_back(Utils::vector\_concat(

Utils::vector\_slice(g, n, g.size()),

Utils::vector\_slice(g, 0, n)

));

result.push\_back(Utils::vector\_concat(

Utils::vector\_slice(d, n, d.size()),

Utils::vector\_slice(d, 0, n)

));

return result;

}

static vector<int> expand(vector<int> block, vector<int> table) {

return Des::permut(block, table);

}

void generate\_keys() {

this->keys = vector<vector<int>>();

auto key = Utils::string\_to\_bit\_array(this->password);

key = Des::permut(key, CP\_1);

auto gd = Utils::n\_split(key, 28);

auto g = gd[0];

auto d = gd[1];

for (auto i = 0; i < 16; i++) {

auto tmpGD = Des::shift(g, d, SHIFT[i]);

g = tmpGD[0];

d = tmpGD[1];

auto tmp = Utils::vector\_concat(g, d);

this->keys.push\_back(Des::permut(tmp, CP\_2));

}

}

static vector<int> substitute(vector<int> d\_e) {

auto sub\_blocks = Utils::n\_split(d\_e, 6);

auto result = vector<int>();

for (auto i = 0; i < sub\_blocks.size(); i++) {

auto block = sub\_blocks[i];

auto block\_row = std::to\_string(block[0]) + std::to\_string(block[5]);

auto row = std::stoi(block\_row, nullptr, 2);

string block\_column = "";

auto tmp = Utils::vector\_slice(block, 1, -1);

for (auto j = 0; j < tmp.size(); j++) {

block\_column += std::to\_string(tmp[j]);

}

auto column = std::stoi(block\_column, nullptr, 2);

auto val = S\_BOX[i][row][column];

auto bin = Utils::int\_bin\_value(val, 4);

for (auto j = 0; j < bin.size(); j++) {

auto bit = bin[j] == '1' ? 1 : 0;

result.push\_back(bit);

}

}

return result;

}

public:

Des(string password, string text) {

this->password = std::move(password);

this->text = std::move(text);

}

string run(int action = 1, bool padding = false) {

if (this->password.size() < KEY\_SIZE) {

throw InvalidKetSize();

} else if (this->password.size() > 8) {

this->password = this->password.substr(0, KEY\_SIZE);

}

if (padding && action == DES\_ENCRYPT) {

this->text = Des::add\_padding(this->text);

} else if (this->text.size() % 8 != 0) {

throw InvalidTextSize();

}

this->generate\_keys();

auto text\_blocks = Utils::n\_split\_string(this->text, 8);

auto result = vector<int>();

for (auto b = 0; b < text\_blocks.size(); b++) {

auto item = text\_blocks[b];

auto block = Utils::string\_to\_bit\_array(item);

block = Des::permut(block, PI);

auto gd = Utils::n\_split(block, 32);

auto g = gd[0];

auto d = gd[1];

auto tmp = vector<int>();

for (auto i = 0; i < 16; i++) {

auto d\_e = Des::expand(d, E);

if (action == DES\_ENCRYPT) {

tmp = Utils::vector\_xor(this->keys[i], d\_e);

} else {

tmp = Utils::vector\_xor(this->keys[15 - i], d\_e);

}

tmp = Des::substitute(tmp);

tmp = Des::permut(tmp, P);

tmp = Utils::vector\_xor(g, tmp);

g = d;

d = tmp;

}

result = Utils::vector\_concat(result, Des::permut(

Utils::vector\_concat(d, g), PI\_1

));

}

auto output = Utils::bit\_array\_to\_string(result);

if (padding && action == DES\_DECRYPT) {

return this->remove\_padding(output);

}

return output;

}

static string encrypt(string password, string text, bool padding = false) {

return Des(password, text).run(DES\_ENCRYPT, padding);

}

static string decrypt(string password, string text, bool padding = false) {

return Des(password, text).run(DES\_DECRYPT, padding);

}

};

class DoubleDes {

public:

static string encrypt(string password1, string password2, string text, bool padding = false) {

auto encrypted\_data = Des::encrypt(password1, text, padding);

auto encrypted\_double\_data = Des::encrypt(password2, encrypted\_data, padding);

return encrypted\_double\_data;

}

static string decrypt(string password1, string password2, string text, bool padding = false) {

auto decrypted\_data = Des::decrypt(password2, text, padding);

auto decrypted\_double\_data = Des::decrypt(password1, decrypted\_data, padding);

return decrypted\_double\_data;

}

};

class TripleDes {

public:

static string encrypt(string password1, string password2, string text, bool padding = false) {

auto encrypted\_data = Des::encrypt(password1, text, padding);

auto encrypted\_double\_data = Des::decrypt(password2, encrypted\_data, false);

auto encrypted\_triple\_data = Des::encrypt(password1, encrypted\_double\_data, padding);

return encrypted\_triple\_data;

}

static string decrypt(string password1, string password2, string text, bool padding = false) {

auto decrypted\_data = Des::decrypt(password1, text, padding);

auto decrypted\_double\_data = Des::encrypt(password2, decrypted\_data, false);

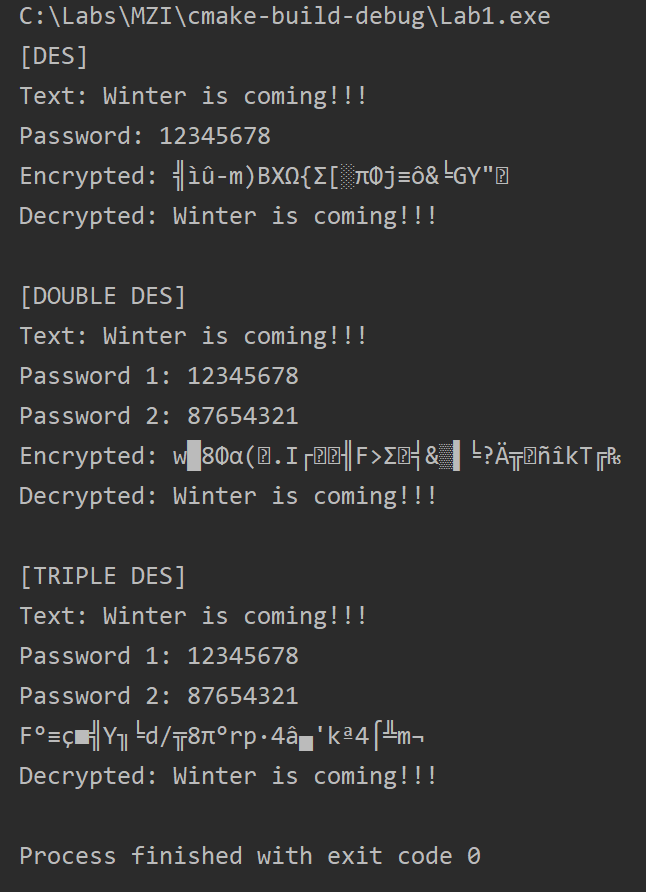
auto decrypted\_triple\_data = Des::decrypt(password1, decrypted\_double\_data, padding);

return decrypted\_triple\_data;

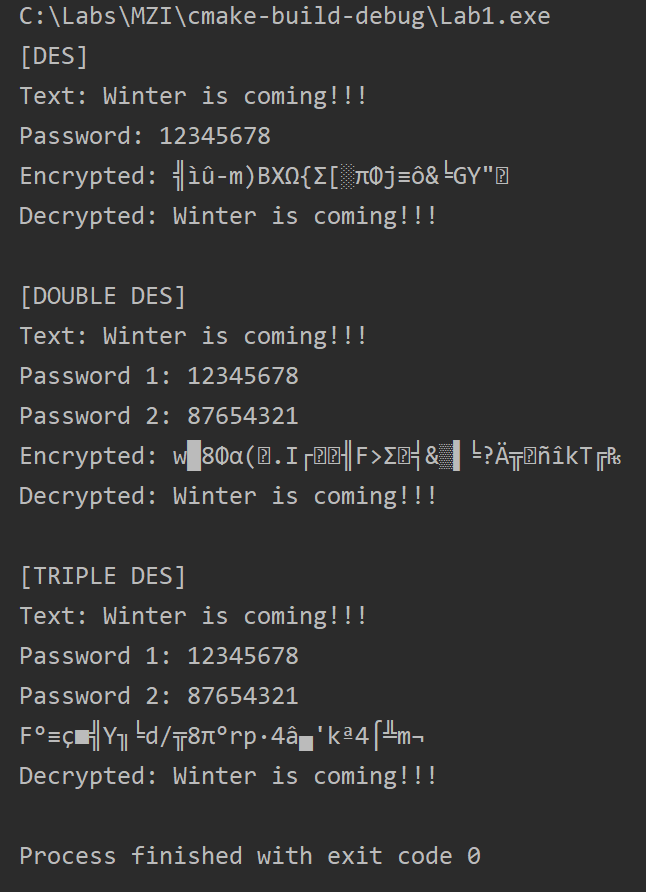
}

};

## Приложение 2. Скриншот работы программы



Скриншот работы алгоритма двойного Des



Скриншот работы алгоритма тройного Des