Министерство образования Республики Беларусь Учреждение образования
Белорусский государственный университет информатики и радиоэлектроники

Кафедра информатики

ОТЧЕТ

по лабораторной работе №1

Симметричная криптография. Двойной и тройной DES

Выполнил:
студент гр. 653503
Лисковец Б.Н.

Проверил:
Артемьев В.С.

Минск 2019

Введение

DES (англ. ***d**ata **e**ncryption **s**tandard*) – алгоритм для симметричного шифрования, разработанный фирмой IBM и утверждённый правительством США в 1977 году как официальный стандарт (FIPS 46-3). Размер блока для DES равен 64 бита. В основе алгоритма лежит сеть Фейстеля с 16 циклами (раундами) и ключом, имеющим длину 56 бит. Алгоритм использует комбинацию нелинейных (S-блоки) и линейных (перестановки E, IP, IP-1) преобразований. Для DES рекомендовано несколько режимов:

- ECB (англ. ***e**lectronic **c**ode **b**ook*) — режим «электронной кодовой книги» (простая замена);
- CBC (англ. ***c**ipher **b**lock **c**haining*) — режим сцепления блоков;
- CFB (англ. ***c**ipher **f**eed **b**ack*) — режим обратной связи по шифротексту;
- OFB (англ. ***o**utput **f**eed **b**ack*) — режим обратной связи по выходу;
- CTR (Counter Mode) — режим счетчика.

Прямым развитием DES в настоящее время является алгоритм Triple DES (3DES). В 3DES шифрование/расшифровка выполняются путём троекратного выполнения алгоритма DES.

Входными данными для блочного шифра служат:

- блок размером n бит;
- ключ размером k бит.

На выходе получается зашифрованный блок размером n бит, причём незначительные различия входных данных, как правило, приводят к существенному изменению результата.

Блочные шифры реализуются путём многократного применения к блокам исходного текста некоторых базовых преобразований.

Базовые преобразования:

- ✓ сложное преобразование на одной локальной части блока;
- ✓ простое преобразование между частями блока.

Так как преобразования производятся поблочно, требуется разделение исходных данных на блоки необходимого размера. При этом формат исходных данных не имеет значения (будь то текстовые документы, изображения или другие файлы). Данные должны интерпретироваться в двоичном виде (как последовательность нулей и единиц) и только после этого должны разбиваться на блоки. Все вышеперечисленное может осуществляться как программными, так и аппаратными средствами.

Задание: изучить шифрование данных при помощи классического алгоритма DES и его улучшенной версии (3DES), реализовать его программно.

Теоретическая часть

Блок-схема

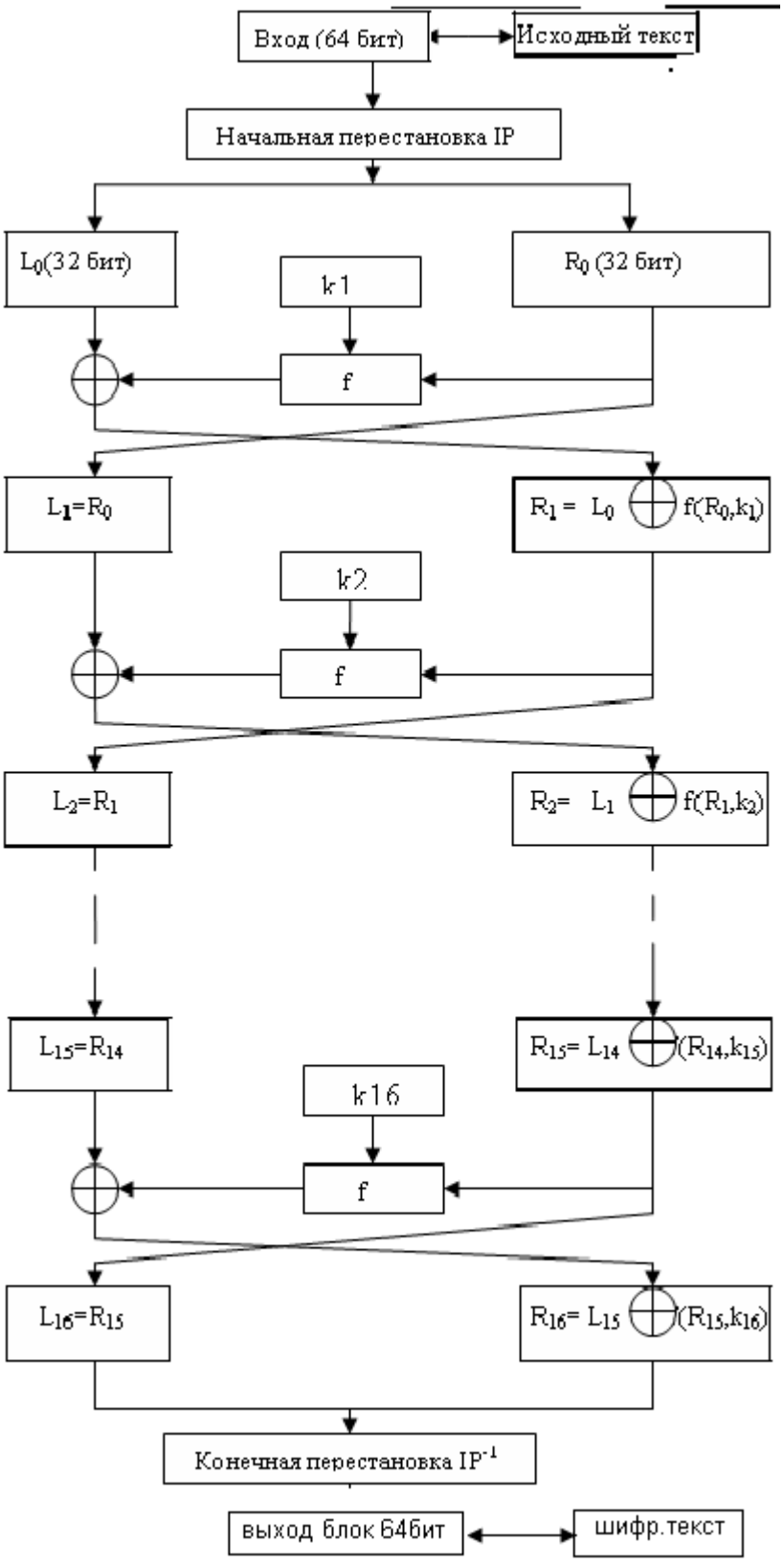

Рисунок 1 – блок-схема алгоритма DES

Алгоритм

1. Исходный текст – блок 64 бит.
2. Начальная перестановка – исходный текст **T (блок 64 бита)** преобразуется с помощью начальной перестановки **IP** (рис. 2)

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

Рисунок 2 – Начальная перестановка **IP**

1. Полученный после начальной перестановки 64-битовый блок IP(T) участвует в 16 циклах преобразования Фейстеля.

   Разбить IP(T) на две части L_i, R_i – 32 младших и старших бита соответственно. Тогда результат i-ой операции является:

$$\begin{aligned} T_i &= L_i R_i \\ L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, k_i) \end{aligned} \qquad (1)$$

1. Функция Фестеля – Аргументами функции являются (32-битовый) R_{i-1} и (48-битовый) k_i, который является результатом преобразования (56-битового) исходного ключа шифра k.
   Для вычисления f требуется:
   1) Функция расширения E.
   2) Сложение по модулю 2 с ключом k_i.
   3) Преобразование S, состоящее из 8 преобразований $S_1,\ldots,S_8$.
   4) Перестановка

   Функция E расширяет (32-битовый) R_{i-1} до 48-битового, путём дублирования некоторых битов из R_{i-1}. Порядок битов указан на рис. 3.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Рисунок 3 – Функция расширения E

Полученный после перестановок результат складывается по модулю 2 с ключом k_i и представляется в виде 8 последовательных блоков $B_1,\ldots,B_8$. Каждый B_j является 6-битным блоком. Далее каждый из B_j блоков трансформируется в 4-битовый блок B_j' с помощью преобразований S_j.

Предположим, что B_3=101111 и мы хотим найти B_3'. Первый и последний разряды B_3 являются двоичной записью числа a, 0<=a<=3, средние 4 разряда представляют число b, 0<=b<=15. Строки таблицы S3 нумеруются от 0 до 3, столбцы таблицы S3 нумеруются от 0 до 15. Пара чисел (a, b) определяет число, находящееся в пересечении строки a и столбца b. Двоичное представление этого числа дает B_3'. В нашем случае a=11_2=3, b=0111_2=3, а число, определяемое парой (3,7), равно 7. Его двоичное представление B_3'=0111.

Значение функции Фестеля получается перестановкой P, применяемой к 32-битовому блоку $B_1',..,B_8'$.

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

Рисунок 4 – Перестановка P

2. Генерация ключей k_i – Ключи k_i получаются из начального **k** ключа (56 бит = 7 байтов или 7 символов в ASCII) следующим образом. Добавляются биты в позиции 8, 16, 24, 32, 40, 48, 56, 64 ключа **k** таким образом, чтобы каждый байт содержал нечетное число единиц. Это используется для обнаружения ошибок при обмене и хранении ключей. Затем делают перестановку для расширенного ключа (кроме добавляемых битов 8, 16, 24, 32, 40, 48, 56, 64).

Эта перестановка определяется двумя блоками C_0 и D_0 по 28 бит кажд ый. C_i, D_i i=1,2,3…получаются из C_{i-1}
, D_{i-1} одним или двумя левыми циклическими сдвигами согласно Рис. 5.

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

Рисунок 5

Ключ k_i, i=1,…16 состоит из 48 бит, выбранных из битов вектора C_i D_i (56 бит) согласно Рис. 6.

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

Рисунок 6

3. Конечная перестановка – Конечная перестановка IP^{-1} действует на T_{16} и является обратной к первоначальной перестановке. Конечная перестановка определяется Рис. 7.

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

Рисунок 7 – Конечная перестановка IP^{-1}

Практическая часть

Введём строку:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis id urna elementum, eleifend ipsum a, ullamcorper mi. Aenean eu magna non dui dictum efficitur et vel lacus. Integer consectetur odio libero, vitae lacinia diam rutrum vel. Nunc at dolor sem. Duis aliquam cursus purus, at euismod nulla accumsan non. Proin nec massa tempus, pulvinar ligula nec, pellentesque metus. Suspendisse volutpat ipsum consectetur dapibus luctus. Ut ut justo consequat, efficitur metus vitae, porta neque. Vestibulum cursus sodales lacus a mollis. Proin libero quam, tincidunt vitae leo sit amet, finibus auctor justo. Nunc vitae bibendum felis, in rhoncus urna. Donec tincidunt scelerisque ullamcorper. Curabitur et felis nec lorem dignissim cursus. Donec vel mi enim. Nam tincidunt tempus arcu vitae tincidunt. Donec faucibus diam est, in auctor ligula egestas a.

Далее запустим программу и на экране появятся зашифрованное сообщение в 16тиричном формате, расшифрованное сообщение в 16тиричном формате, расшифрованное сообщение в воспринимаемом человеком формате.

Рисунок 8 – Работа программы

Приложение А. Текст программы

main.cpp

```
#include <cstdint>
#include <iostream>
#include <fstream>
#include <string>

#include "block.h"
#include "triple.h"

std::vector<uint8_t> prompt_key(int len = 8) {
  std::string key;
  std::cout << "Enter key:" << std::endl;
  std::getline(std::cin, key);
  key.resize(len, 0);
  return std::vector<uint8_t>(key.begin(), key.end());
}

std::vector<uint8_t> prompt_msg() {
  std::string msg;
  std::cout << "Enter msg:" << std::endl;
  std::getline(std::cin, msg);

  int remain = msg.size() % 8;
  if (remain > 0) {
    int whole = msg.size() / 8;
    msg.resize((whole + 1) * 8, 0);
  }

  return std::vector<uint8_t>(msg.begin(), msg.end());
}

int main() {
  std::vector<uint8_t> src = prompt_msg();
  std::vector<uint8_t> key = prompt_key(24);

  TripleDESCipher dc(key);

  std::vector<uint8_t> dst1 = dc.Encrypt(src);
  std::cout << "enc: " << std::hex;
  for (std::size_t i = 0; i < dst1.size(); i++) {
    std::cout << (uint64_t)(dst1[i]) << " ";
  }
  std::cout << std::dec << std::endl;

  std::vector<uint8_t> dst2 = dc.Decrypt(dst1);
  std::cout << "dec: " << std::hex;
  for (std::size_t i = 0; i < dst2.size(); i++) {
    std::cout << (uint64_t)(dst2[i]) << " ";
  }
  std::cout << std::dec << std::endl;

  std::cout << "dec(text): ";
```

```
 for (std::size_t i = 0; i < dst2.size(); i++) {
  std::cout << (char)(dst2[i]);
 }
 std::cout << std::endl;

 return 0;
}
```

cipher.cpp

```
#include <algorithm>
#include <cassert>
#include <cstring>
#include <iterator>

#include "block.h"
#include "cipher.h"
#include "const.h"

DESCipher::DESCipher(const std::vector<uint8_t> &key) {
 // incorrect key size
 assert(key.size() == 8);

 this->generateSubkeys(key);
}

const std::vector<uint64_t> &DESCipher::GetSubkeys() const {
 return this->subkeys;
}

std::vector<uint8_t> DESCipher::Encrypt(const std::vector<uint8_t> &msg) {
 return this->crypt(msg, true);
}

std::vector<uint8_t> DESCipher::Decrypt(const std::vector<uint8_t> &msg) {
 return this->crypt(msg, false);
}

std::vector<uint8_t> DESCipher::crypt(const std::vector<uint8_t> &msg,
                                      bool encrypt) {
 assert(msg.size() > 0 && msg.size() % blockSize == 0);

 std::size_t blocks = msg.size() / blockSize;

 std::vector<uint8_t> res(msg.size());

 std::vector<uint8_t> resBuffer(blockSize);
 std::vector<uint8_t> msgBuffer(blockSize);

 for (std::size_t i = 0; i < blocks; i++) {
  std::fill(resBuffer.begin(), resBuffer.end(), 0);
  std::copy(msg.begin() + i * blockSize, msg.begin() + (i + 1) * blockSize,
            msgBuffer.begin());

  if (encrypt)
   this->encrypt(resBuffer, msgBuffer);
  else
   this->decrypt(resBuffer, msgBuffer);

  std::copy(resBuffer.begin(), resBuffer.end(), res.begin() + i * blockSize);
 }
 return res;
}
```

```
void DESCipher::generateSubkeys(const std::vector<uint8_t> &keyBytes) {
 if (!feistelBoxInited) {
  initFeistelBox();
  feistelBoxInited = true;
 }

 // apply PC1 permutation to key
 // uint64_t key = binary.BigEndian.Uint64(keyBytes);
 uint64_t key = 0;
 for (int i = 0; i < 8; i++) {
  key += (static_cast<uint64_t>(keyBytes[i]) << blockSize * (7 - i));
 }

 uint64_t permutedKey = permuteBlock(key, permutedChoice1);

 // rotate halves of permuted key according to the rotation schedule
 std::vector<uint32_t> leftRotations =
    ksRotate(static_cast<uint32_t>(permutedKey >> 28));
 std::vector<uint32_t> rightRotations =
    ksRotate(static_cast<uint32_t>(permutedKey << 4) >> 4);

 // generate subkeys
 for (int8_t i = 0; i < 16; i++) {
  // combine halves to form 56-bit input to PC2
  uint64_t pc2Input = static_cast<uint64_t>(leftRotations[i]) << 28 |
            static_cast<uint64_t>(rightRotations[i]);
  // apply PC2 permutation to 7 byte input
  this->subkeys[i] = unpack(permuteBlock(pc2Input, permutedChoice2));
 }
}

void DESCipher::encrypt(std::vector<uint8_t> &dst,
            const std::vector<uint8_t> &src) {
 // input not full block
 assert(src.size() >= blockSize);

 // output not full block
 assert(dst.size() >= blockSize);

 encryptBlock(this->subkeys, dst, src);
}

void DESCipher::decrypt(std::vector<uint8_t> &dst,
            const std::vector<uint8_t> &src) {
 // input not full block
 assert(src.size() >= blockSize);

 // output not full block
 assert(dst.size() >= blockSize);

 // if (InexactOverlap(dst, blockSize, src, blockSize)) {
 // throw std::length_error("invalid buffer overlap");
 // }

 decryptBlock(this->subkeys, dst, src);
}
```

block.cpp

```
#include <cassert>
#include <cstdint>
#include <cstring>
#include <utility>
```

```cpp
#include <vector>

#include "block.h"
#include "const.h"

uint32_t feistelBox[8][64] = {{0}};
bool feistelBoxInited = false;

void cryptBlock(std::vector<uint64_t> &subkeys, std::vector<uint8_t> &dst,
                std::vector<uint8_t> src, bool decrypt) {
  uint64_t b = 0;
  assert(src.size() == 8);
  for (int i = 0; i < 8; i++) {
    b += (static_cast<uint64_t>(src[i]) << 8 * (7 - i));
  }

  b = permuteInitialBlock(b);

  uint32_t left = static_cast<uint32_t>(b >> 32);
  uint32_t right = static_cast<uint32_t>(b);

  left = (left << 1) | (left >> 31);
  right = (right << 1) | (right >> 31);

  if (decrypt) {
    for (uint8_t i = 0; i < 8; i++) {
      auto pair =
          feistel(left, right, subkeys[15 - 2 * i], subkeys[15 - (2 * i + 1)]);
      left = pair.first;
      right = pair.second;
    }
  } else {
    for (uint8_t i = 0; i < 8; i++) {
      auto pair = feistel(left, right, subkeys[2 * i], subkeys[2 * i + 1]);
      left = pair.first;
      right = pair.second;
    }
  }

  left = (left << 31) | (left >> 1);
  right = (right << 31) | (right >> 1);

  // switch left & right and perform final permutation
  uint64_t preOutput =
      (static_cast<uint64_t>(right) << 32) | static_cast<uint64_t>(left);

  uint64_t out = permuteFinalBlock(preOutput);
  assert(dst.size() == 8);
  for (int i = 0; i < 8; i++) {
    dst[i] = (out >> 8 * (7 - i));
  }
}

void encryptBlock(std::vector<uint64_t> &subkeys, std::vector<uint8_t> &dst,
                  const std::vector<uint8_t> &src) {
  cryptBlock(subkeys, dst, src, false);
}

void decryptBlock(std::vector<uint64_t> &subkeys, std::vector<uint8_t> &dst,
                  const std::vector<uint8_t> &src) {
  cryptBlock(subkeys, dst, src, true);
}

std::pair<uint32_t, uint32_t> feistel(uint32_t l, uint32_t r, uint64_t k0,
                                      uint64_t k1) {
```

```
  uint32_t t = 0;

  t = r ^ static_cast<uint32_t>(k0 >> 32);
  l ^= feistelBox[7][t & 0x3f] ^ feistelBox[5][(t >> 8) & 0x3f] ^
      feistelBox[3][(t >> 16) & 0x3f] ^ feistelBox[1][(t >> 24) & 0x3f];

  t = ((r << 28) | (r >> 4)) ^ static_cast<uint32_t>(k0);
  l ^= feistelBox[6][(t)&0x3f] ^ feistelBox[4][(t >> 8) & 0x3f] ^
      feistelBox[2][(t >> 16) & 0x3f] ^ feistelBox[0][(t >> 24) & 0x3f];

  t = l ^ static_cast<uint32_t>(k1 >> 32);
  r ^= feistelBox[7][t & 0x3f] ^ feistelBox[5][(t >> 8) & 0x3f] ^
      feistelBox[3][(t >> 16) & 0x3f] ^ feistelBox[1][(t >> 24) & 0x3f];

  t = ((l << 28) | (l >> 4)) ^ static_cast<uint32_t>(k1);
  r ^= feistelBox[6][(t)&0x3f] ^ feistelBox[4][(t >> 8) & 0x3f] ^
      feistelBox[2][(t >> 16) & 0x3f] ^ feistelBox[0][(t >> 24) & 0x3f];

  return std::make_pair(l, r);
}

uint64_t permuteBlock(uint64_t src, const std::vector<uint8_t> &permutation) {
  uint64_t block = 0;
  std::size_t len = permutation.size();
  for (uint8_t position = 0; position < len; position++) {
    uint8_t n = permutation[position];
    uint64_t bit = (src >> n) & 1;
    block |= (bit << static_cast<uint64_t>((len - 1) - position));
  }
  return block;
}

void initFeistelBox() {
  for (uint8_t s = 0; s < 8; s++) {
    for (uint8_t i = 0; i < 4; i++) {
      for (uint8_t j = 0; j < 16; j++) {
        uint64_t f = static_cast<uint64_t>(sBoxes[s][i][j])
                << (4 * (7 - static_cast<unsigned long>(s)));
        f = permuteBlock(f, permutationFunction);

        // Row is determined by the 1st and 6th bit.
        // Column is the middle four bits.
        uint8_t row = static_cast<uint8_t>(((i & 2) << 4) | (i & 1));
        uint8_t col = static_cast<uint8_t>(j << 1);
        uint8_t t = row | col;

        // The rotation was performed in the feistel rounds, being factored out
        // and now mixed into the feistelBox.
        f = (f << 1) | (f >> 31);

        feistelBox[s][t] = static_cast<uint32_t>(f);
      }
    }
  }
}

uint64_t permuteInitialBlock(uint64_t block) {
  // block = b7 b6 b5 b4 b3 b2 b1 b0 (8 bytes)
  uint64_t b1 = block >> 48;
  uint64_t b2 = block << 48;
  block ^= b1 ^ b2 ^ b1 << 48 ^ b2 >> 48;

  // block = b1 b0 b5 b4 b3 b2 b7 b6
  b1 = (block >> 32 & 0xff00ff);
  b2 = (block & 0xff00ff00);
```

```
  block ^=
     (b1 << 32) ^ b2 ^ (b1 << 8) ^ (b2 << 24); // exchange b0 b4 with b3 b7

  // block is now b1 b3 b5 b7 b0 b2 b4 b7, the permutation:
  //               ...  8
  //               ... 24
  //               ... 40
  //               ... 56
  //  7  6  5  4  3  2  1  0
  // 23 22 21 20 19 18 17 16
  //               ... 32
  //               ... 48

  // exchange 4,5,6,7 with 32,33,34,35 etc.
  b1 = block & 0x0f0f00000f0f0000;
  b2 = block & 0x0000f0f00000f0f0;
  block ^= b1 ^ b2 ^ (b1 >> 12) ^ (b2 << 12);

  // block is the permutation:
  //
  //   [+8]        [+40]
  //
  //  7  6  5  4
  // 23 22 21 20
  //  3  2  1  0
  // 19 18 17 16    [+32]

  // exchange 0,1,4,5 with 18,19,22,23
  b1 = block & 0x3300330033003300;
  b2 = block & 0x00cc00cc00cc00cc;
  block ^= b1 ^ b2 ^ (b1 >> 6) ^ (b2 << 6);

  // block is the permutation:
  // 15 14
  // 13 12
  // 11 10
  //  9  8
  //  7  6
  //  5  4
  //  3  2
  //  1  0 [+16] [+32] [+64]

  // exchange 0,2,4,6 with 9,11,13,15:
  b1 = block & 0xaaaaaaaa55555555;
  block ^= b1 ^ (b1 >> 33) ^ (b1 << 33);

  // block is the permutation:
  // 6 14 22 30 38 46 54 62
  // 4 12 20 28 36 44 52 60
  // 2 10 18 26 34 42 50 58
  // 0  8 16 24 32 40 48 56
  // 7 15 23 31 39 47 55 63
  // 5 13 21 29 37 45 53 61
  // 3 11 19 27 35 43 51 59
  // 1  9 17 25 33 41 49 57
  return block;
}

// permuteInitialBlock is equivalent to the permutation defined
// by finalPermutation.
uint64_t permuteFinalBlock(uint64_t block) {
  // Perform the same bit exchanges as permuteInitialBlock
  // but in reverse order.
  uint64_t b1 = block & 0xaaaaaaaa55555555;
  block ^= b1 ^ (b1 >> 33) ^ (b1 << 33);
```

```
  b1 = block & 0x3300330033003300;
  uint64_t b2 = block & 0x00cc00cc00cc00cc;
  block ^= b1 ^ b2 ^ (b1 >> 6) ^ (b2 << 6);

  b1 = block & 0x0f0f00000f0f0000;
  b2 = block & 0x0000f0f00000f0f0;
  block ^= b1 ^ b2 ^ (b1 >> 12) ^ (b2 << 12);

  b1 = (block >> 32) & 0xff00ff;
  b2 = (block & 0xff00ff00);
  block ^= (b1 << 32) ^ b2 ^ (b1 << 8) ^ (b2 << 24);

  b1 = block >> 48;
  b2 = block << 48;
  block ^= b1 ^ b2 ^ (b1 << 48) ^ (b2 >> 48);
  return block;
}

std::vector<uint32_t> ksRotate(uint32_t in) {
  auto out = std::vector<uint32_t>(16);
  uint32_t last = in;
  for (uint8_t i = 0; i < 16; i++) {
    // 28-bit circular left shift
    uint32_t left = (last << (4 + ksRotations[i])) >> 4;
    uint32_t right = (last << 4) >> (32 - ksRotations[i]);
    out[i] = left | right;
    last = out[i];
  }
  return out;
}

// Expand 48-bit input to 64-bit, with each 6-bit block padded by extra two bits
// at the top. By doing so, we can have the input blocks (four bits each), and
// the key blocks (six bits each) well-aligned without extra shifts/rotations
// for alignments.
uint64_t unpack(uint64_t x) {
  return ((x >> (6 * 1)) & 0xff) << (8 * 0) |
         ((x >> (6 * 3)) & 0xff) << (8 * 1) |
         ((x >> (6 * 5)) & 0xff) << (8 * 2) |
         ((x >> (6 * 7)) & 0xff) << (8 * 3) |
         ((x >> (6 * 0)) & 0xff) << (8 * 4) |
         ((x >> (6 * 2)) & 0xff) << (8 * 5) |
         ((x >> (6 * 4)) & 0xff) << (8 * 6) |
         ((x >> (6 * 6)) & 0xff) << (8 * 7);
}
```

triple.cpp

```
#include "triple.h"
#include "block.h"
#include <cassert>

TripleDESCipher::TripleDESCipher(const std::vector<uint8_t> &key) {
  // incorrect key size
  assert(key.size() == 24);

  d1 = new DESCipher(std::vector<uint8_t>(key.begin(), key.begin() + 8));
  d2 = new DESCipher(std::vector<uint8_t>(key.begin() + 8, key.begin() + 16));
  d3 = new DESCipher(std::vector<uint8_t>(key.begin() + 16, key.end()));
}

TripleDESCipher::~TripleDESCipher() {
  delete d1;
  delete d2;
  delete d3;
}
```

```
std::vector<uint8_t> TripleDESCipher::Encrypt(const std::vector<uint8_t> &msg) {
 return this->crypt(msg, true);
}

std::vector<uint8_t> TripleDESCipher::Decrypt(const std::vector<uint8_t> &msg) {
 return this->crypt(msg, false);
}

std::vector<uint8_t> TripleDESCipher::crypt(const std::vector<uint8_t> &msg,
                                            bool encrypt) {
 assert(msg.size() > 0 && msg.size() % blockSize == 0);

 std::size_t blocks = msg.size() / blockSize;

 std::vector<uint8_t> res(msg.size());

 std::vector<uint8_t> resBuffer(blockSize);
 std::vector<uint8_t> msgBuffer(blockSize);

 for (std::size_t i = 0; i < blocks; i++) {
  std::fill(resBuffer.begin(), resBuffer.end(), 0);
  std::copy(msg.begin() + i * blockSize, msg.begin() + (i + 1) * blockSize,
            msgBuffer.begin());

  if (encrypt)
   this->encrypt(resBuffer, msgBuffer);
  else
   this->decrypt(resBuffer, msgBuffer);

  std::copy(resBuffer.begin(), resBuffer.end(), res.begin() + i * blockSize);
 }
 return res;
}

void TripleDESCipher::encrypt(std::vector<uint8_t> &dst,
                              const std::vector<uint8_t> &src) {
 // input not full block
 assert(src.size() >= blockSize);

 // output not full block
 assert(dst.size() >= blockSize);

 uint64_t b = 0;
 assert(src.size() == 8);
 for (int i = 0; i < 8; i++) {
  b += (static_cast<uint64_t>(src[i]) << 8 * (7 - i));
 }

 b = permuteInitialBlock(b);

 uint32_t left = static_cast<uint32_t>(b >> 32);
 uint32_t right = static_cast<uint32_t>(b);

 left = (left << 1) | (left >> 31);
 right = (right << 1) | (right >> 31);

 auto sub1 = d1->GetSubkeys();
 auto sub2 = d2->GetSubkeys();
 auto sub3 = d3->GetSubkeys();

 for (int i = 0; i < 8; i++) {
  auto pair = feistel(left, right, sub1[2 * i], sub1[2 * i + 1]);
  left = pair.first;
```

```
    right = pair.second;
  }
  for (int i = 0; i < 8; i++) {
    auto pair = feistel(right, left, sub2[15 - 2 * i], sub2[15 - (2 * i + 1)]);
    right = pair.first;
    left = pair.second;
  }
  for (int i = 0; i < 8; i++) {
    auto pair = feistel(left, right, sub3[2 * i], sub3[2 * i + 1]);
    left = pair.first;
    right = pair.second;
  }

  left = (left << 31) | (left >> 1);
  right = (right << 31) | (right >> 1);

  uint64_t preOutput =
      (static_cast<uint64_t>(right) << 32) | static_cast<uint64_t>(left);

  uint64_t out = permuteFinalBlock(preOutput);
  assert(dst.size() == 8);
  for (int i = 0; i < 8; i++) {
    dst[i] = (out >> 8 * (7 - i));
  }
}

void TripleDESCipher::decrypt(std::vector<uint8_t> &dst,
                              const std::vector<uint8_t> &src) {
  // input not full block
  assert(src.size() >= blockSize);

  // output not full block
  assert(dst.size() >= blockSize);

  uint64_t b = 0;
  assert(src.size() == 8);
  for (int i = 0; i < 8; i++) {
    b += (static_cast<uint64_t>(src[i]) << 8 * (7 - i));
  }

  b = permuteInitialBlock(b);

  uint32_t left = static_cast<uint32_t>(b >> 32);
  uint32_t right = static_cast<uint32_t>(b);

  left = (left << 1) | (left >> 31);
  right = (right << 1) | (right >> 31);

  auto sub1 = d1->GetSubkeys();
  auto sub2 = d2->GetSubkeys();
  auto sub3 = d3->GetSubkeys();

  for (int i = 0; i < 8; i++) {
    auto pair = feistel(left, right, sub3[15 - 2 * i], sub3[15 - (2 * i + 1)]);
    left = pair.first;
    right = pair.second;
  }
  for (int i = 0; i < 8; i++) {
    auto pair = feistel(right, left, sub2[2 * i], sub2[2 * i + 1]);
    right = pair.first;
    left = pair.second;
  }
  for (int i = 0; i < 8; i++) {
    auto pair = feistel(left, right, sub1[15 - 2 * i], sub1[15 - (2 * i + 1)]);
    left = pair.first;
    right = pair.second;
```

```
  }

  left = (left << 31) | (left >> 1);
  right = (right << 31) | (right >> 1);

  uint64_t preOutput =
      (static_cast<uint64_t>(right) << 32) | static_cast<uint64_t>(left);

  uint64_t out = permuteFinalBlock(preOutput);
  assert(dst.size() == 8);
  for (int i = 0; i < 8; i++) {
    dst[i] = (out >> 8 * (7 - i));
  }
}
```