Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОССУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОННИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

Дисциплина: «Методы Защиты Информации»

Тема: «Алгоритмы симметричного шифрования. СТБ 34.101.31-2011 (БелТ)»

Выполнил:

студент гр. 653502

Куликов А. Д.

Проверил:

В. С. Артемьев

Минск 2019

# Постановка задачи

Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи стандарта СТБ 34.101.31-2011 (БелТ) в режимах простой замены и сцепления блоков.

# Схема

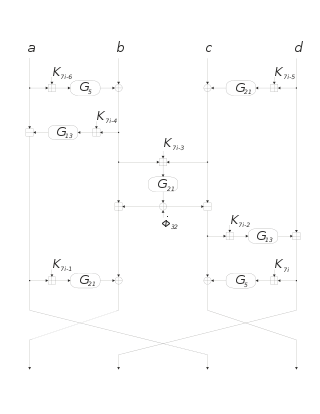
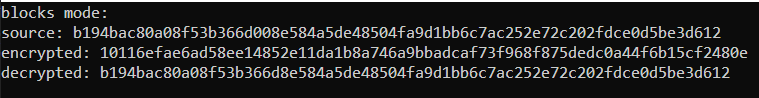


Рис. 1. Прямое преобразование сетью Фейстеля.

# Результат работы программы



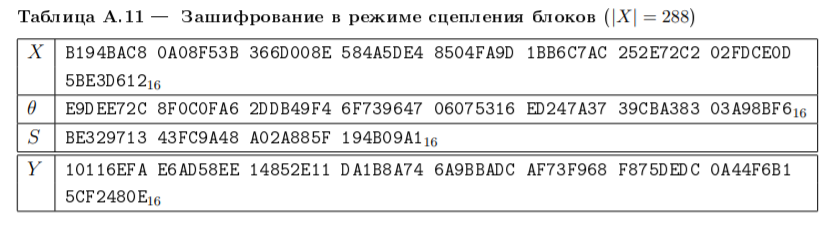
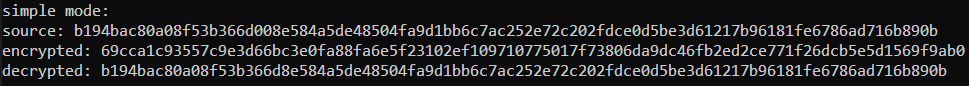


Рисунок 1 Пример зашифрования в режиме сцепления блоков из документации



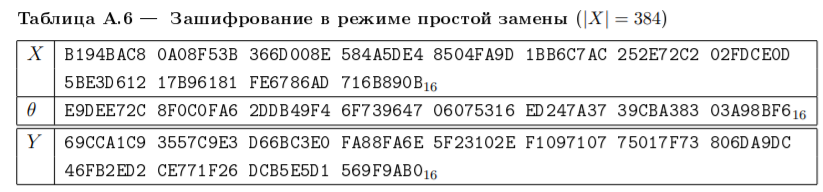


Рисунок 2 Пример зашифрования в режиме простой замены из документации

## Вывод

Данный алгоритм более удобен в реализации по сравнению с DES, т.к. при использовании типа byte (char) не требуется проводить дополнительных операций с данными для примедения их к требуемому типу (но условие, что нужно разбить на байты, остаётся).

# Исходный код программы

gost.cpp

#include "gost.h"

#include "utils.h"

#include "constants.h"

#include <iostream>

vector<bool> gost::h\_permutation(vector<bool> u) const

{

const auto y = (u[0] << 3) + (u[1] << 2) + (u[2] << 1) + u[3];

const auto x = (u[4] << 3) + (u[5] << 2) + (u[6] << 1) + u[7];

return get\_bits(H[y][x], 8);

}

vector<bool> gost::g(const vector<bool>& u, const int r) const

{

const auto block\_size = 8;

const auto len = u.size();

vector<bool> u\_new;

u\_new.reserve(len);

for (auto i = 0; i < 4; i++)

{

vector<bool> cur\_u(u.begin() + block\_size \* i, u.begin() + block\_size \* (i + 1));

cur\_u = h\_permutation(cur\_u);

u\_new.insert(u\_new.end(), cur\_u.begin(), cur\_u.end());

}

u\_new = rev\_bytes(u\_new);

rotate(u\_new.begin(), u\_new.begin() + r, u\_new.end());

u\_new = rev\_bytes(u\_new);

return u\_new;

}

vector<vector<bool>> gost::generate\_keys(const string& key) const

{

const auto size = 256;

const auto block\_size = 32;

const vector<char> bytes(key.begin(), key.end());

const auto bits = get\_bits(bytes);

const vector<bool> key\_bits(bits.begin(), bits.begin() + size);

vector<vector<bool>> keys;

for (auto i = 0; i < 7; i++)

for (auto j = 0; j < 8; j++)

keys.emplace\_back(key\_bits.begin() + block\_size \* j, key\_bits.begin() + block\_size \* (j + 1));

return keys;

}

vector<bool> gost::encrypt\_block(const vector<bool>& block, const vector<vector<bool>>& keys) const

{

const auto part\_size = 32;

vector<bool> res;

vector<bool> a(block.begin(), block.begin() + part\_size);

vector<bool> b(block.begin() + part\_size, block.begin() + part\_size \* 2);

vector<bool> c(block.begin() + part\_size \* 2, block.begin() + part\_size \* 3);

vector<bool> d(block.begin() + part\_size \* 3, block.begin() + part\_size \* 4);

vector<bool> e;

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

{

b = xor\_v(b, g(plus\_vec(a, keys[7 \* i], 32), 5));

c = xor\_v(c, g(plus\_vec(d, keys[7 \* i + 1], 32), 21));

a = minus\_vec(a, g(plus\_vec(b, keys[7 \* i + 2], 32), 13), 32);

const auto i\_vec = rev\_bytes(get\_bits(ll(i + 1) % (1LL << 32), 32));

e = g(plus\_vec(b, plus\_vec(c, keys[7 \* i + 3], 32), 32), 21);

e = xor\_v(e, i\_vec);

b = plus\_vec(b, e, 32);

c = minus\_vec(c, e, 32);

d = plus\_vec(d, g(plus\_vec(c, keys[7 \* i + 4], 32), 13), 32);

b = xor\_v(b, g(plus\_vec(a, keys[7 \* i + 5], 32), 21));

c = xor\_v(c, g(plus\_vec(d, keys[7 \* i + 6], 32), 5));

swap\_vec(a, b);

swap\_vec(c, d);

swap\_vec(b, c);

}

res.insert(res.end(), b.begin(), b.end());

res.insert(res.end(), d.begin(), d.end());

res.insert(res.end(), a.begin(), a.end());

res.insert(res.end(), c.begin(), c.end());

return res;

}

vector<bool> gost::decrypt\_block(const vector<bool>& block, const vector<vector<bool>>& keys) const

{

const auto part\_size = 32;

vector<bool> res;

vector<bool> a(block.begin(), block.begin() + part\_size);

vector<bool> b(block.begin() + part\_size, block.begin() + part\_size \* 2);

vector<bool> c(block.begin() + part\_size \* 2, block.begin() + part\_size \* 3);

vector<bool> d(block.begin() + part\_size \* 3, block.begin() + part\_size \* 4);

vector<bool> e;

for (auto i = 8; i > 0; i--)

{

b = xor\_v(b, g(plus\_vec(a, keys[7 \* i - 1], 32), 5));

c = xor\_v(c, g(plus\_vec(d, keys[7 \* i - 2], 32), 21));

a = minus\_vec(a, g(plus\_vec(b, keys[7 \* i - 3], 32), 13), 32);

const auto i\_vec = rev\_bytes(get\_bits(ll(i) % (1LL << 32), 32));

e = g(plus\_vec(b, plus\_vec(c, keys[7 \* i - 4], 32), 32), 21);

e = xor\_v(e, i\_vec);

b = plus\_vec(b, e, 32);

c = minus\_vec(c, e, 32);

d = plus\_vec(d, g(plus\_vec(c, keys[7 \* i - 5], 32), 13), 32);

b = xor\_v(b, g(plus\_vec(a, keys[7 \* i - 6], 32), 21));

c = xor\_v(c, g(plus\_vec(d, keys[7 \* i - 7], 32), 5));

swap\_vec(a, b);

swap\_vec(c, d);

swap\_vec(a, d);

}

res.insert(res.end(), c.begin(), c.end());

res.insert(res.end(), a.begin(), a.end());

res.insert(res.end(), d.begin(), d.end());

res.insert(res.end(), b.begin(), b.end());

return res;

}

gost::gost()

{

keys\_ = generate\_keys(default\_key\_);

}

gost::gost(const string& base\_key)

{

keys\_ = generate\_keys(base\_key);

}

string gost::encrypt\_simple(const string& message) const

{

const vector<char> bytes(message.begin(), message.end());

const auto bits = get\_bits(bytes);

auto blocks = get\_blocks(bits, 128, false);

vector<vector<bool>> encrypted\_blocks;

encrypted\_blocks.reserve(blocks.size());

const auto len = blocks.size();

if (len > 2) for (unsigned int i = 0; i < len - 2; i++) encrypted\_blocks.push\_back(encrypt\_block(blocks[i], keys\_));

const auto rest = 128 - blocks[len - 1].size();

if (rest != 0)

{

auto yn = encrypt\_block(blocks[len - 2], keys\_);

auto r = vector<bool>(yn.end() - rest, yn.end());

yn = vector<bool>(yn.begin(), yn.end() - rest);

auto xn = blocks[len - 1];

xn.insert(xn.end(), r.begin(), r.end());

const auto y\_prev = encrypt\_block(xn, keys\_);

encrypted\_blocks.push\_back(y\_prev);

encrypted\_blocks.push\_back(yn);

}

else

{

if (len > 1) encrypted\_blocks.push\_back(encrypt\_block(blocks[len - 2], keys\_));

encrypted\_blocks.push\_back(encrypt\_block(blocks[len - 1], keys\_));

}

return restore\_message(encrypted\_blocks);

}

string gost::decrypt\_simple(const string& message) const

{

const vector<char> bytes(message.begin(), message.end());

const auto bits = get\_bits(bytes);

auto blocks = get\_blocks(bits, 128, false);

vector<vector<bool>> encrypted\_blocks;

encrypted\_blocks.reserve(blocks.size());

const auto len = blocks.size();

if (len > 2) for (unsigned int i = 0; i < len - 2; i++) encrypted\_blocks.push\_back(decrypt\_block(blocks[i], keys\_));

const auto rest = 128 - blocks[len - 1].size();

if (rest != 0)

{

auto yn = decrypt\_block(blocks[len - 2], keys\_);

auto r = vector<bool>(yn.end() - rest, yn.end());

yn = vector<bool>(yn.begin(), yn.end() - rest);

auto xn = blocks[len - 1];

xn.insert(xn.end(), r.begin(), r.end());

const auto y\_prev = decrypt\_block(xn, keys\_);

encrypted\_blocks.push\_back(y\_prev);

encrypted\_blocks.push\_back(yn);

}

else

{

if (len > 1) encrypted\_blocks.push\_back(decrypt\_block(blocks[len - 2], keys\_));

encrypted\_blocks.push\_back(decrypt\_block(blocks[len - 1], keys\_));

}

return restore\_message(encrypted\_blocks);

}

string gost::encrypt\_blocks(const string& message, const string& sync) const

{

const vector<char> sync\_bytes(sync.begin(), sync.end());

const auto sync\_bits = get\_bits(sync\_bytes);

const vector<char> bytes(message.begin(), message.end());

const auto bits = get\_bits(bytes);

auto blocks = get\_blocks(bits, 128, false);

vector<vector<bool>> encrypted\_blocks;

encrypted\_blocks.reserve(blocks.size());

const auto len = blocks.size();

encrypted\_blocks.push\_back(sync\_bits);

if (len > 2)

for (unsigned int i = 0; i < len - 2; i++)

encrypted\_blocks.push\_back(encrypt\_block(xor\_v(blocks[i], encrypted\_blocks[i]), keys\_));

const auto rest = 128 - blocks[len - 1].size();

if (rest != 0)

{

auto yn = encrypt\_block(xor\_v(blocks[len - 2], encrypted\_blocks[len - 2]), keys\_);

auto r = vector<bool>(yn.end() - rest, yn.end());

yn = vector<bool>(yn.begin(), yn.end() - rest);

auto xn = xor\_v(blocks[len - 1], yn);

xn.insert(xn.end(), r.begin(), r.end());

const auto y\_prev = encrypt\_block(xn, keys\_);

encrypted\_blocks.push\_back(y\_prev);

encrypted\_blocks.push\_back(yn);

}

else

{

if (len > 1) encrypted\_blocks.push\_back(encrypt\_block(xor\_v(blocks[len - 2], encrypted\_blocks[len - 2]), keys\_));

encrypted\_blocks.push\_back(encrypt\_block(xor\_v(blocks[len - 1], encrypted\_blocks[len - 1]), keys\_));

}

return restore\_message(vector<vector<bool>>(encrypted\_blocks.begin() + 1, encrypted\_blocks.end()));

}

string gost::decrypt\_blocks(const string& message, const string& sync) const

{

const vector<char> sync\_bytes(sync.begin(), sync.end());

const auto sync\_bits = get\_bits(sync\_bytes);

const vector<char> bytes(message.begin(), message.end());

const auto bits = get\_bits(bytes);

auto blocks = get\_blocks(bits, 128, false);

auto filled\_blocks = get\_blocks(bits, 128, true);

vector<vector<bool>> encrypted\_blocks;

encrypted\_blocks.reserve(blocks.size());

const auto len = blocks.size();

if (len > 2)

for (unsigned int i = 0; i < len - 2; i++)

encrypted\_blocks.push\_back(xor\_v(decrypt\_block(blocks[i], keys\_), !i ? sync\_bits : blocks[i - 1]));

const auto rest = 128 - blocks[len - 1].size();

if (rest != 0)

{

auto yn = xor\_v(decrypt\_block(blocks[len - 2], keys\_), filled\_blocks[len - 1]);

auto r = vector<bool>(yn.end() - rest, yn.end());

yn = vector<bool>(yn.begin(), yn.end() - rest);

auto xn = blocks[len - 1];

xn.insert(xn.end(), r.begin(), r.end());

const auto y\_prev = xor\_v(decrypt\_block(xn, keys\_), len > 2 ? blocks[len - 3] : sync\_bits);

encrypted\_blocks.push\_back(y\_prev);

encrypted\_blocks.push\_back(yn);

}

else

{

if (len > 1) encrypted\_blocks.push\_back(xor\_v(decrypt\_block(blocks[len - 2], keys\_), len == 2 ? sync\_bits : blocks[len - 3]));

encrypted\_blocks.push\_back(xor\_v(decrypt\_block(blocks[len - 1], keys\_), blocks[len - 2]));

}

return restore\_message(encrypted\_blocks);

}