

Кузнечик («Kuznechik»)

(Version 1.0)

Выполнил Цыкунов А.И.

Оглавление

1. Теоретическая информация.....	3
1.1. Обозначения.....	4
1.2. Значения параметров.....	5
1.2.1. Нелинейное биективное преобразование.....	5
1.2.2. Линейное преобразование.....	5
1.3. Преобразования.....	7
1.4. Наши соглашения и упрощения обозначений.....	8
1.5. Алгоритм развёртки ключа.....	9
1.6. Базовый алгоритма шифрования.....	10
1.6.1. Алгоритм зашифрования.....	10
1.6.2. Алгоритм расшифрования.....	10
1.7. Наши дополнения и расширения ГОСТ.....	11
1.7.1. Таблица обратной подстановки.....	11
1.7.2. Итерационные константы.....	11
2. Практическая реализация.....	13
2.1. Структура программной реализации.....	13
2.1.0 Файлы зависимостей (tmp).....	13
2.1.1 Предварительные определения и объявления.....	14
2.1.2. Схема шифрования и расшифрования.....	16
2.1.3. Преобразование X	18
2.1.4. Преобразования S и S^{-1}	19
2.1.5. Преобразования L и L^{-1}	20
2.1.6. Генерирование итерационных ключей.....	22
2.2. Листинги файлов исходного кода.....	25
2.2.1. Файл kuznechik.h.....	25
2.2.2. Файл kuznechik.c.....	25

1. Теоретическая информация

Кузнечик («*Kuznechik*») - российский симметричный блочный алгоритм шифрования, определённый в ГОСТ 34.12 — 2018. Данный ГОСТ определяет 2 алгоритма шифрования: алгоритмы, оперирующие 128-битными блоками и 64-битными блоками данных. В данном тексте будет рассматриваться только 128-битный алгоритм шифрования. Изначально этот алгоритм был описан в стандарте ГОСТ 34.12 — 2015, где указывалось, что на данный алгоритм можно ссылаться, как на «*Kuznyechik*», однако в стандарте 2018 года указано, что на данный алгоритм можно ссылаться, как на «*Kuznechik*». Мы же в данном тексте будем ссылаться на этот алгоритм, как на **кузнечик** или **Кузнечик**.

Так же существует ГОСТ 34.13 — 2018 (переизданный ГОСТ 34.13 — 2015), который определяет режимы работы алгоритма и способы увеличения последнего блока в случае, если его размер не кратен размеру блока кузнечика.

Параметры алгоритма шифрования

Название	Kuznechik Кузнечик
Документ	ГОСТ 34.12 - 2018
Авторы	ФСБ России, АО «ИнфоТеКС»
Тип	Симметричный блочный SP - сеть
Размер блока	128 бит (16 байт)
Длина ключа	256 бит (32 байта)
Число раундов	10

1.1. Обозначения

ГОСТ 34.12 — 2018 вводит ряд обозначений для описания алгоритма Кузнечик:

- V^* - множество всех двоичных строк конечной длины, включая пустую строку;
- V_s - множество всех двоичных строк длины s , где s – целое неотрицательное число; нумерация подстрок и компонент строки осуществляется справа налево начиная с нуля;
- $U \times W$ - прямое (декартово) произведение множества U и множества W ;
- $|A|$ - число компонент (длина) строки $A \in V^*$ (если A – пустая строка, то $|A| = 0$);
- $A||B$ - конкатенация строк $A, B \in V^*$, т.е. строка из $V_{|A| + |B|}$, в которой подстрока с большими номерами компонент из $V_{|A|}$ совпадает со строкой A , а подстрока с меньшими номерами компонент из $V_{|B|}$ совпадает со строкой B ;
- \oplus - операция покомпонентного сложения по модулю 2 двух двоичных строк одинаковой длины;
- Z_2^s - кольцо вычетов по модулю 2^s ;
- \mathbb{F} - конечное поле $GF(2)[x] / p(x)$, где $p(x) = x^8 + x^7 + x^6 + x + 1 \in GF(2)[x]$; элементы поля \mathbb{F} представляются целыми числами, причем элементу $z_0 + z_1 \cdot \theta + \dots + z_7 \cdot \theta^7 \in \mathbb{F}$ соответствует число $z_0 + 2 \cdot z_1 + \dots + 2^7 \cdot z_7$, где $z_i \in \{0, 1\}$, $i = 0, 1, \dots, 7$ и θ обозначает класс вычетов по модулю $p(x)$, содержащий x ;
- $\text{Vec}_s: Z_2^s \rightarrow V_s$ - биективное отображение, сопоставляющее элементу кольца Z_2^s его двоичное представление, т.е. для любого элемента $z \in Z_2^s$, представленного в виде $z_0 + 2 \cdot z_1 + \dots + 2^{s-1} \cdot z_{s-1}$, где $z_i \in \{0, 1\}$, $i = 0, 1, \dots, s-1$, выполнено равенство $\text{Vec}_s(z) = z_{s-1}||z_{s-2}||\dots||z_1||z_0$;
- $\text{Int}_s: V_s \rightarrow Z_2^s$ - отображение, обратное к отображению Vec_s , т.е. $\text{Int}_s = (\text{Vec}_s)^{-1}$;
- $\Delta: V_8 \rightarrow \mathbb{F}$ - биективное отображение, сопоставляющее двоичной строке из V_8 элемент поля \mathbb{F} следующим образом: строке $z_7||\dots||z_1||z_0$, где $z_i \in \{0, 1\}$, $i = 0, 1, \dots, 7$, соответствует элемент $z_0 + z_1 \cdot \theta + \dots + z_7 \cdot \theta^7 \in \mathbb{F}$;
- $\nabla: \mathbb{F} \rightarrow V_8$ - отображение, обратное к отображению Δ , т.е. $\nabla = \Delta^{-1}$;
- $\Phi\Psi$ - композиция отображений, при которой отображение сначала выполняется Ψ , а затем Φ ;
- Φ^s - композиция отображений Φ^{s-1} и Φ , причём $\Phi^1 = \Phi$;

1.2. Значения параметров

1.2.1. Нелинейное биективное преобразование

В качестве нелинейного биективного преобразования выступает подстановка $\pi = \text{Vec}_8 \pi' \text{Int}_8: V_8 \rightarrow V_8$, где $\pi': Z_2^8 \rightarrow Z_2^8$. Значения подстановки π' записаны ниже в виде массива $\pi' = (\pi'(0), \pi'(1), \dots, \pi'(255))$:

$\pi' = ($
252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77,
233, 119, 240, 219, 147, 46, 153, 186, 23, 54, 241, 187, 20, 205, 95, 193,
249, 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142, 79,
5, 132, 2, 174, 227, 106, 143, 160, 6, 11, 237, 152, 127, 212, 211, 31,
235, 52, 44, 81, 234, 200, 72, 171, 242, 42, 104, 162, 253, 58, 206, 204,
181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156, 183, 93, 135,
21, 161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177,
50, 117, 25, 61, 255, 53, 138, 126, 109, 84, 198, 128, 195, 189, 13, 87,
223, 245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3,
224, 15, 236, 222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74,
167, 151, 96, 115, 30, 0, 98, 68, 26, 184, 56, 130, 100, 159, 38, 65,
173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165, 125, 105, 213, 149, 59,
7, 88, 179, 64, 134, 172, 29, 247, 48, 55, 107, 228, 136, 217, 231, 137,
225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144, 202, 216, 133, 97,
32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82,
89, 166, 116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182
).

1.2.2. Линейное преобразование

Линейное преобразование задается отображением $\ell: V_8^{16} \rightarrow V_8$, которое определяется следующим образом:

$$\begin{aligned} \ell(a_{15}, \dots, a_0) = & \nabla(148 \cdot \Delta(a_{15}) + 32 \cdot \Delta(a_{14}) + 133 \cdot \Delta(a_{13}) + 16 \cdot \Delta(a_{12}) + \\ & 194 \cdot \Delta(a_{11}) + 192 \cdot \Delta(a_{10}) + 1 \cdot \Delta(a_9) + 251 \cdot \Delta(a_8) + 1 \cdot \Delta(a_7) + 192 \cdot \Delta(a_6) + \\ & 194 \cdot \Delta(a_5) + 16 \cdot \Delta(a_4) + 133 \cdot \Delta(a_3) + 32 \cdot \Delta(a_2) + 148 \cdot \Delta(a_1) + 1 \cdot \Delta(a_0)) \end{aligned}$$

для любых $a_i \in V_8$, $i = 0, 1, \dots, 15$, где операции сложения и умножения осуществляются в поле \mathbb{F} , а константы являются элементами поля в указанном ранее смысле.

1.3. Преобразования

При реализации алгоритма зашифрования и расшифрования используются следующие преобразования:

Преобразование	Описание
$X[k]: V_{128} \rightarrow V_{128}$	$X[k] = k \oplus a$, где $k, a \in V_{128}$
$S: V_{128} \rightarrow V_{128}$	$S(a) = S(a_{15} \dots a_0) = \pi(a_{15}) \dots \pi(a_0)$, где $a_{15} \dots a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$S^{-1}: V_{128} \rightarrow V_{128}$	преобразование, обратное к преобразованию S
$R: V_{128} \rightarrow V_{128}$	$R(a) = R(a_{15} \dots a_0) = \ell(a_{15}, \dots, a_0)a_{15} \dots a_1$, где $a = a_{15} \dots a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$L: V_{128} \rightarrow V_{128}$	$L(a) = R^{16}(a)$, где $a \in V_{128}$
$R^{-1}: V_{128} \rightarrow V_{128}$	преобразование, обратное к преобразованию R , которое может быть вычислено, например, следующим образом: $R^{-1}(a) = R(a_{15} \dots a_0) = a_{14} \dots a_0 \ell(a_{14}, \dots, a_0, a_{15})$, где $a = a_{15} \dots a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$L^{-1}: V_{128} \rightarrow V_{128}$	$L^{-1}(a) = (R^{-1}(a))^{16}$, где $a \in V_{128}$
$F[k]: V_{128} \times V_{128} \rightarrow V_{128} \times V_{128}$	$F[k](a_1, a_0) = (LSX[k](a_1) \oplus a_0, a_1)$, где $k, a_0, a_1 \in V_{128}$

Стоит отметить, что преобразование X является обратным к самому себе, т.е. $X^{-1} = X$, т.к. $X[k]X[k](a) = k \oplus (k \oplus a) = a$. Поэтому в ГОСТ не вводится X^{-1} явно.

1.4. Наши соглашения и упрощения обозначений

ГОСТ определяет довольно подробно обозначения с целью избавления от неоднозначностей и ошибочных трактовок преобразований. Мы упростим данные обозначения:

1. Конкатенацию будет обозначать просто слитным написанием, вместо использования $\|$, т.е. $a_{15}\|...\|a_0 \rightarrow a_{15}...a_0$.
2. Алгоритм Кузнечик оперирует с блоком размером 128 бит, этот блок мы назовём состоянием (state).
3. Мы будем опускать обозначения отображений: Vec, Int, Δ и ∇ , подразумевая их неявно.
4. Поскольку индексы массивов в языке Си, как и во многих других языках программирования, начинается с 0, то оперирование с данными в обратном порядке может вызвать путаницу, поэтому мы будем рассматривать состояния в виде $a_0...a_{15}$ вместо $a_{15}...a_0$, просто изменив нумерацию. Стоит отметить побочные изменения:
 - Запись преобразования X никак не изменится.
 - Преобразования S и S^{-1} : $S(a) = S(a_0...a_{15}) = \pi(a_0)...\pi(a_{15})$, $S^{-1}(a) = S^{-1}(a_0...a_{15}) = \pi^{-1}(a_0)...\pi^{-1}(a_{15})$.
 - Линейное преобразование теперь записывается так $\ell(a_0, \dots, a_{15}) = 148 \cdot a_0 + 32 \cdot a_1 + 133 \cdot a_2 + 16 \cdot a_3 + 194 \cdot a_4 + 192 \cdot a_5 + 1 \cdot a_6 + 251 \cdot a_7 + 1 \cdot a_8 + 192 \cdot a_9 + 194 \cdot a_{10} + 16 \cdot a_{11} + 133 \cdot a_{12} + 32 \cdot a_{13} + 148 \cdot a_{14} + 1 \cdot a_{15}$.
 - Изменится и запись преобразований R и R^{-1} : $R(a) = R(a_0...a_{15}) = \ell(a_0, \dots, a_{15})a_0...a_{14}$, $R^{-1}(a) = R^{-1}(a_0...a_{15}) = a_1...a_{15}\ell(a_1, \dots, a_{15}, a_0)$.

Далее мы будем использовать описанные выше упрощения и введённые обозначения.

1.5. Алгоритм развёртки ключа

Для генерации итерационных ключей используются 32 итерационные константы, которые вычисляются следующим образом:

$$C_i = L(i), \text{ где } i = 1, \dots, 32.$$

Итерационные ключи $K_i \in V_{128}$, $i = 1, \dots, 10$, вырабатываются на основе ключа $K = k_0 \dots k_{255} \in V_{256}$, $k \in V_1$, $i = 0, \dots, 255$, и определяются равенствами:

$$K_1 = k_0 \dots k_{127}$$

$$K_2 = k_{128} \dots k_{255}$$

$$(K_{2i+1}, K_{2i+2}) = F[C_{8(i-1)+8}] \dots F[C_{8(i-1)+1}](K_{2i-1}, K_{2i}), i = 1, 2, 3, 4.$$

1.6. Базовый алгоритма шифрования

1.6.1. Алгоритм зашифрования

Алгоритм зашифрования в зависимости от значений итерационных ключей $K_i \in V_{128}$, $i = 1, \dots, 10$, реализует подстановку $E_{K_1, \dots, K_{10}}$, заданную на множестве V_{128} в соответствии с равенством:

$$E_{K_1, \dots, K_{10}}(a) = X[K_{10}]LSX[K_9] \dots LSX[K_2]LSX[K_1](a), \text{ где } a \in V_{128}.$$

1.6.2. Алгоритм расшифрования

Алгоритм расшифрования в зависимости от значений итерационных ключей $K_i \in V_{128}$, $i = 1, \dots, 10$, реализует подстановку $D_{K_1, \dots, K_{10}}$, заданную на множестве V_{128} в соответствии с равенством:

$$D_{K_1, \dots, K_{10}}(a) = X[K_1]S^{-1}L^{-1}X[K_2] \dots S^{-1}L^{-1}X[K_{10}](a), \text{ где } a \in V_{128}.$$

1.7. Наши дополнения и расширения ГОСТ

1.7.1. Таблица обратной подстановки

В ГОСТ 34.12 — 2018 явным образом не указана таблица обратной подстановки, которая используется для расшифрования. Мы приведём эту таблицу здесь.

$\pi^{-1} = (\pi^{-1}(0), \pi^{-1}(1), \dots, \pi^{-1}(255)) = ($
165, 45, 50, 143, 14, 48, 56, 192, 84, 230, 158, 57, 85, 126, 82, 145,
100, 3, 87, 90, 28, 96, 7, 24, 33, 114, 168, 209, 41, 198, 164, 63,
224, 39, 141, 12, 130, 234, 174, 180, 154, 99, 73, 229, 66, 228, 21, 183,
200, 6, 112, 157, 65, 117, 25, 201, 170, 252, 77, 191, 42, 115, 132, 213,
195, 175, 43, 134, 167, 177, 178, 91, 70, 211, 159, 253, 212, 15, 156, 47,
155, 67, 239, 217, 121, 182, 83, 127, 193, 240, 35, 231, 37, 94, 181, 30,
162, 223, 166, 254, 172, 34, 249, 226, 74, 188, 53, 202, 238, 120, 5, 107,
81, 225, 89, 163, 242, 113, 86, 17, 106, 137, 148, 101, 140, 187, 119, 60,
123, 40, 171, 210, 49, 222, 196, 95, 204, 207, 118, 44, 184, 216, 46, 54,
219, 105, 179, 20, 149, 190, 98, 161, 59, 22, 102, 233, 92, 108, 109, 173,
55, 97, 75, 185, 227, 186, 241, 160, 133, 131, 218, 71, 197, 176, 51, 250,
150, 111, 110, 194, 246, 80, 255, 93, 169, 142, 23, 27, 151, 125, 236, 88,
247, 31, 251, 124, 9, 13, 122, 103, 69, 135, 220, 232, 79, 29, 78, 4,
235, 248, 243, 62, 61, 189, 138, 136, 221, 205, 11, 19, 152, 2, 147, 128,
144, 208, 36, 52, 203, 237, 244, 206, 153, 16, 68, 64, 146, 58, 1, 38,
18, 26, 72, 104, 245, 129, 139, 199, 214, 32, 10, 8, 0, 76, 215, 116,
).

1.7.2. Итерационные константы

В ГОСТ 34.12 — 2018 128 битные итерационные константы не указаны явно, однако нет никакого смысла каждый раз их вычислять заново, т.к. они фиксированы. Мы их приведём здесь в шестнадцатеричной записи (по 2 шестнадцатеричные цифры на байт).

$C_1 = 6ea276726c487ab85d27bd10dd849401$

$C_2 = dc87ece4d890f4b3ba4eb92079cbeb02$

$C_3 = b2259a96b4d88e0be7690430a44f7f03$

$C_4 = 7bcd1b0b73e32ba5b79cb140f2551504$

$C_5 = 156f6d791fab511deabb0c502fd18105$

$C_6 = a74af7efab73df160dd208608b9efe06$
 $C_7 = c9e8819dc73ba5ae50f5b570561a6a07$
 $C_8 = f6593616e6055689adfb18027aa2a08$
 $C_9 = 98fb40648a4d2c31f0dc1c90fa2ebe09$
 $C_{10} = 2adedaf23e95a23a17b518a05e61c10a$
 $C_{11} = 447cac8052ddd8824a92a5b083e5550b$
 $C_{12} = 8d942d1d95e67d2c1a6710c0d5ff3f0c$
 $C_{13} = e3365b6ff9ae07944740add0087bab0d$
 $C_{14} = 5113c1f94d76899fa029a9e0ac34d40e$
 $C_{15} = 3fb1b78b213ef327fd0e14f071b0400f$
 $C_{16} = 2fb26c2c0f0aacd1993581c34e975410$
 $C_{17} = 41101a5e6342d669c4123cd39313c011$
 $C_{18} = f33580c8d79a5862237b38e3375cbf12$
 $C_{19} = 9d97f6babbd222da7e5c85f3ead82b13$
 $C_{20} = 547f77277ce987742ea93083bcc24114$
 $C_{21} = 3add015510a1fdcc738e8d936146d515$
 $C_{22} = 88f89bc3a47973c794e789a3c509aa16$
 $C_{23} = e65aedb1c831097fc9c034b3188d3e17$
 $C_{24} = d9eb5a3ae90ffa5834ce2043693d7e18$
 $C_{25} = b7492c48854780e069e99d53b4b9ea19$
 $C_{26} = 056cb6de319f0eeb8e80996310f6951a$
 $C_{27} = 6bcec0ac5dd77453d3a72473cd72011b$
 $C_{28} = a22641319aec1fd835291039b686b1c$
 $C_{29} = cc843743f6a4ab45de752c1346ecff1d$
 $C_{30} = 7ea1add5427c254e391c2823e2a3801e$
 $C_{31} = 1003dba72e345ff6643b95333f27141f$
 $C_{32} = 5ea7d8581e149b61f16ac1459ceda820$

2. Практическая реализация

2.1. Структура программной реализации

Файлы с исходным кодом: **kuznechik.c kuznechik.h**

Зависимости: **GF256_operations.c GF256_operations.h**

2.1.0 Файлы зависимостей (tmp)

(Описание файлов **GF256_operations.c** и **GF256_operations.h** приводится здесь временно. Потом их описание будет перенесено в другой справочный файл!)

Файл **GF256_operations.h** содержит условия препроцессора **#ifndef** → **#define** → **#endif** с целью предотвращения многократного включения этого заголовочного файла в другие файлы. Так же в этом файле создаётся псевдоним **byte** для типа данных **uint8_t** для более понятного кода. Ну и в конце объявляется внешняя функция **byte GF256_mul(byte a, byte b, byte modulo)**, смысл которой перемножить два многочлена *a* и *b* по модулю *modulo*.

```
/* GF256_operations.h */

#ifndef __GF256_operations_H__
#define __GF256_operations_H__

#include <stdint.h>

typedef uint8_t byte;

extern byte GF256_mul(byte a, byte b, byte modulo);

#endif
```

Почему эта функция работает с байтами, если мы должны перемножить многочлены? Всё довольно просто, любому многочлену вида $a_7 \cdot x^7 + \dots + a_1 \cdot x + a_0$ ($a_i \in \{0, 1\}$, $i = 0, \dots, 7$) можно однозначным образом сопоставить 8 разрядную двоичную строку V_8 (байт) вида $a_7 \dots a_1 a_0$. Перемножение многочленов по модулю другого многочлена эквивалентно побитовому перемножению в столбик двух байтов, соответствующих умножаемым многочленам, по модулю другого байта, соответствующего модульному многочлену. Ниже приведён исходный код (код файла **GF256_operations.c**) побитовому перемножению двух байт по модулю третьего байта:

```

/* GF256_operations.c */

#include "GF256_operations.h"

byte GF256_mul(byte a, byte b, byte modulo) {
    byte c = 0; /* accumulator for the product of the multiplication */
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomial for b has a constant term, add the
corresponding a to p */
            c ^= a; /* addition in GF(2^m) is an XOR of the polynomial coefficients */

        if (a & 0x80) /* GF modulo: if a has a nonzero term x^7, then must be
reduced when it becomes x^8 */
            a = (a << 1) ^ modulo; /* subtract (XOR) the primitive polynomial modulo
*/
        else
            a <<= 1; /* equivalent to a*x */
        b >>= 1;
    }
    return c;
}

```

2.1.1 Предварительные определения и объявления

Для начала определим некоторые константные значения алгоритма шифрования, к которым мы будем обращаться в коде.

```

static const int BLOCK_SIZE = 16;
static const int PI_SBOX_SIZE = 256;
static const int MODULO_POLY = 0xc3;

```

BLOCK_SIZE хранит размера блока (размер состояния) алгоритма шифрования в байтах. PI_SBOX_SIZE хранит размер массива подстановок π и π^{-1} . Самое интересное с константой MODULO_POLY, почему она равна 0xc3? В алгоритме шифрования Кузнечик используется конечное поле \mathbb{F} и неприводимый многочлен на этом поле $p(x) = x^8 + x^7 + x^6 + x + 1$, который мы можем представить в виде байта 11000011 (x^8 не учитывается) или 0xc3 в шестнадцатеричной записи.

Определим состояние (state) алгоритма шифрования, как массив из 16 байт ($16 \cdot 8 = 128$ бит):

```

typedef byte state[BLOCK_SIZE];

```

У алгоритма шифрования есть прямая и обратная подстановки S и S^{-1} по 256 байт каждая, которые мы в коде назовём **Pi** и **reverse_Pi** соответственно. В исходном коде мы будем обозначать значения подстановок в 16 системе счисления (здесь приводится лишь их фрагменты этих подстановок, их полные списки приведены в листинге **kuznechik.c**):

```
static const byte Pi[PI_SBOX_SIZE] = {
    0xfc, 0xee, 0xdd, 0x11, 0xcf, 0x6e, 0x31, 0x16,
    ...,
    0xd1, 0x66, 0xaf, 0xc2, 0x39, 0x4b, 0x63, 0xb6
};

static const byte reverse_Pi[PI_SBOX_SIZE] = {
    0xa5, 0x2d, 0x32, 0x8f, 0x0e, 0x30, 0x38, 0xc0,
    ...,
    0xd6, 0x20, 0x0a, 0x08, 0x00, 0x4c, 0xd7, 0x74
};
```

Массив коэффициентов линейного преобразования ℓ мы обозначим, как массив **l_coefficients**, из 16 элементов, его коэффициенты будут представлены в шестнадцатеричном виде:

```
static const byte l_coefficients[BLOCK_SIZE] = {
    0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01, 0xfb,
    0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01
};
```

Так же мы приведём здесь не полностью массив итерационных констант (полный набор смотри в листинге **kuznechik.c**). Как уже было написано выше, данные константы можно вычислить предварительно.

```
static const state iter_C[32] = {
    { 0x6e, 0xa2, 0x76, 0x72, 0x6c, 0x48, 0x7a, 0xb8, 0x5d, 0x27, 0xbd, 0x10, 0xdd, 0x84, 0x94, 0x01 },
    ...,
    { 0x5e, 0xa7, 0xd8, 0x58, 0x1e, 0x14, 0x9b, 0x61, 0xf1, 0x6a, 0xc1, 0x45, 0x9c, 0xed, 0xa8, 0x20 }
};
```

Так же добавим глобальный статистический массив **iter_key**, который будет содержать вычисленные итерационные ключи шифрования, т.к. таких ключей 10, то размер массива должен быть соответствующий.

```
static state iter_key[10];
```

Стоит обратить внимание, что все объявленные массивы и константы были объявлены статическими (модификатор **static** языка Си). Это значит, что данные массивы и константы ограничены областью видимости файла **kuznechik.c**, т.е. они будут не доступны вне этого файла, т.к. в этом нет никакой необходимости. Так же стоит обратить внимание на **typedef** определения типа **state**, как массив из 16 байт. Данный **typedef** так же ограничен областью видимости файла **kuznechik.c**, т.к. в реализациях других алгоритмах шифрования тоже будет определяться состояние (state), но оно может быть другим и, чтобы не было конфликта имён при компиляции файлов с реализациями нескольких алгоритмов шифрования, мы определяем **state** внутри файла **kuznechik.c**, а не внутри файла **kuznechik.h**.

2.1.2. Схема шифрования и расшифрования

Опираясь на пункты 1.6.1 и 1.6.2, мы можем схематично изобразить процедуры шифрования и расшифрования (см. рисунок 1).

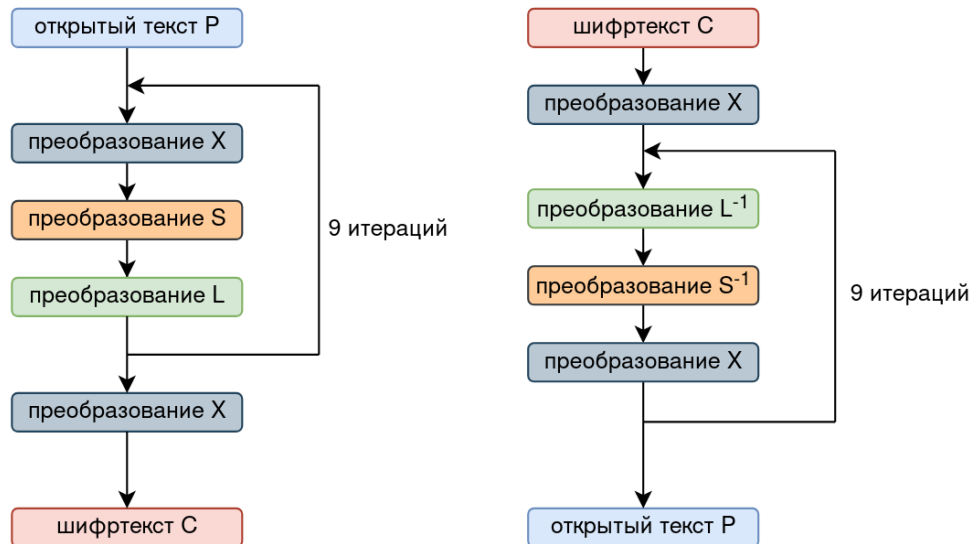


Рисунок 1. схемы шифрования и расшифрования алгоритма шифрования Кузнецик

Определим в функции **kuznechik.h** две внешние функции для шифрования и расшифрования:

```
extern void kuznechick_encrypt(const byte *blk, byte *out_blk);
extern void kuznechick_decrypt(const byte *blk, byte *out_blk);
```

Данные функции принимают по 2 указателя на первый байт состояния (state), причём первый указатель является константным, т.к. в случае шифрования это открытый текст, а в случае расшифрования — шифртекст, которые не должны меняться внутри функций шифрования и расшифрования соответственно.

Определим данные функции внутри файла **kuznechik.c**.

```
void kuznechick_encrypt(const byte *blk, byte *out_blk)
{
    int i;
    memcpy(out_blk, blk, BLOCK_SIZE);

    for(i = 0; i < 9; i++)
    {
        X_transformation(iter_key[i], out_blk, out_blk);
        S_transformation(out_blk, out_blk);
        L_transformation(out_blk, out_blk);
    }
    X_transformation(out_blk, iter_key[9], out_blk);
}
```



```

void kuznechick_decrypt(const byte *blk, byte *out_blk)
{
    int i;
    memcpy(out_blk, blk, BLOCK_SIZE);

    X_transformation(out_blk, iter_key[9], out_blk);
    for(i = 8; i >= 0; i--)
    {
        reverse_L_transformation(out_blk, out_blk);
        reverse_S_transformation(out_blk, out_blk);
        X_transformation(iter_key[i], out_blk, out_blk);
    }
}

```

Реализация функция шифрования и расшифрования совпадают с приведённой выше схемой. При шифровании мы используем прямой порядок итерационных ключей (с 1 по 10), а при расшифровании — обратный порядок ключей (с 10 по 1). Функция **memcpy** просто копирует данные из входного блока **blk** в блок **out_blk**.

2.1.3. Преобразование X

Преобразование X — простое сложение по модулю 2 (побитовый XOR) 128-битных строк: массива состояния (state) и итерационного ключа. Данное преобразование делает процедуру шифрования и расшифрования зависимой от ключа (принцип Кергоффа).

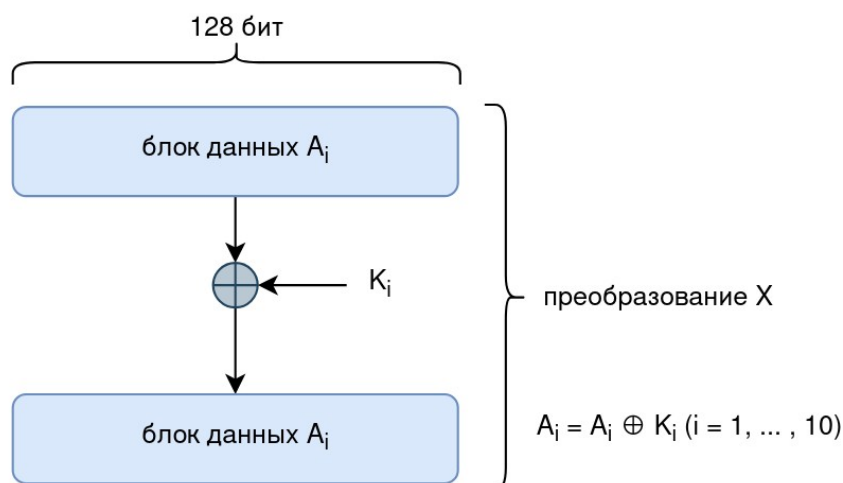


Рисунок 2. Преобразование X алгоритма шифрования Кузнечик

При расшифровании итерационные ключи идут в обратном порядке, поэтому преобразование X при расшифровании будет выглядеть немного иначе (см. рисунок 3).

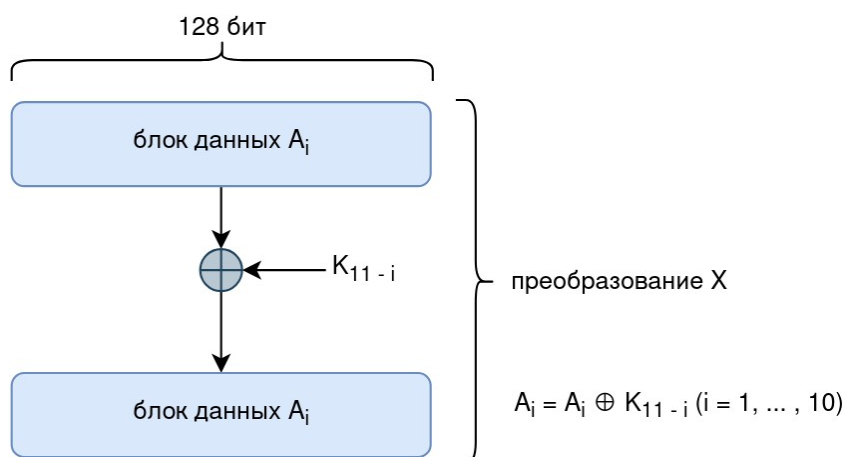


Рисунок 3. Преобразование X при расшифровании

В любом случае, при реализации преобразования X, порядок следования ключей не имеет значения (преобразование X не зависит от шифрования и расшифрования).

```
static void X_transformation(const byte *a, const byte *b, byte *c) {  
    for(int i = 0; i < BLOCK_SIZE; i++)  
        c[i] = a[i] ^ b[i];  
}
```

В данном случае, функция `X_transformation()` принимает три указателя на байты, первые 2 из которых являются указателем на первый байт итерационного ключа и на первый байт состояния (порядок не важен, операция XOR коммутативна), оба указателя являются константными, чтобы не менять значения своих аргументов. Третий указатель нужен для ссылки на первый байт состояния результата, куда будет записан результат сложения по модулю 2 первых двух аргументов. В цикле `for` мы просто перебираем все 16 байт состояния и ключа и производит их побитовый XOR.

2.1.4. Преобразования S и S⁻¹

Преобразование S — побайтовая подстановка. Данное преобразование вносит нелинейность в процедуру шифрования (принцип конфузии), защищая алгоритм от дифференциального и линейного криптоанализа (см. рисунок 4).

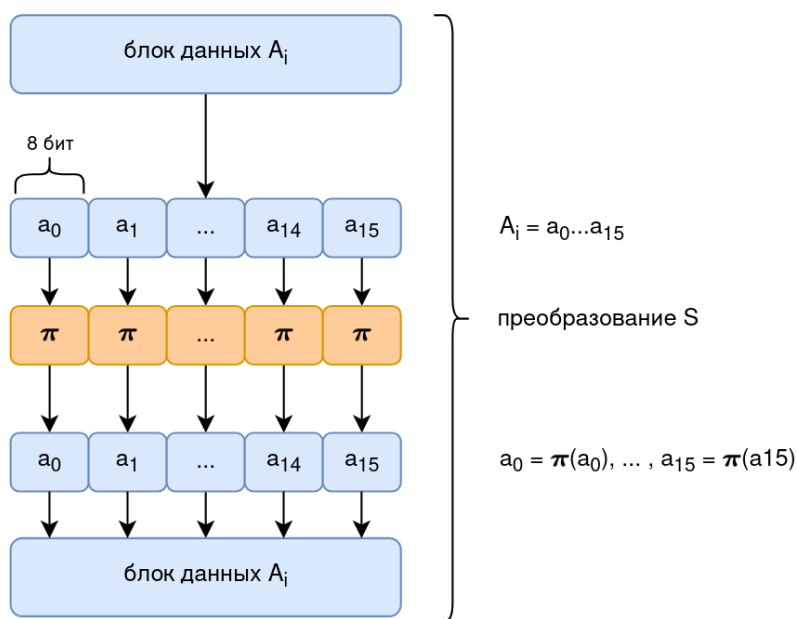


Рисунок 4. Преобразование S

Для реализации преобразования S нам нужно создать функцию, которая будет принимать 2 указателя: константный указатель на первый байт входных данных (входного состояния) и указатель на первый байт выходного состояния, результатом которого будет подстановка π всех 16 байт входного состояния:

```
static void S_transformation(const byte *in_data, byte *out_data) {
    for(int i = 0; i < BLOCK_SIZE; i++)
        out_data[i] = Pi[in_data[i]];
}
```

Обратное преобразование S⁻¹ аналогично преобразованию S.

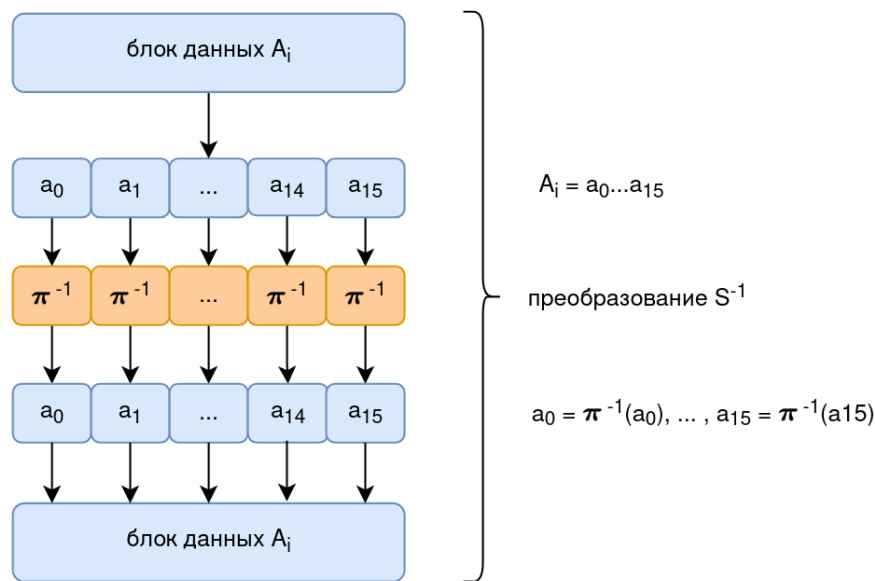


Рисунок 5. Обратное преобразование S

Реализация преобразования S^{-1} аналогично реализации преобразовании S .

```
static void reverse_S_transformation(const byte *in_data, byte *out_data) {
    for(int i = 0; i < BLOCK_SIZE; i++)
        out_data[i] = reverse_Pi[in_data[i]];
}
```

2.1.5. Преобразования L и L^{-1}

Преобразования L и L^{-1} строятся на преобразованиях R и R^{-1} соответственно по следующим формулам: $L(a) = R^{16}(a)$ и $L^{-1}(a) = (R^{-1})^{16}(a)$, где $a \in V_{128}$. Данные преобразования являются линейными и необходимы для формирования зависимости каждого бита блока шифртекста от всех битов открытого текста, устраняя статистические характеристики открытого текста (принцип диффузии).

Преобразование $R(a)$ представляет собой сдвиг всех 16 байт состояния a вправо и изменением первого байта по правилу $a_0 = \ell(a_0, \dots, a_{15})$, где $a_i \in V_8$, $i = 0, \dots, 15$. Преобразование L повторяет преобразование R 16 раз, тем самым затрагивая все байты состояния a (рисунок 6).

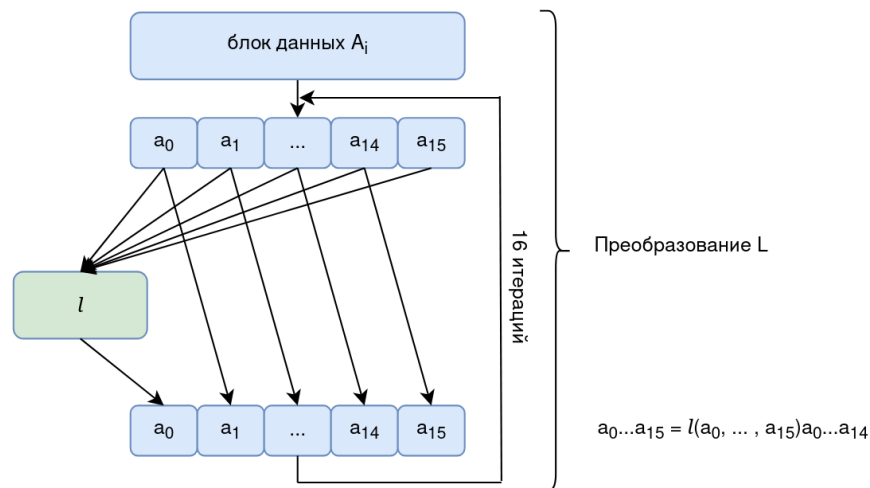


Рисунок 5. Преобразование L

Для начала реализуем преобразование R. Соответственно, функция должна принимать указатель на первый байт состояния, сдвинуть все байты вправо и присвоить первому байту результат линейного преобразования ℓ :

```
static void R_transformation(byte *state) {
    byte a_0 = state[15];
    state internal;

    for(int i = BLOCK_SIZE-2; i >= 0; i--) {
        internal[i+1] = state[i];
        a_0 ^= GF256_mul(state[i], l_coefficients[i], MODULO_POLY);
    }

    internal[0] = a_0;
    memcpy(state, internal, BLOCK_SIZE);
}
```

Строка `internal[i+1] = state[i]` сдвигает все байты вправо, а следующая строка вычисляет линейное преобразование ℓ .

Преобразование L тогда реализуется тривиально:

```
static void L_transformation(const byte *in_data, byte *out_data) {
    state internal;
    memcpy(internal, in_data, BLOCK_SIZE);

    for(int i = 0; i < BLOCK_SIZE; i++)
        R_transformation(internal);

    memcpy(out_data, internal, BLOCK_SIZE);
}
```

Аналогичным образом реализуются преобразования R^{-1} и L^{-1} .

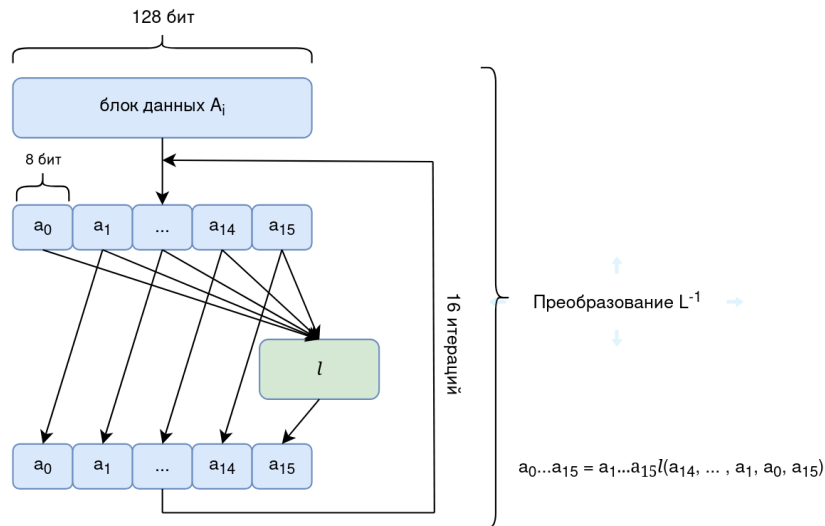


Рисунок 6. Обратное преобразование L

```
static void reverse_R_transformation(byte *state) {
    byte a_15 = state[0];
    state internal;

    for(int i = 0; i < BLOCK_SIZE-1; i++) {
        internal[i] = state[i+1];
        a_15 ^= GF256_mul(state[i+1], l_coefficients[i], MODULO_POLY);
    }

    internal[15] = a_15;
    memcpy(state, internal, BLOCK_SIZE);
}

static void reverse_L_transformation(const byte *in_data, byte *out_data) {
    state internal;
    memcpy(internal, in_data, BLOCK_SIZE);

    for(int i = 0; i < BLOCK_SIZE; i++)
        reverse_R_transformation(internal);

    memcpy(out_data, internal, BLOCK_SIZE);
}
```

2.1.6. Генерирование итерационных ключей

Процедура генерации итерационных ключей в алгоритме Кузнечик представляет собой применение 32-раундовую сеть Фейстеля, где в качестве функции F выступает композиция преобразований $LSX[C_i]$, где i — номер итерации, начиная с нуля. Первые два ключа получаются из носового 256-битного ключа, первый итерационный ключ — первые 128 бит основного ключа, второй итерационный ключ — последние 128 бит основного ключа. На основе первых двух итерационных ключей строятся остальные. На 8 итерации мы генерирования итерационных ключей мы получим 3 и 4 ключи, на 16 — 5 и 6 ключи т.д.

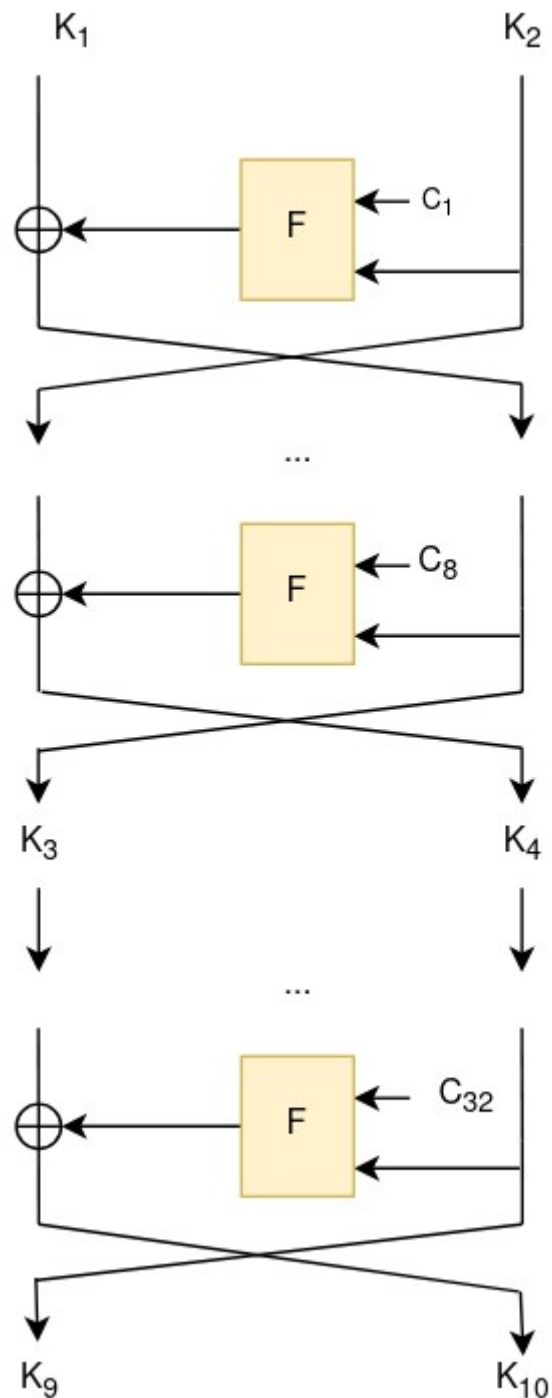


Рисунок 7. Генерирование итерационных ключей

Реализуем функцию F , которая будет принимать 2 указателя на первый байт входных 128 битных ключей, 2 указателя на первый байт сгенерированных ключей, а так же итерационную константу.

```
static void F_function(const byte *in_key1, const byte *in_key2,
    byte *out_key1, byte *out_key2, const byte *iter_const) {
    state internal;
    memcpy(out_key2, in_key1, BLOCK_SIZE);
    X_transformation(in_key1, iter_const, internal);
    S_transformation(internal, internal);
}
```

```

    L_transformation(internal, internal);
    X_transformation(internal, in_key2, out_key1);
}

```

Функция генерирования итерационных констант приведена ниже.

```

static void expand_key_function(const byte *key1, const byte *key2) {
    byte iter1[64], iter2[64], iter3[64], iter4[64];

    //calc_iter_consts_C();

    memcpy(iter_key[0], key1, 64);
    memcpy(iter_key[1], key2, 64);
    memcpy(iter1, key1, 64);
    memcpy(iter2, key2, 64);

    for (int i = 0; i < 4; i++)
    {
        F_function(iter1, iter2, iter3, iter4, iter_C[0 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[1 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[2 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[3 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[4 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[5 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[6 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[7 + 8 * i]);
        memcpy(iter_key[2 * i + 2], iter1, 64);
        memcpy(iter_key[2 * i + 3], iter2, 64);
    }
}

```


2.2. Листинги файлов исходного кода

2.2.1. Файл kuznechik.h

```
#ifndef __KUZNECHIK_H__
#define __KUZNECHIK_H__

extern void kuznechick_encrypt(const byte *blk, byte *out_blk);
extern void kuznechick_decrypt(const byte *blk, byte *out_blk);

#endif

#ifndef __KUZNECHIK_H__
#define __KUZNECHIK_H__

extern void kuznechick_encrypt(const byte *blk, byte *out_blk);
extern void kuznechick_decrypt(const byte *blk, byte *out_blk);

#endif
```

2.2.2. Файл kuznechik.c

```
#include <string.h>
#include <stdint.h>
#include "kuznechik.h"
#include "../math/GF256_operations.h"

static const int BLOCK_SIZE = 16;
static const int PI_SBOX_SIZE = 256;
static const int MODULO_POLY = 0xc3;

/*
    The Kuznechik algorithm perates with a set of 16 bytes,
    so we called it a state.
*/

typedef byte state[BLOCK_SIZE];

/*
    The block size according to Kuznechik is 128 bits. 128 bits = 16 bytes
*/
```

Kuznechik uses the Galois field $GF(2^8)$ modulo

the irreducible polynomial $p(x) = x^8 + x^7 + x^6 + x + 1$.

*This polynomial corresponds to 11000011 in binary representation
and 0xc3 in hexadecimal representation.*

**/*

```
static state iter_key[10];
```

*/**

Pi substitution table.

**/*

```
static const byte Pi[PI_SBOX_SIZE] = {  
    0xfc, 0xee, 0xdd, 0x11, 0xcf, 0x6e, 0x31, 0x16,  
    0xfb, 0xc4, 0xfa, 0xda, 0x23, 0xc5, 0x04, 0x4d,  
    0xe9, 0x77, 0xf0, 0xdb, 0x93, 0x2e, 0x99, 0xba,  
    0x17, 0x36, 0xf1, 0xbb, 0x14, 0xcd, 0x5f, 0xc1,  
    0xf9, 0x18, 0x65, 0x5a, 0xe2, 0x5c, 0xef, 0x21,  
    0x81, 0x1c, 0x3c, 0x42, 0x8b, 0x01, 0x8e, 0x4f,  
    0x05, 0x84, 0x02, 0xae, 0xe3, 0x6a, 0x8f, 0xa0,  
    0x06, 0x0b, 0xed, 0x98, 0x7f, 0xd4, 0xd3, 0x1f,  
    0xeb, 0x34, 0x2c, 0x51, 0xea, 0xc8, 0x48, 0xab,  
    0xf2, 0x2a, 0x68, 0xa2, 0xfd, 0x3a, 0xce, 0xcc,  
    0xb5, 0x70, 0x0e, 0x56, 0x08, 0x0c, 0x76, 0x12,  
    0xbf, 0x72, 0x13, 0x47, 0x9c, 0xb7, 0x5d, 0x87,  
    0x15, 0xa1, 0x96, 0x29, 0x10, 0x7b, 0x9a, 0xc7,  
    0xf3, 0x91, 0x78, 0x6f, 0x9d, 0x9e, 0xb2, 0xb1,  
    0x32, 0x75, 0x19, 0x3d, 0xff, 0x35, 0x8a, 0x7e,  
    0x6d, 0x54, 0xc6, 0x80, 0xc3, 0xbd, 0x0d, 0x57,  
    0xdf, 0xf5, 0x24, 0xa9, 0x3e, 0xa8, 0x43, 0xc9,  
    0xd7, 0x79, 0xd6, 0xf6, 0x7c, 0x22, 0xb9, 0x03,  
    0xe0, 0x0f, 0xec, 0xde, 0x7a, 0x94, 0xb0, 0xbc,  
    0xdc, 0xe8, 0x28, 0x50, 0x4e, 0x33, 0x0a, 0x4a,  
    0xa7, 0x97, 0x60, 0x73, 0x1e, 0x00, 0x62, 0x44,  
    0x1a, 0xb8, 0x38, 0x82, 0x64, 0x9f, 0x26, 0x41,  
    0xad, 0x45, 0x46, 0x92, 0x27, 0x5e, 0x55, 0x2f,  
    0x8c, 0xa3, 0xa5, 0x7d, 0x69, 0xd5, 0x95, 0x3b,  
    0x07, 0x58, 0xb3, 0x40, 0x86, 0xac, 0x1d, 0xf7,  
    0x30, 0x37, 0x6b, 0xe4, 0x88, 0xd9, 0xe7, 0x89,
```

```
0xe1, 0x1b, 0x83, 0x49, 0x4c, 0x3f, 0xf8, 0xfe,  
0x8d, 0x53, 0xaa, 0x90, 0xca, 0xd8, 0x85, 0x61,  
0x20, 0x71, 0x67, 0xa4, 0x2d, 0x2b, 0x09, 0x5b,  
0xcb, 0x9b, 0x25, 0xd0, 0xbe, 0xe5, 0x6c, 0x52,  
0x59, 0xa6, 0x74, 0xd2, 0xe6, 0xf4, 0xb4, 0xc0,  
0xd1, 0x66, 0xaf, 0xc2, 0x39, 0x4b, 0x63, 0xb6  
};
```

```
/*
```

```
Reverse Pi substitution table.
```

```
*/
```

```
static const byte reverse_Pi[PI_SBOX_SIZE] = {  
    0xa5, 0x2d, 0x32, 0x8f, 0x0e, 0x30, 0x38, 0xc0,  
    0x54, 0xe6, 0x9e, 0x39, 0x55, 0x7e, 0x52, 0x91,  
    0x64, 0x03, 0x57, 0x5a, 0x1c, 0x60, 0x07, 0x18,  
    0x21, 0x72, 0xa8, 0xd1, 0x29, 0xc6, 0xa4, 0x3f,  
    0xe0, 0x27, 0x8d, 0x0c, 0x82, 0xea, 0xae, 0xb4,  
    0x9a, 0x63, 0x49, 0xe5, 0x42, 0xe4, 0x15, 0xb7,  
    0xc8, 0x06, 0x70, 0x9d, 0x41, 0x75, 0x19, 0xc9,  
    0xaa, 0xfc, 0x4d, 0xbf, 0x2a, 0x73, 0x84, 0xd5,  
    0xc3, 0xaf, 0x2b, 0x86, 0xa7, 0xb1, 0xb2, 0x5b,  
    0x46, 0xd3, 0x9f, 0xfd, 0xd4, 0x0f, 0x9c, 0x2f,  
    0x9b, 0x43, 0xef, 0xd9, 0x79, 0xb6, 0x53, 0x7f,  
    0xc1, 0xf0, 0x23, 0xe7, 0x25, 0x5e, 0xb5, 0x1e,  
    0xa2, 0xdf, 0xa6, 0xfe, 0xac, 0x22, 0xf9, 0xe2,  
    0x4a, 0xbc, 0x35, 0xca, 0xee, 0x78, 0x05, 0x6b,  
    0x51, 0xe1, 0x59, 0xa3, 0xf2, 0x71, 0x56, 0x11,  
    0x6a, 0x89, 0x94, 0x65, 0x8c, 0xbb, 0x77, 0x3c,  
    0x7b, 0x28, 0xab, 0xd2, 0x31, 0xde, 0xc4, 0x5f,  
    0xcc, 0xcf, 0x76, 0x2c, 0xb8, 0xd8, 0x2e, 0x36,  
    0xdb, 0x69, 0xb3, 0x14, 0x95, 0xbe, 0x62, 0xa1,  
    0x3b, 0x16, 0x66, 0xe9, 0x5c, 0x6c, 0x6d, 0xad,  
    0x37, 0x61, 0x4b, 0xb9, 0xe3, 0xba, 0xf1, 0xa0,  
    0x85, 0x83, 0xda, 0x47, 0xc5, 0xb0, 0x33, 0xfa,  
    0x96, 0x6f, 0x6e, 0xc2, 0xf6, 0x50, 0xff, 0x5d,  
    0xa9, 0x8e, 0x17, 0x1b, 0x97, 0x7d, 0xec, 0x58,  
    0xf7, 0x1f, 0xfb, 0x7c, 0x09, 0x0d, 0x7a, 0x67,  
    0x45, 0x87, 0xdc, 0xe8, 0x4f, 0x1d, 0x4e, 0x04,
```

```

0xeb, 0xf8, 0xf3, 0x3e, 0x3d, 0xbd, 0x8a, 0x88,
0xdd, 0xcd, 0x0b, 0x13, 0x98, 0x02, 0x93, 0x80,
0x90, 0xd0, 0x24, 0x34, 0xcb, 0xed, 0xf4, 0xce,
0x99, 0x10, 0x44, 0x40, 0x92, 0x3a, 0x01, 0x26,
0x12, 0x1a, 0x48, 0x68, 0xf5, 0x81, 0x8b, 0xc7,
0xd6, 0x20, 0x0a, 0x08, 0x00, 0x4c, 0xd7, 0x74
};

/*
Coefficients used in the linear transformation of  $l(a_0, a_1, \dots, a_{15}) =$ 
 $148*a_0 + 32*a_1 + 133*a_2 + 16*a_3 + 194*a_4 + 192*a_5 + 1*a_6 + 251*a_7 + 1*a_8 + 192*a_9 +$ 
 $+ 194*a_{10} + 16*a_{11} + 133*a_{12} + 32*a_{13} + 148*a_{14} + 1*a_{15}.$ 
*/
static const byte l_coefficients[BLOCK_SIZE] = {
    0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01, 0xfb,
    0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01
};

/*
Iterative constants (iter_C) that are used when the key is expanded.
These constants can be calculated using the calc_iter_consts_C() function.
*/
static const state iter_C[32] = {
    { 0x6e, 0xa2, 0x76, 0x72, 0x6c, 0x48, 0x7a, 0xb8, 0x5d, 0x27, 0xbd, 0x10, 0xdd, 0x84, 0x94, 0x01 },
    { 0xdc, 0x87, 0xec, 0xe4, 0xd8, 0x90, 0xf4, 0xb3, 0xba, 0x4e, 0xb9, 0x20, 0x79, 0xcb, 0xeb, 0x02 },
    { 0xb2, 0x25, 0x9a, 0x96, 0xb4, 0xd8, 0x8e, 0x0b, 0xe7, 0x69, 0x04, 0x30, 0xa4, 0x4f, 0x7f, 0x03 },
    { 0x7b, 0xcd, 0x1b, 0x0b, 0x73, 0xe3, 0x2b, 0xa5, 0xb7, 0x9c, 0xb1, 0x40, 0xf2, 0x55, 0x15, 0x04 },
    { 0x15, 0x6f, 0x6d, 0x79, 0x1f, 0xab, 0x51, 0x1d, 0xea, 0xbb, 0x0c, 0x50, 0x2f, 0xd1, 0x81, 0x05 },
    { 0xa7, 0x4a, 0xf7, 0xef, 0xab, 0x73, 0xdf, 0x16, 0x0d, 0xd2, 0x08, 0x60, 0x8b, 0x9e, 0xfe, 0x06 },
    { 0xc9, 0xe8, 0x81, 0x9d, 0xc7, 0x3b, 0xa5, 0xae, 0x50, 0xf5, 0xb5, 0x70, 0x56, 0x1a, 0x6a, 0x07 },
    { 0xf6, 0x59, 0x36, 0x16, 0xe6, 0x05, 0x56, 0x89, 0xad, 0xfb, 0xa1, 0x80, 0x27, 0xaa, 0x2a, 0x08 },
    { 0x98, 0xfb, 0x40, 0x64, 0x8a, 0x4d, 0x2c, 0x31, 0xf0, 0xdc, 0x1c, 0x90, 0xfa, 0x2e, 0xbe, 0x09 },
    { 0x2a, 0xde, 0xda, 0xf2, 0x3e, 0x95, 0xa2, 0x3a, 0x17, 0xb5, 0x18, 0xa0, 0x5e, 0x61, 0xc1, 0x0a },
    { 0x44, 0x7c, 0xac, 0x80, 0x52, 0xdd, 0xd8, 0x82, 0x4a, 0x92, 0xa5, 0xb0, 0x83, 0xe5, 0x55, 0x0b },
    { 0x8d, 0x94, 0x2d, 0x1d, 0x95, 0xe6, 0x7d, 0x2c, 0x1a, 0x67, 0x10, 0xc0, 0xd5, 0xff, 0x3f, 0x0c },
    { 0xe3, 0x36, 0x5b, 0x6f, 0xf9, 0xae, 0x07, 0x94, 0x47, 0x40, 0xad, 0xd0, 0x08, 0x7b, 0xab, 0x0d },
    { 0x51, 0x13, 0xc1, 0xf9, 0x4d, 0x76, 0x89, 0x9f, 0xa0, 0x29, 0xa9, 0xe0, 0xac, 0x34, 0xd4, 0x0e },
    { 0x3f, 0xb1, 0xb7, 0x8b, 0x21, 0x3e, 0xf3, 0x27, 0xfd, 0x0e, 0x14, 0xf0, 0x71, 0xb0, 0x40, 0x0f },

```

```

    { 0x2f, 0xb2, 0x6c, 0x2c, 0x0f, 0x0a, 0xac, 0xd1, 0x99, 0x35, 0x81, 0xc3, 0x4e, 0x97, 0x54, 0x10 },
    { 0x41, 0x10, 0x1a, 0x5e, 0x63, 0x42, 0xd6, 0x69, 0xc4, 0x12, 0x3c, 0xd3, 0x93, 0x13, 0xc0, 0x11 },
    { 0xf3, 0x35, 0x80, 0xc8, 0xd7, 0x9a, 0x58, 0x62, 0x23, 0x7b, 0x38, 0xe3, 0x37, 0x5c, 0xbf, 0x12 },
    { 0x9d, 0x97, 0xf6, 0xba, 0xbb, 0xd2, 0x22, 0xda, 0x7e, 0x5c, 0x85, 0xf3, 0xea, 0xd8, 0x2b, 0x13 },
    { 0x54, 0x7f, 0x77, 0x27, 0x7c, 0xe9, 0x87, 0x74, 0x2e, 0xa9, 0x30, 0x83, 0xbc, 0xc2, 0x41, 0x14 },
    { 0x3a, 0xdd, 0x01, 0x55, 0x10, 0xa1, 0xfd, 0xcc, 0x73, 0x8e, 0x8d, 0x93, 0x61, 0x46, 0xd5, 0x15 },
    { 0x88, 0xf8, 0x9b, 0xc3, 0xa4, 0x79, 0x73, 0xc7, 0x94, 0xe7, 0x89, 0xa3, 0xc5, 0x09, 0xaa, 0x16 },
    { 0xe6, 0x5a, 0xed, 0xb1, 0xc8, 0x31, 0x09, 0x7f, 0xc9, 0xc0, 0x34, 0xb3, 0x18, 0x8d, 0x3e, 0x17 },
    { 0xd9, 0xeb, 0x5a, 0x3a, 0xe9, 0x0f, 0xfa, 0x58, 0x34, 0xce, 0x20, 0x43, 0x69, 0x3d, 0x7e, 0x18 },
    { 0xb7, 0x49, 0x2c, 0x48, 0x85, 0x47, 0x80, 0xe0, 0x69, 0xe9, 0x9d, 0x53, 0xb4, 0xb9, 0xea, 0x19 },
    { 0x05, 0x6c, 0xb6, 0xde, 0x31, 0x9f, 0x0e, 0xeb, 0x8e, 0x80, 0x99, 0x63, 0x10, 0xf6, 0x95, 0x1a },
    { 0x6b, 0xce, 0xc0, 0xac, 0x5d, 0xd7, 0x74, 0x53, 0xd3, 0xa7, 0x24, 0x73, 0xcd, 0x72, 0x01, 0x1b },
    { 0xa2, 0x26, 0x41, 0x31, 0x9a, 0xec, 0xd1, 0xfd, 0x83, 0x52, 0x91, 0x03, 0x9b, 0x68, 0x6b, 0x1c },
    { 0xcc, 0x84, 0x37, 0x43, 0xf6, 0xa4, 0xab, 0x45, 0xde, 0x75, 0x2c, 0x13, 0x46, 0xec, 0xff, 0x1d },
    { 0x7e, 0xa1, 0xad, 0xd5, 0x42, 0x7c, 0x25, 0x4e, 0x39, 0x1c, 0x28, 0x23, 0xe2, 0xa3, 0x80, 0x1e },
    { 0x10, 0x03, 0xdb, 0xa7, 0x2e, 0x34, 0x5f, 0xf6, 0x64, 0x3b, 0x95, 0x33, 0x3f, 0x27, 0x14, 0x1f },
    { 0x5e, 0xa7, 0xd8, 0x58, 0x1e, 0x14, 0x9b, 0x61, 0xf1, 0x6a, 0xc1, 0x45, 0x9c, 0xed, 0xa8, 0x20 }
};

/*
    Just XOR operation 128-bit blocks
*/

static void X_transformation(const byte *a, const byte *b, byte *c) {
    for(int i = 0; i < BLOCK_SIZE; i++)
        c[i] = a[i] ^ b[i];
}

/*
    Simple Pi substitution (each byte is replaced by the corresponding
    byte in Pi table.
*/

static void S_transformation(const byte *in_data, byte *out_data) {
    for(int i = 0; i < BLOCK_SIZE; i++)
        out_data[i] = Pi[in_data[i]];
}

/*
    Simple reverse Pi substitution (each byte is replaced by the corresponding
    byte in reverse Pi table.
*/

```

```

*/

static void reverse_S_transformation(const byte *in_data, byte *out_data) {
    for(int i = 0; i < BLOCK_SIZE; i++)
        out_data[i] = reverse_Pi[in_data[i]];
}

/*

    Shifting all bytes to the right and calculating the first byte using the linear transformation l:
    [a0, a1, ... , a15] --> [l(a0, a1, ... , a15), a0, a1, ... , a14].

*/

static void R_transformation(byte *state) {
    byte a_0 = state[15];
    state internal;

    for(int i = BLOCK_SIZE-2; i >= 0; i--) {
        internal[i+1] = state[i];
        a_0 ^= GF256_mul(state[i], l_coefficients[i], MODULO_POLY);
    }

    internal[0] = a_0;
    memcpy(state, internal, BLOCK_SIZE);
}

/*

    Shifting all bytes to the left and calculating the last byte using the linear transformation l:
    [a0, a1, ... , a15] --> [a1, a2, ... , a14, l(a1, a2, ... , a15), ].

*/

static void reverse_R_transformation(byte *state) {
    byte a_15 = state[0];
    state internal;

    for(int i = 0; i < BLOCK_SIZE-1; i++) {
        internal[i] = state[i+1];
        a_15 ^= GF256_mul(state[i+1], l_coefficients[i], MODULO_POLY);
    }

    internal[15] = a_15;
    memcpy(state, internal, BLOCK_SIZE);
}

```

```

/*
    Just 16 times R_transformation
*/

static void L_transformation(const byte *in_data, byte *out_data) {
    state internal;
    memcpy(internal, in_data, BLOCK_SIZE);

    for(int i = 0; i < BLOCK_SIZE; i++)
        R_transformation(internal);

    memcpy(out_data, internal, BLOCK_SIZE);
}

/*
    Just 16 times reverse_R_transformation
*/

static void reverse_L_transformation(const byte *in_data, byte *out_data) {
    state internal;
    memcpy(internal, in_data, BLOCK_SIZE);

    for(int i = 0; i < BLOCK_SIZE; i++)
        reverse_R_transformation(internal);

    memcpy(out_data, internal, BLOCK_SIZE);
}

/*
    F_function is used to expand encryption key and obtain iterative keys.

    F_function(a1, a0, key) = (L(S(X(a1, key))) + a0, a1), where + is means an XOR operation.

    In the implemented version of the F_function in_key1 and in_key2 correspond a1 and a0,
    iter_const correspond key, out_key1 correspond L(S(X(a1, key))) + a0, out_key2 correspond a1.
*/

static void F_function(const byte *in_key1, const byte *in_key2,
                      byte *out_key1, byte *out_key2, const byte *iter_const) {
    state internal;
    memcpy(out_key2, in_key1, BLOCK_SIZE);

```

```

X_transformation(in_key1, iter_const, internal);
S_transformation(internal, internal);
L_transformation(internal, internal);
X_transformation(internal, in_key2, out_key1);
}

```

```

static void expand_key_function(const byte *key1, const byte *key2) {
    byte iter1[64], iter2[64], iter3[64], iter4[64];

```

```

    //calc_iter_consts_C();

```

```

    memcpy(iter_key[0], key1, 64);
    memcpy(iter_key[1], key2, 64);
    memcpy(iter1, key1, 64);
    memcpy(iter2, key2, 64);

```

```

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

```

```

    {
        F_function(iter1, iter2, iter3, iter4, iter_C[0 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[1 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[2 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[3 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[4 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[5 + 8 * i]);
        F_function(iter1, iter2, iter3, iter4, iter_C[6 + 8 * i]);
        F_function(iter3, iter4, iter1, iter2, iter_C[7 + 8 * i]);
        memcpy(iter_key[2 * i + 2], iter1, 64);
        memcpy(iter_key[2 * i + 3], iter2, 64);
    }

```

```

}

```

```

/*

```

Before encryption, a function expand_key_function must be called to fill in the array of iterative keys (iter_key).

```

    kuznechick_encrypt(blk, out_blk) = out_blk = X(...(LSX(LSX(blk, iter_key[0]), iter_key[1]), ... ), iter_key[9])

```

```

*/

```

```

void kuznechick_encrypt(const byte *blk, byte *out_blk)

```



```
{  
    int i;  
    memcpy(out_blk, blk, BLOCK_SIZE);  
  
    for(i = 0; i < 9; i++)  
    {  
        X_transformation(iter_key[i], out_blk, out_blk);  
        S_transformation(out_blk, out_blk);  
        L_transformation(out_blk, out_blk);  
    }  
    X_transformation(out_blk, iter_key[9], out_blk);  
}  
  
void kuznechick_decrypt(const byte *blk, byte *out_blk)  
{  
    int i;  
    memcpy(out_blk, blk, BLOCK_SIZE);  
  
    X_transformation(out_blk, iter_key[9], out_blk);  
    for(i = 8; i >= 0; i--)  
    {  
        reverse_L_transformation(out_blk, out_blk);  
        reverse_S_transformation(out_blk, out_blk);  
        X_transformation(iter_key[i], out_blk, out_blk);  
    }  
}
```