Kuznechik

(Version 1.1)

Performed by Tsykunov A.

Table of contents

1. Theoretical information

Kuznechik is a Russian symmetric block encryption algorithm defined in GOST 34.12 — 2018. This GOST defines 2 encryption algorithms: an algorithm that operates with 128-bit blocks and with 64-bit data blocks. In this text, only the 128-bit encryption algorithm will be considered. Initially, this algorithm was described in the GOST 34.12 — 2015 standard, which indicated that this algorithm can be referred to as "Kuznyechik", however, the 2018 standard states that this algorithm can be referred to as "Kuznechik". In this text, we will refer to this algorithm as a **Kuznechik**.

There is also GOST 34.13 — 2018 (and its predecessor GOST 34.13 — 2015), which defines the operating modes of the algorithm and ways to supplement the last block of plaintext.

Encryption algorithm parameters

Name	Kuznechik
Published	2015
Document	GOST 34.12 - 2018
Authors	FSB of Russia, infotecs
Type	Symmetrical block SP network
Block size	128 bits (16 bytes)
Key length	256 bits (32 bytes)
Number of rounds	10

1.1. Designations

GOST 34.12 — 2018 introduces a number of designations to describe the Kuznechik algorithm:

- V^* - the set of all binary strings of finite length, including an empty string;
- V_S - the set of all binary strings of length s, where s is a non–negative integer; the numbering of substrings and components of the string is carried out from right to left starting from zero;
- U × W - the direct (Cartesian) product of the set U and the set W;
- |A| - the number of components (length) of the string $A \in V^*$ (if A is an empty string, then |A| = 0);
- A||B - string concatenation A, B $\in V^*$, that is, a string from $V_{|A| + |B|}$, in which a substring with large component numbers consists of $V_{|A|}$ matches the string A, and the substring with smaller numbers is a component of $V_{|B|}$ matches the string B;
- $\oplus$ - the operation of component-by-component addition modulo 2 of two binary strings of the same length;
- Z_{2^s} - the ring of deductions modulo 2^s;
- $\mathbb{F}$ - the final field $GF(2)[x] / p(x)$, where $p(x) = x^8 + x^7 + x^6 + x + 1 \in GF(2)[x]$; field elements of $\mathbb{F}$ they are represented by integers, and the element $z_0 + z_1 \cdot \theta + \ldots + z_7 \cdot \theta^7 \in \mathbb{F}$ corresponds to the number $z_0 + 2 \cdot z_1 + \ldots + 2^7 \cdot z_7$, where $z_i \in \{0, 1\}$, i = 0, 1, … , 7 and θ *denotes the class of deductions modulo* $p(x)$, containing x;
- Vec_s: $Z_{2^s} \rightarrow V_s$ - a bijective mapping matching an element of a ring Z_{2^s} its binary representation, i.e. for any element $z \in Z_{2^s}$, presented in the form of $z_0 + 2 \cdot z_1 + \ldots + 2^{s-1} \cdot z_{s-1}$, where $z_i \in \{0, 1\}, i = 0, 1, \ldots , s - 1$, equality is fulfilled $Vec_s(z) = z_{s-1}||z_{s-2}||\ldots||z_1||z_0$;
- Int_s: $V_s \rightarrow Z_{2^s}$ - the reverse of the display Vec_s, i.e. $Int_s = (Vec_s)^{-1}$;
- Δ: $V_8 \rightarrow \mathbb{F}$ - a bijective mapping matching a binary string from V_8 field element of $\mathbb{F}$ as follows: the string $z_7||\ldots||z_1||z_0$, where $z_i \in \{0, 1\}, i = 0, 1, \ldots , 7$, corresponds to the element $z_0 + z_1 \cdot \theta + \ldots + z_7 \cdot \theta^7 \in \mathbb{F}$;
- ∇: $\mathbb{F} \rightarrow V_8$ - the reverse of the display Δ, i.e. $\nabla = \Delta^{-1}$;
- $\Phi\Psi$ - the composition of the mappings in which the mapping is first performed Ψ, and then Φ;

- Φ^S - the composition of the displays Φ^{S-1} and Φ, with $\Phi^1 = \Phi$;

1.2. Parameter values

1.2.1. Nonlinear bijective transformation

Substitution acts as a nonlinear bijective transformation $\pi = \mathrm{Vec}_8 \pi' \mathrm{Int}_8 \colon V_8 \rightarrow V_8$, where $\pi' \colon \mathbb{Z}_{2^8} \rightarrow \mathbb{Z}_{2^8}$. Substitution values π' written below as an array $\pi' = (\pi'(0), \pi'(1), \ldots, \pi'(255))$:

π' = (

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. Linear transformation

The linear transformation is defined by the mapping $\ell \colon V_8^{16} \rightarrow V_8$, which is defined as follows:

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

for any $a_i \in V_8$, $i = 0, 1, \dots, 15$, where addition and multiplication operations are performed in the field $\mathbb{F}$, and constants are elements of the field in the sense indicated earlier.

1.3. Transformations

When implementing the encryption and decryption algorithm, the following transformations are used:

Transformation	Description
$X[k]: V_{128} \rightarrow V_{128}$	$X[k] = k \oplus a$, where $k, a \in V_{128}$
$S: V_{128} \rightarrow V_{128}$	$S(a) = S(a_{15}\|\|...\|\|a_0) = \pi(a_{15})\|\|...\|\| \pi(a_0)$, where $a_{15}\|\|...\|\|a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$S^{-1}: V_{128} \rightarrow V_{128}$	the inverse of the transformation S
$R: V_{128} \rightarrow V_{128}$	$R(a) = R(a_{15}\|\|...\|\|a_0) = \ell(a_{15}, \dots, a_0)a_{15}\|\|...\|\|a_1$, where $a = a_{15}\|\|...\|\|a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$L: V_{128} \rightarrow V_{128}$	$L(a) = R^{16}(a)$, where $a \in V_{128}$
$R^{-1}: V_{128} \rightarrow V_{128}$	the inverse of the transformation R, which can be calculated, for example, as follows: $R^{-1}(a) = R(a_{15}\|\|...\|\|a_0) = a_{14}\|\|...\|\|a_0 \|\|\ell(a_{14}, \dots, a_0, a_{15})$, where $a = a_{15}\|\|...\|\|a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, \dots, 15$
$L^{-1}: V_{128} \rightarrow V_{128}$	$L^{-1}(a) = (R^{-1}(a))^{16}$, where $a \in V_{128}$
$F[k]: V_{128} \times V_{128} \rightarrow V_{128} \times V_{128}$	$F[k](a_1, a_0) = (LSX[k](a_1) \oplus a_0, a_1)$, where $k, a_0, a_1 \in V_{128}$

It is worth noting that the transformation X it is the reverse of itself, i.e. $X^{-1} = X$, i.e. $X[k]X[k](a) = k \oplus (k \oplus a) = a$. Therefore, GOST is not introduced X^{-1} obviously.

1.4. Our conventions and simplifications of designations

GOST defines the designations in some detail in order to get rid of ambiguities and erroneous interpretations of transformations. We will simplify these designations:

1. Concatenation will be denoted simply by a single spelling, instead of using ||, i.e. $a_{15}||...||a_0 \rightarrow a_{15}...a_0$.
2. The Kuznechik algorithm operates with a block of 128 bits, we will call this block a state.
3. We will omit the notation of the mappings: Vec, Int, Δ and ∇, implicitly implying them.
4. Since array indexes in C, as in many other programming languages, start with 0, reverse numbering can cause confusion in software implementation, so we will consider states in the form $a_0...a_{15}$ instead of $a_{15}...a_0$, just by changing the numbering. It is worth noting the side effects:
   - Recording the transformation X it will not change in any way (this is a bitwise operation).
   - Transformations S and S^{-1}: $S(a) = S(a_0...a_{15}) = \pi(a_0)...\pi(a_{15})$, $S^{-1}(a) = S^{-1}(a_0...a_{15}) = \pi^{-1}(a_0)...\pi^{-1}(a_{15})$.
   - The linear transformation is now written as $\ell(a_0, \dots, a_{15}) = 148 \cdot a_0 + 32 \cdot a_1 + 133 \cdot a_2 + 16 \cdot a_3 + 194 \cdot a_4 + 192 \cdot a_5 + 1 \cdot a_6 + 251 \cdot a_7 + 1 \cdot a_8 + 192 \cdot a_9 + 194 \cdot a_{10} + 16 \cdot a_{11} + 133 \cdot a_{12} + 32 \cdot a_{13} + 148 \cdot a_{14} + 1 \cdot a_{15}$.
   - The transformation record will also change R and R^{-1}: $R(a) = R(a_0...a_{15}) = \ell(a_0, \dots, a_{15})a_0...a_{14}$ $R^{-1}(a) = R(a_0...a_{15}) = a_1...a_{15}\ell(a_1, \dots, a_{15}, a_0)$.

Next, we will use the simplifications described above and the notation introduced.

1.5. Key schedule algorithm

To generate iterative keys, 32 iterative constants are used, which are calculated as follows:

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

Iterative keys $K_i \in V_{128}$, $i = 1, \dots , 10$, they are generated based on the key $K = k_0 \ldots k_{255} \in V_{256}$, $k \in V_1$, $i = 0, \dots , 255$, and are determined by the equalities:

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

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

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

1.6. Basic encryption algorithm

1.6.1. The encryption algorithm

The encryption algorithm depends on the values of the iterative keys $K_i \in V_{128}$, i = 1, … , 10, implements substitution $E_{K1, \dots, K10}$, given on the set V_{128} according to equality:

$$E_{K1, \dots, K10}(a) = X[K_{10}]LSX[K_9]...LSX[K_2]LSX[K_1](a), \text{ where } a \in V_{128}.$$

1.6.2. The decryption algorithm

The decryption algorithm depends on the values of the iterative keys $K_i \in V_{128}$, i = 1, … , 1, implements substitution $D_{K1, \dots, K10}$, given on the set V_{128} according to equality:

$$D_{K1, \dots, K10}(a) = X[K_1]S^{-1}L^{-1}X[K_2]...S^{-1}L^{-1}X[K_{10}](a), \text{ where } a \in V_{128}.$$

1.7. Our additions and extensions of GOST

1.7.1. The reverse substitution table

GOST 34.12 — 2018 does not explicitly specify the reverse substitution table, which is used for decryption. We will provide this table here.

$\pi^{-1} = (\pi^{-1}(0), \pi^{-1}(1), \dots, \pi^{-1}(255)) = ($

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. Iterative constants

In GOST 34.12 — 2018, 128-bit iterative constants are not explicitly specified, but there is no point in calculating them again every time, because they are fixed. We will give them here in hexadecimal notation (2 hexadecimal digits per byte).

C_1 = 6ea276726c487ab85d27bd10dd849401

C_2 = dc87ece4d890f4b3ba4eb92079cbeb02

C_3 = b2259a96b4d88e0be7690430a44f7f03

C_4 = 7bcd1b0b73e32ba5b79cb140f2551504

C_5 = 156f6d791fab511deabb0c502fd18105

C_6 = a74af7efab73df160dd208608b9efe06

C_7 = c9e8819dc73ba5ae50f5b570561a6a07

C_8 = f6593616e6055689adfba18027aa2a08

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} = a22641319aecd1fd835291039b686b1c

C_{29} = cc843743f6a4ab45de752c1346ecff1d

C_{30} = 7ea1add5427c254e391c2823e2a3801e

C_{31} = 1003dba72e345ff6643b95333f27141f

C_{32} = 5ea7d8581e149b61f16ac1459ceda820

2. Practical implementation

2.1. The structure of the software implementation

Source code files: **kuznechik.c kuznechik.h**

Dependencies: **GF256_operations.c GF256_operations.h**

2.1.0. Dependency files (tmp)

(Description of the files **GF256_operations.c** and **GF256_operations.h** It is given here temporarily. Then their description will be transferred to another help file!)

File **GF256_operations.h** contains preprocessor conditions **#ifndef** → **#define** → **#endif** in order to prevent this header file from being included in other files multiple times. An alias is also created in this file **byte** for the data type **uint8_t** for a more understandable code. At the end, an external function is declared **byte GF256_mul(byte a, byte b, byte modulo)**, the meaning of which is to multiply two polynomials a and b and 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
```

Why does this function work with bytes if we have to multiply the polynomials? It's pretty simple, for any polynomial of the form $a_7 \cdot x^7 + \ldots + a_1 \cdot x + a_0$ ($a_i \in \{0, 1\}$, $i = 0, \ldots, 7$) it is possible to uniquely match an 8-bit binary string V_8 (bytes) of the form $a_7 \ldots a_1 a_0$. Пmultiplication of polynomials modulo another polynomial is equivalent to bitwise multiplication into a column of two bytes corresponding to the multiplied polynomials modulo another byte corresponding to the modular polynomial. Below is the source code (file code **GF256_operations.c**) пbitwise multiplication of two bytes modulo the third byte:

```
/* 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. Preliminary definitions and declarations

First, let's define some constant values of the encryption algorithm, which we will refer to in the code.

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

BLOCK_SIZE stores the block size (state size) of the encryption algorithm in bytes. PI_SBOX_SIZE stores the size of the array of substitutions π and π^{-1}. The most interesting thing with the MODULO_POLY constant is why it is equal to 0xc3? The Kuznechik encryption algorithm uses a finite field $\mathbb{F}$ and an irreducible polynomial over this field $p(x) = x^8 + x^7 + x^6 + x + 1$, which we can represent as a byte 11000011 (x^8 is not taken into account) or 0xc3 in hexadecimal notation.

Let's define the state of the encryption algorithm as an array of 16 bytes ($16 \cdot 8 = 128$ bits):

```
typedef byte state[BLOCK_SIZE];
```

The encryption algorithm has forward and reverse substitutions S and S^{-1} 256 bytes each, which we will call in the code **Pi** or **reverse_Pi** accordingly. In the source code, we will denote the values of substitutions in the 16 number system (only fragments of these substitutions are given here, their complete lists are given in the listing **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
};
```

Array of linear transformation coefficients ℓ we will denote as an array **l_coefficients**, of the 16 elements, its coefficients will be presented in hexadecimal form:

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

We will also give here an incomplete array of iterative constants (see the full set in the listing **kuznechik.c**). As already mentioned above, these constants can be calculated in advance.

```
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 }
};
```

We will also add a global statistical array **iter_key**, which will contain numerical iterative encryption keys, since there are 10 such keys, then the size of the array must be appropriate.

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

Cit is worth noting that all declared arrays and constants have been declared static (modifier **static** the C language). This means that these arrays and constants are limited by the scope of the file **kuznechik.c**, that is, they will not be available outside of this file, because there is no need for this. It is also worth paying attention to **typedef** type definitions **state**, кas an array of 16 bytes. This **typedef** it is also limited by the scope of the file **kuznechik.c**, because the state will also be determined in implementations of other encryption algorithms, but it may be different and, so that there is no name conflict when compiling files with implementations of several encryption algorithms, we define **state** inside the file **kuznechik.c**, and not inside the file **kuznechik.h**.

2.1.2. Encryption and decryption scheme

Based on paragraphs 1.6.1 and 1.6.2, we can schematically depict the encryption and decryption procedures (see Figure 1).

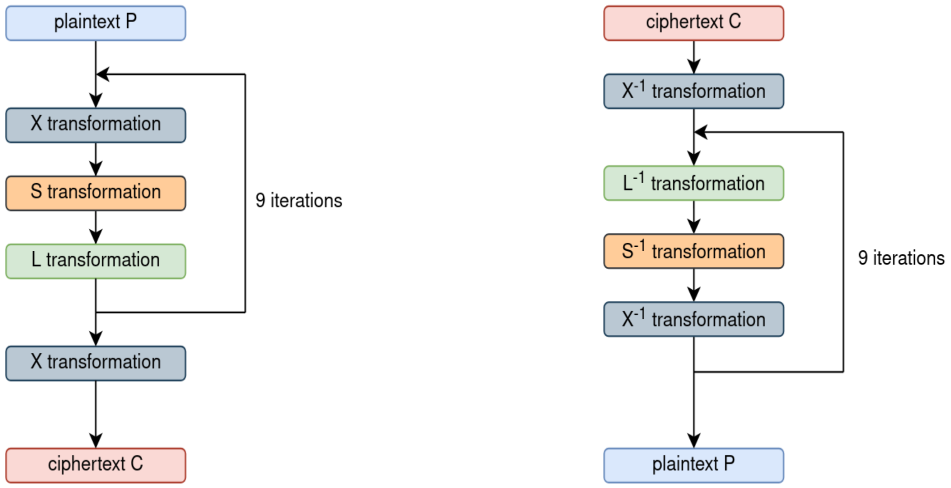

Figure 1. Encryption and decryption schemes of the Kuznechik algorithm

Define in the function **kuznechik.h** two external functions for encryption and decryption:

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

These functions take 2 pointers to the first byte of the state, and the first pointer is constant, because in the case of encryption it is plaintext, and in the case of decryption it is ciphertext, which should not change inside the encryption and decryption functions, respectively.

Let's define these functions inside the file **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);
   }
}
```

The implementation of the encryption and decryption function is the same as the above scheme. When encrypting, we use the forward order of iterative keys (from 1 to 10), and when decrypting, we use the reverse order of keys (from 10 to 1). Function **memcpy** just copies the data from the input block **blk** to the block **out_blk**.

2.1.3. X transformation

The X transformation is a simple addition modulo 2 (bitwise XOR) 128-bit strings: an array of state and an iterative key. This transformation makes the encryption and decryption procedure key-dependent (Kerkhoff's principle).

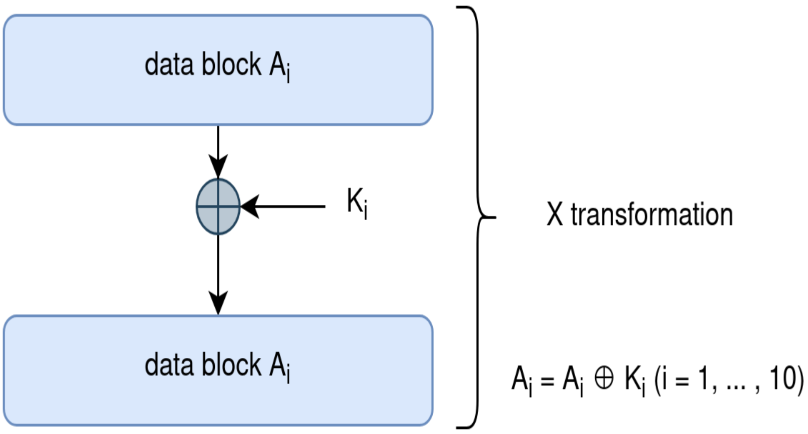

Figure 2. Transformation X of the Kuznechik encryption algorithm

When decrypting, the iterative keys go in reverse order, so the X transformation will look a little different when decrypting (see Figure 3).

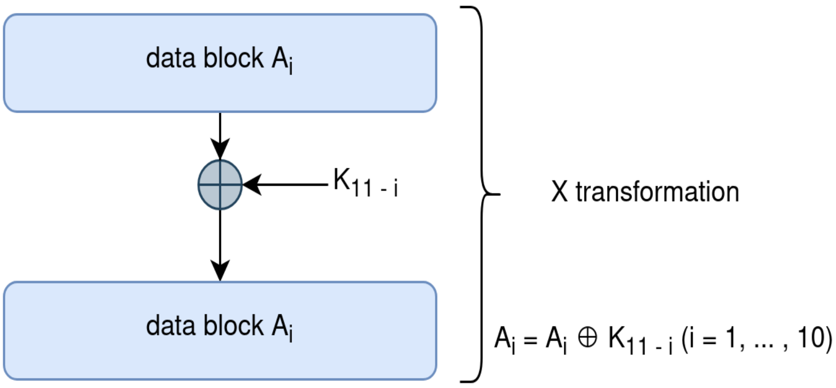

Figure 3. X transformation during decryption

In any case, when implementing the X transformation, the order of the keys does not matter (the X transformation does not depend on encryption and decryption).

```
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];
}
```

In this case, the function **X_transformation()** пtakes three pointers to bytes, the first 2 of which are pointers to the first byte of the iterative key and to the first byte of the state (the order is not important, the XOR operation is commutative), both pointers are constant so as not to change the values of their arguments. The third pointer is needed to refer to the first byte of the result state, where the result of addition modulo 2 of the first two arguments will be written. In the loop **for** we simply iterate through all 16 bytes of the state and key and produce their bitwise XOR.

2.1.4. S and S^{-1} transformation

The S transformation is a byte—by-byte substitution. This transformation introduces non-linearity into the encryption procedure (the principle of confusion), protecting the algorithm from differential and linear cryptanalysis (see Figure 4).

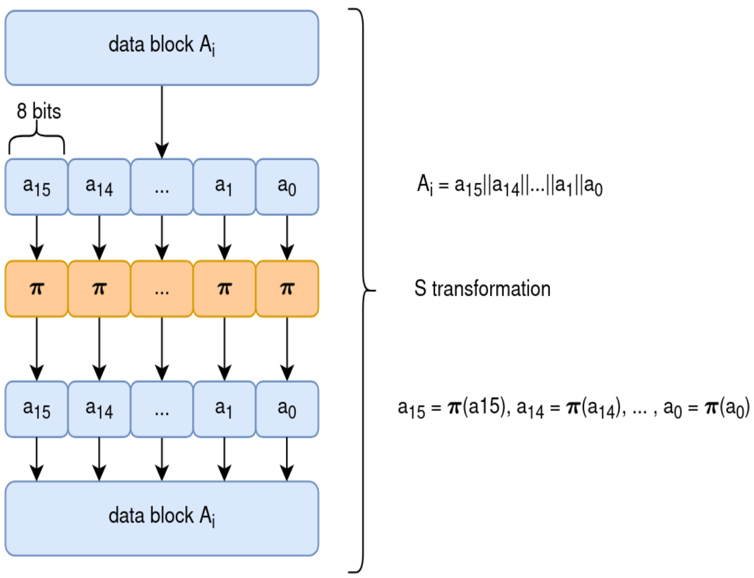

Figure 4. The S transformation

To implement the S transformation, we need to create a function that will accept 2 pointers: a constant pointer to the first byte of the input data (input state) and a pointer to the first byte of the output state, the result of which will be a substitution of all 16 bytes of the input state:

```
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]];
}
```

Reverse transformation S^{-1} similarly transformation S.

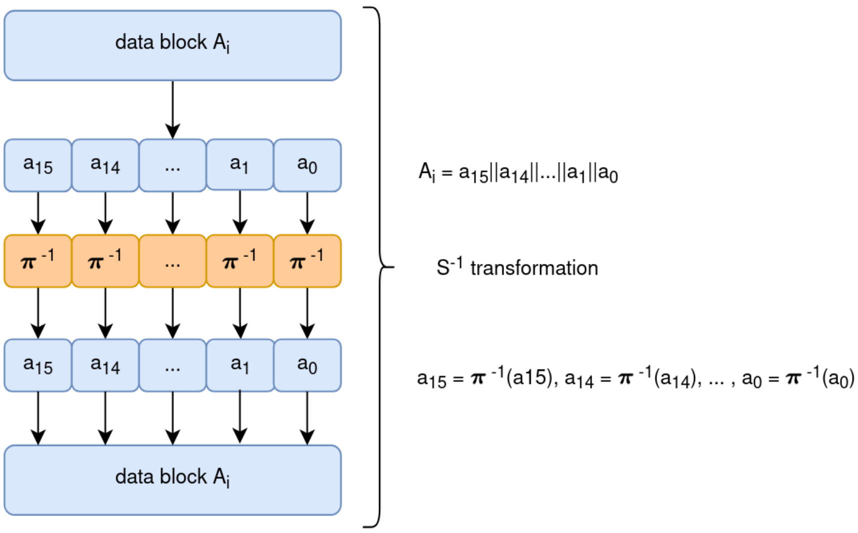

Figure 5. The inverse transformation of S

Realization transformation S^{-1} similar to the implementation of the transformation 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. L and L^{-1} transformation

Transformations L and L^{-1} they are based on transformations R and R^{-1} accordingly, according to the following formulas: $L(a) = R^{16}(a)$ and $L^{-1}(a) = (R^{-1})^{16}(a)$, where $a \in V_{128}$. These transformations are linear and necessary to form the dependence of each bit of the ciphertext block on all bits of the plaintext, eliminating the statistical characteristics of the plaintext (the principle of diffusion).

Transformation R(a) represents a shift of all 16 bytes of the state a to the right and changing the first byte according to the rule $a_0 = \ell(a_0, \ldots, a_{15})$, where $a_i \in V_8$, $i = 0, \ldots, 15$. The L transformation repeats the R transformation 16 times, thereby affecting all bytes of state a (Figure 6).

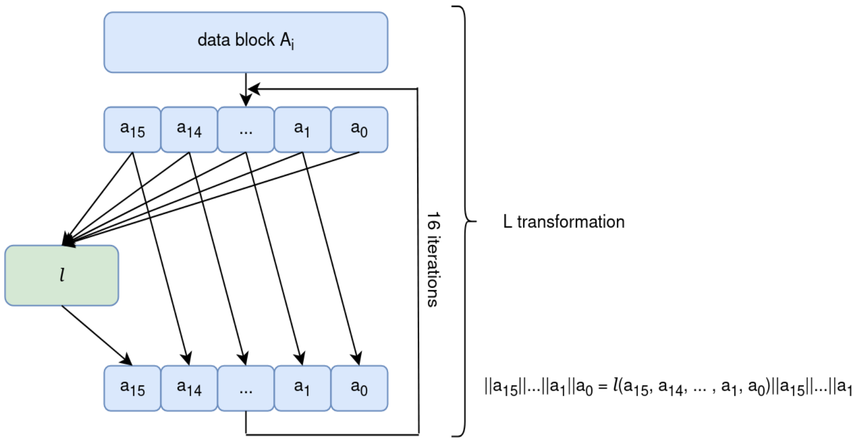

Figure 5. L transformation

To begin with, we implement the R transformation. Accordingly, the function must take a pointer to the first byte of the state, shift all bytes to the right and assign the result of a linear transformation to the first byte ℓ:

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

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

	state[0] = a_0;
}
```

Line state[i+1] = state[i] shifts all bytes to the right, and the next line calculates the linear transformation ℓ.

Transformation L then it is implemented trivially:

```
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);
}
```

Transformations are implemented in a similar way R^{-1} and L^{-1}.

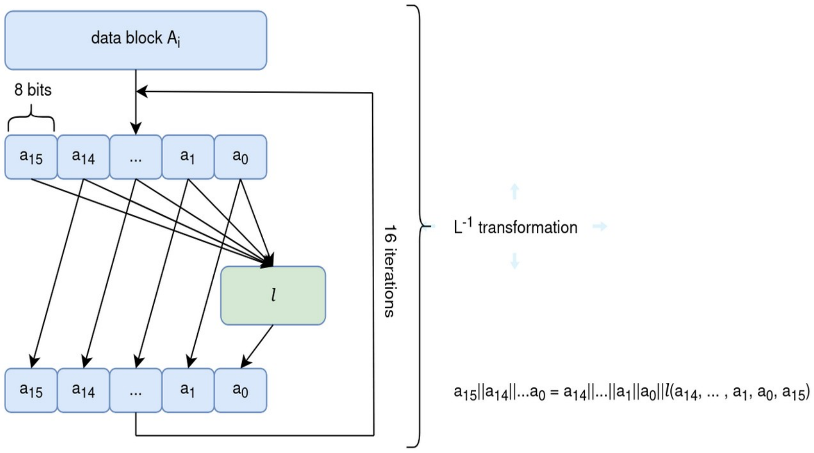

Figure 6. The inverse transformation of L

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

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

        state[15] = a_15;
}

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. Generating iterative keys

The procedure for generating iterative keys in the Kuznechik algorithm is an application of a 32-round Feistel network, where the composition of transformations acts as a function F LSX[C_i], where i — the iteration number, starting from zero. The first two keys are obtained from a 256-bit nasal key, the first iterative key is the first 128 bits of the main key, the second iterative key is the last 128 bits of the main key. Based on the first two iterative keys, the rest are built. On the 8th iteration, we generate iterative keys, we get 3 and 4 keys, on the 16th — 5 and 6 keys, etc.

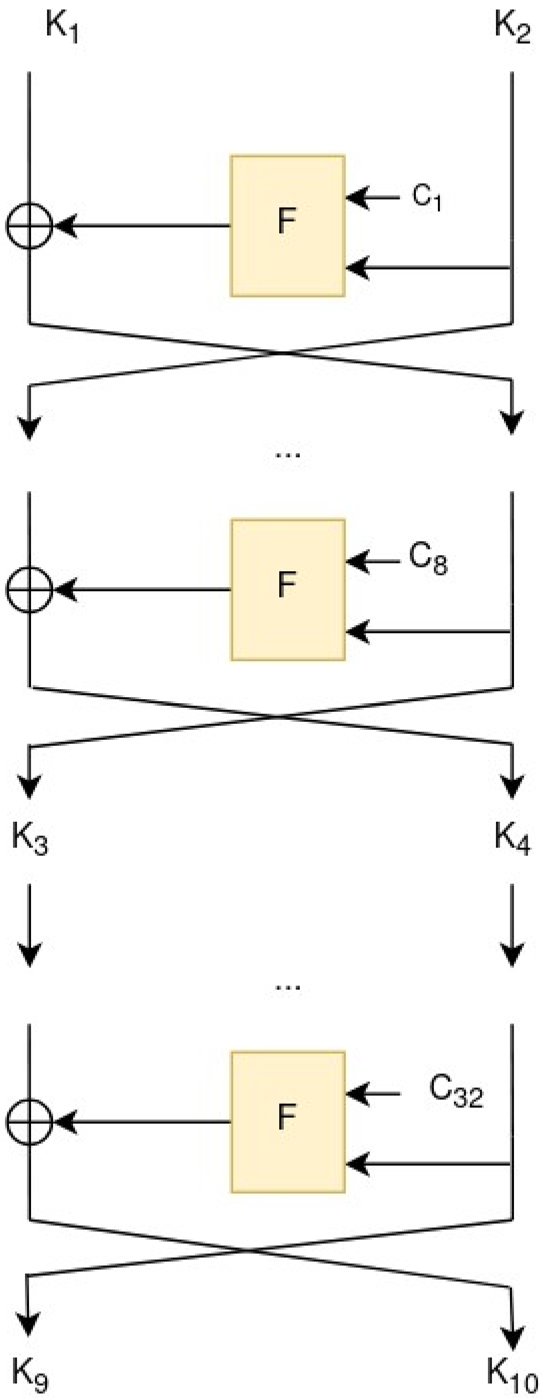

Figure 7. Generating iterative keys

We implement the function F, which will take 2 pointers to the first byte of the input 128-bit keys, 2 pointers to the first byte of the generated keys, as well as an iterative constant.

```
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);
}
```

The function for generating iterative constants is given below.

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

        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. Source code file listings

2.2.1. File 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
```

2.2.2. File 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(a0, a1, ..., a15) =
	148*a0 + 32*a1 + 133*a2 + 16*a3 + 194*a4 + 192*a5 + 1*a6 + 251*a7 + 1*a8 + 192*a9 +
	+ 194*a10 + 16*a11 + 133*a12 + 32*a13 + 148*a14 + 1*a15.
*/
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];

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

	state[0] = a_0;
 }


/*
	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];

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

	state[15] = a_15;
 }

/*
	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);
    }
}
```