SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE U VARAŽDINU

Seminarski rad iz kolegija Strukture podataka i algoritmi

Enkripcijski algoritam „Kuznyechik“

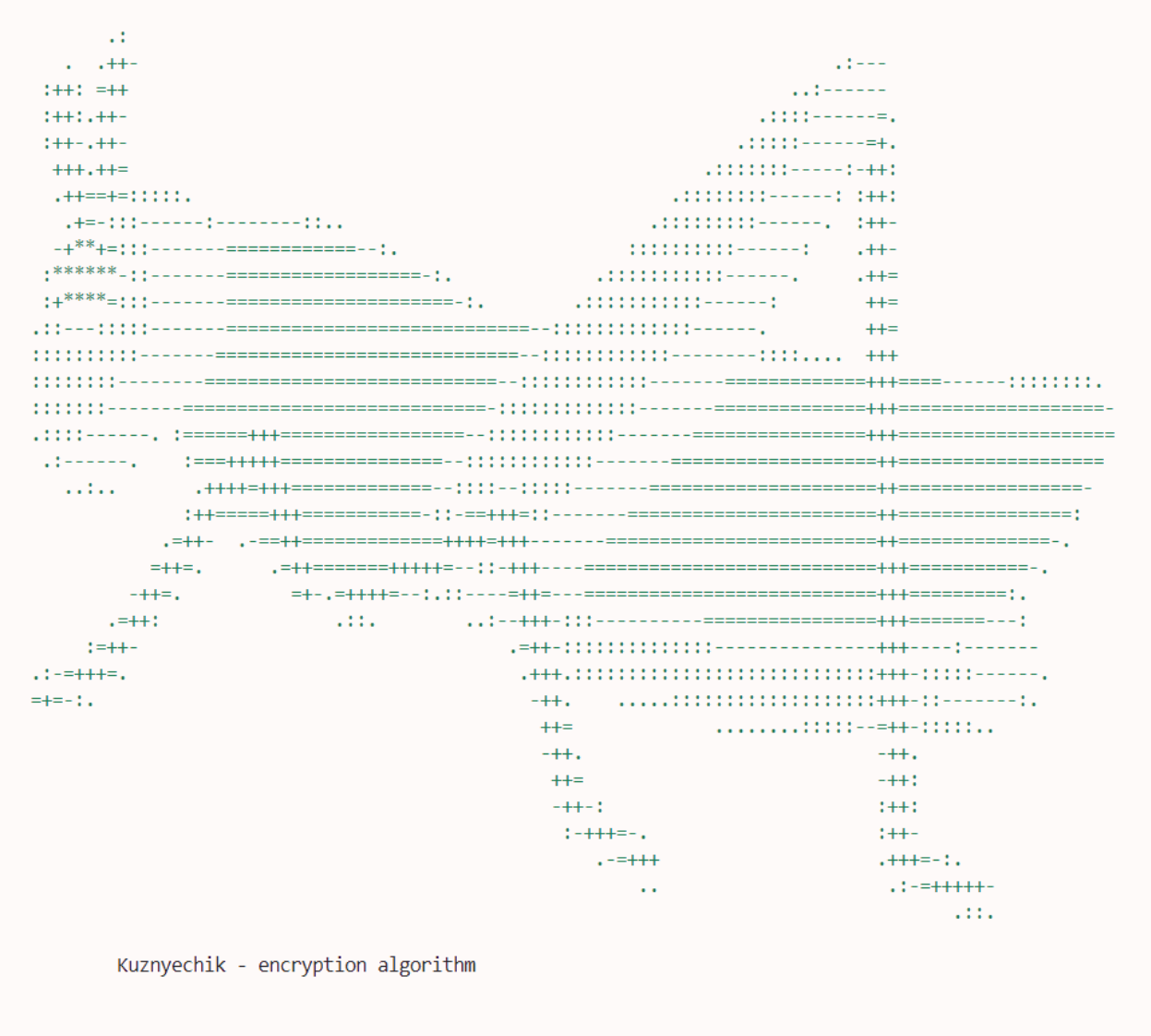
(GOST R 34.12-2015)

Mentor: Prof. dr. sc. Alen Lovrenčić

Pavel Folnović

JMBAG**:** 0016166246

U Varaždinu, 2025.



Slika 1 - Kuznyechik („skakavac“) od ASCII charactera (Samostalna izrada u alatu www.asciiart.eu, ispis u VS Code Ubuntu terminalu)

Sadržaj

1. Uvod

Motivacija za odabir Kuznyechika

Odabir Kuznyechik blok šifre kao teme ovog seminarskog rada motiviran je njezinim jedinstvenim značajkama i značajem unutar modernih kriptografskih standarda. Kuznyechik, kao dio ruskog GOST R 34.12-2015 standarda, nudi zanimljivu priliku za proučavanje algoritama koji se koriste izvan zapadnih kriptografskih standarda. Osim toga, njen dizajn kombinira jednostavnost i sigurnost, čime pruža vrijednu osnovu za dublje razumijevanje kriptografskih sustava i njihovih primjena. Motivacija također proizlazi iz želje za povezivanjem teorijskih znanja iz područja struktura podataka i algoritama s praktičnom implementacijom složenih kriptografskih metoda.

Važnost blok šifri u suvremenoj kriptografiji

Blok šifra je kriptografski algoritam koji obrađuje podatke u blokovima fiksne duljine, pretvarajući ih u šifrirane blokove pomoću ključne funkcije. Blok šifre igraju ključnu ulogu u osiguravanju povjerljivosti podataka u komunikacijskim i računalnim sustavima. Njihova primjena obuhvaća širok spektar područja, uključujući zaštitu podataka u prijenosu, sigurnost baza podataka te kriptografske protokole poput TLS-a (Transfer Layer Security) (ISO/IEC (2006)). Zbog svoje sposobnosti da osiguraju visok stupanj sigurnosti uz prihvatljive performanse, blok šifre poput Kuznyechika su neizostavan alat u aplikacijama koje izravno ili neizravno koristimo svakodnevno. Uz sve veću potrebu za zaštitom osjetljivih podataka u digitalnom dobu, proučavanje i razumijevanje blok šifri postaje od kritične važnosti za buduće stručnjake u području računalne sigurnosti.

Ciljevi seminarskog rada

Cilj ovog seminarskog rada je detaljno proučiti teorijske osnove i dizajn Kuznyechik blok šifre, uz razvoj i implementaciju algoritama za enkripciju i dekripciju u programskom jeziku C++. Implementacija će biti popraćena dijagramima toka koji će vizualno prikazati

procese enkripcije i dekripcije. Osim toga, rad će analizirati ispravnost algoritma te vremensku složenost istog.

2. Pregled Kuznyechik blok šifre

Povijest i razvoj

Kuznyechik, što na ruskom znači "skakavac", moderna je simetrična blok šifra koju je razvila ruska tvrtka InfoTeCS JSC kao dio nastojanja da se unaprijede nacionalni standardi za zaštitu podataka. Objavljena je 2015. godine kao sastavni dio GOST R 34.12-2015, koji je uveden kako bi zamijenio prethodni GOST 28147-89 standard, poznat kao Magma, koji se koristio od 1989. godine. Stari standard više nije mogao zadovoljiti rastuće sigurnosne zahtjeve digitalnog doba, pa je razvijen Kuznyechik kako bi osigurao bolje performanse, veću sigurnost i kompatibilnost s modernim tehnologijama (ISO/IEC (2015)).

Kuznyechik je dio ruskog kriptografskog okvira koji je certificiran od strane Federalne službe za tehničku i izvoznu kontrolu (FSTEC) i Federalne službe sigurnosti (FSB). Njegova uspostava dio je šireg trenda jačanja nacionalne kibernetičke sigurnosti i neovisnosti od međunarodnih kriptografskih standarda poput AES-a (Advanced Encryption Standard). Osim što je standardiziran u Rusiji, Kuznyechik je također opisan u RFC 7801, što omogućuje njegovu potencijalnu integraciju u globalne kriptografske sustave. (Dolmatov, 2016, Section 2)

Ključne značajke Kuznyechika

Kuznyechik se ističe nizom značajki koje ga čine jednom od najmodernijih i najsigurnijih blok šifri u svom razredu. Jedna od ključnih prednosti je duljina ključa od 256 bita, koja pruža otpornost na brute-force napade daleko iznad trenutnih praktičnih mogućnosti računalnih sustava. Uz to, blokovi podataka od 128 bita omogućuju učinkovitu obradu velikih količina podataka bez kompromitiranja sigurnosti.

Struktura algoritma temelji se na mreži zamjena i permutacija (SPN), koja koristi 10 rundi za enkripciju i dekripciju. Svaka runda uključuje niz transformacija koje kombiniraju

nelinearne i linearne operacije. Nelinearne transformacije implementirane su pomoću S-Box funkcija, koje su posebno dizajnirane da budu otporne na napredne tehnike kriptoanalize, poput linearne i diferencijalne analize. Linearne transformacije koriste matricu koja povećava difuziju, osiguravajući da se promjene u ulaznim podacima ili ključu brzo šire kroz cijeli blok. (Dolmatov, 2016, Section 4)

Kuznyechik je optimiziran za rad u softveru i hardveru. Njegov dizajn omogućuje brzu implementaciju na širokom rasponu platformi, uključujući uređaje s ograničenim resursima. Zahvaljujući njegovoj modularnosti, može se prilagoditi specifičnim aplikacijama, od zaštite podataka u prijenosu do implementacije u sigurnosne protokole.

Usporedba s drugim blok šiframa

Usporedimo li Kuznyechik s AES-om, koji je globalni standard u kriptografiji, vidimo neke sličnosti, ali i ključne razlike. Oba algoritma koriste blokove podataka od 128 bita, no Kuznyechik koristi samo jednu duljinu ključa od 256 bita, dok AES omogućuje varijabilnu duljinu ključa od 128, 192 ili 256 bita. AES koristi mrežu zamjena i permutacija s 10, 12 ili 14 rundi, ovisno o duljini ključa, dok Kuznyechik koristi fiksnih 10 rundi. Kuznyechikova S-Box funkcija ima jedinstveni dizajn koji je optimiziran za otpornost na suvremene napade, dok AES koristi S-Box funkcije izvedene iz inverzije u polju $GF(2^8)$.

U odnosu na stariji ruski standard GOST 28147-89 (Magma), Kuznyechik donosi značajna poboljšanja. Magma koristi 64-bitne blokove i Feistelovu strukturu (dijeljenje podataka na blokove), što ju čini manje otpornom na suvremene napade i manje učinkovitom pri obradi velikih količina podataka. Kuznyechik prelazi na 128-bitne blokove i SPN arhitekturu, što omogućuje bolju sigurnost, veću brzinu i moderniju otpornost na kriptoanalizu.

3. Teorijske osnove

Matematički principi na kojima se temelji Kuznyechik

Kuznyechik blok šifra oslanja se na niz matematičkih principa i transformacija koje osiguravaju visoku sigurnost i otpornost na različite vrste kriptoanalitičkih napada. Temelj algoritma su operacije izvedene u konačnim poljima $GF(2^8)$, nelinearne bijekcije, linearne transformacije te procesi generiranja ključeva. Svaka komponenta dizajnirana je kako bi osigurala difuziju i konfuziju podataka, ključne za učinkovitost kriptografskih algoritama.

1. Polje $GF(2^8)$

Polje konačnih elemenata $GF(2^8)$ koristi se za implementaciju osnovnih operacija unutar Kuznyechik algoritma. Polje se definira preko redukcijskog polinoma $p(x) = x^8 + x^7 + x^6 + x + 1$, koji osigurava da svi elementi polja mogu biti predstavljeni kao polinomi stupnja manjeg od 8. To omogućuje rad s binarnim brojevima, gdje svaka vrijednost može biti predstavljena kao vektor duljine 8 bita.

Operacije unutar $GF(2^8)$ uključuju:

- Zbrajanje ($\oplus$): Provodi se bitwise XOR-om između dva binarna broja, čime se osigurava da je svaka operacija reverzibilna.
- Množenje: Realizira se kao standardno množenje polinoma, uz redukciju rezultata pomoću $p(x)$. Kada rezultat množenja prelazi 8 bita, višak se uklanja XOR-om s vrijednošću ($0xC3$), koja predstavlja koeficijente polinoma $p(x)$.

Ovaj sustav omogućuje sigurnu manipulaciju podacima u svakom koraku algoritma i osigurava homogeni matematički model za sve transformacije.

2. X-transformacija (XOR operacija)

X-transformacija, također poznata kao ključna miješanja, primjenjuje ekskluzivnu logičku operaciju XOR između trenutnog stanja i rundnog ključa. Matematički je definirana kao

$(X_k(a) = k \oplus a)$, gdje su *k* (ključ) i ***a*** (stanje) 128-bitni vektori. Svaki bit u ***a*** i *k* obrađuje se neovisno, čime se osigurava da je operacija brza i reverzibilna.

Ova transformacija je ključna za integraciju sigurnosnih elemenata iz ključeva u podatke. Pruža početnu i završnu zaštitu podataka u svakom krugu enkripcije. Reverzibilnost XOR operacije omogućuje jednostavno vraćanje na prethodno stanje tijekom dekripcije.

3. S-transformacija (Substitucija)

S-transformacija unosi nelinearnost u algoritam, koristeći unaprijed definiranu S-Box tablicu (π, odnosno Pi) koja je bijektivna, tj. svaka vrijednost ima jedinstven ekvivalent. Ova tablica pretvara svaki bajt ulaza u odgovarajući bajt izlaza prema fiksnom skupu pravila. Matematički se izražava kao:

$$S(a) = \big(\pi(a_0), \pi(a_1), \dots, \pi(a_{15})\big)$$

gdje su $a_0, a_1, \dots, a_{15}$ elementi 128-bitnog stanja podijeljenog na 16 bajtova.

S-Box je dizajniran tako da bude otporan na linearne i diferencijalne kriptoanalize, osiguravajući minimalnu mogućnost predikcije izlaznih vrijednosti na temelju ulaznih. Inverzna S-transformacija S^{-1} koristi inverznu tablicu π^{-1} za povratak iz enkriptiranog u originalno stanje tijekom dekripcije.

4. R-transformacija (Linearna transformacija)

R-transformacija implementira linearnu difuziju, gdje svaki bajt trenutnog stanja doprinosi konačnoj vrijednosti novog bajta pomoću fiksnog vektora *c*:

$$a'_{15} = \sum_{i=0}^{15} c_i \cdot a_i$$

Množenje se provodi unutar $GF(2^8)$, dok se svi rezultati zbrajaju pomoću XOR operacije. Nakon toga, stanje se ciklički pomiče desno: $(a_0, a_1, \dots, a_{15}) \rightarrow (a'_{15}, a_0, \dots, a_{14})$

Ovaj pomak osigurava da se informacije iz svih dijelova stanja ravnomjerno raspoređuju kroz cijeli vektor, čime se povećava otpornost na kriptoanalizu.

5. L-transformacija

L-transformacija je iterativna primjena R-transformacije 16 puta, što dodatno jača difuziju. Matematički:

$$L(a) = R^{16}(a)$$

gdje je *a* 128-bitni ulazni vektor.

6. Generiranje ključeva

Master ključ duljine 256 bita dijeli se na dva 128-bitna dijela (K_1, K_2).Generiranje rundnih ključeva koristi konstantne vrijednosti C_i, izračunate kao:

$$C_i = L\big(\mathrm{Vec}_{128}(i)\big).$$

Proces uključuje iterativnu primjenu transformacije *F*, definirane kao:

$$F[C_i](a_1, a_0) = \Big(L\Big(S\big(X[C_i](a_1)\big)\Big) \oplus a_0, a_1\Big)$$

Ovaj postupak generira 10 rundnih ključeva koji se koriste tijekom enkripcije i dekripcije.

7. Enkripcija

Enkripcijski proces uključuje primjenu niza transformacija nad ulaznim podacima i rundnim ključevima:

$$E(a) = X[K_{10}]L\left(S\left(X[K_9]\left(L\left(S\left(X[K_8]\left(\dots\left(L\left(S\big(X[K_1](a)\big)\right)\right)\right)\right)\right)\right)\right)\right),$$

gdje su $K_1, \dots, K_{10}$ rundni ključevi. Svaki krug uključuje *X, S* i *L* transformacije, čime se osigurava potpuna konfuzija i difuzija podataka.

8. Dekripcija

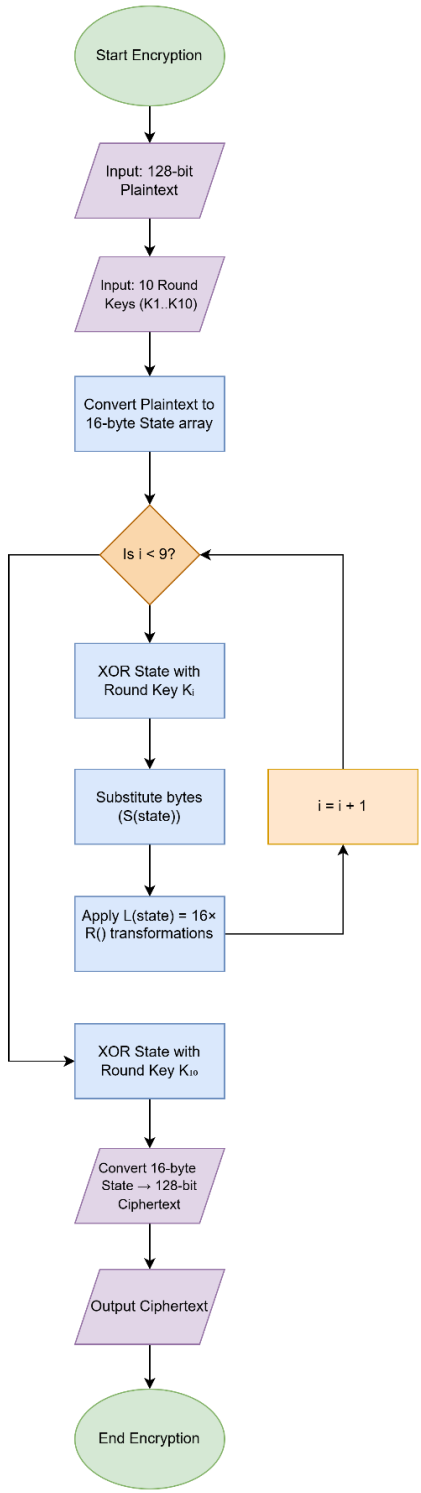
Dekripcija koristi inverzne transformacije kako bi rekonstruirala originalne podatke:

$$D(a) = X[K_1]L^{-1}\left(S^{-1}\left(X[K_2]\left(L^{-1}\left(S^{-1}\left(\dots\left(X[K_{10}](a)\right)\right)\right)\right)\right)\right)$$

Ovaj postupak koristi inverzne tablice i operacije te prati obrnut redoslijed transformacija primijenjenih tijekom enkripcije.

4. Implementacija

Dijagram toka za enkripciju



Slika 2 - Dijagram toka za enkripciju (Samostalna izrada u alatu draw.io, 2025.)

Programski kod za enkripciju

```cpp
#include <iostream>
#include <array>
#include <cstdint>
#include <vector>
#include <iomanip>
#include <cstring>

using namespace std;

using uint128_t = std::array<uint64_t, 2>;
using uint256_t = std::array<uint64_t, 4>;

void generateRoundKeys(const uint256_t& masterKey, std::array<uint128_t, 10>& roundKeys);
void F(const uint8_t* key, uint8_t* a1, uint8_t* a0);
void X(uint8_t* state, const uint8_t* key);
void L(uint8_t* state);
void L_inv(uint8_t* state);
void S(uint8_t* state);
void S_inv(uint8_t* state);
void R(uint8_t* state);
void R_inv(uint8_t* state);
uint8_t gf_mul(uint8_t a, uint8_t b);
```

Slika 3 – Zaglavlja, tipovi podataka i funkcije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

1. Uključivanje zaglavlja (#include direktive)

Prvi dio koda uključuje osnovne zaglavlje datoteke koje omogućuju korištenje različitih funkcionalnosti u programu. Evo što svako od ovih zaglavlja pruža:

- *#include <iostream>* omogućuje korištenje standardnih ulazno-izlaznih funkcija, poput std::cout za ispis na konzolu.
- *#include <array>* omogućuje korištenje std::array, što je ključna struktura podataka u ovom programu za rad s 128-bitnim i 256-bitnim blokovima.
- *#include <cstdint>* omogućuje korištenje precizno definiranih tipova podataka sa specifičnim duljinama, poput uint8_t i uint64_t, koji predstavljaju 8-bitne i 64-bitne cijele brojeve bez predznaka.
- *#include <vector>* omogućuje korištenje std::vector za dinamičke nizove, iako se u ovom dijelu koda primarno koristi za generiranje konstanti.

- *#include <iomanip>* omogućuje formatiranje izlaznih podataka, što je korisno kod ispisa heksadecimalnih vrijednosti.
- *#include <cstring>* omogućuje korištenje funkcija za manipulaciju memorijom, poput memcpy i memmove, koje se koriste za kopiranje i pomicanje podataka unutar memorije.

2. Definicija tipova podataka

Sljedeći dio koda koristi using direktive za definiranje novih tipova podataka koji će se koristiti kroz cijeli algoritam. Ovo olakšava rad s velikim blokovima podataka i ključevima.

- *using uint128_t = std::array<uint64_t, 2>;*
  Ova direktiva definira uint128_t kao niz od dva 64-bitna cijela broja (uint64_t). Budući da standardni C++ nema ugrađeni 128-bitni tip podataka, ovaj niz se koristi za reprezentaciju 128-bitnih blokova podataka. Svaki blok podataka u algoritmu Kuznyechik ima duljinu od 128 bita.
- *using uint256_t = std::array<uint64_t, 4>;*
  Ova direktiva definira uint256_t kao niz od četiri 64-bitna cijela broja. Ovo se koristi za reprezentaciju ključeva duljine 256 bita. Master ključ koji se koristi za generiranje rundnih ključeva ima upravo ovu duljinu.

Ove definicije tipova omogućuju jednostavnije i čitljivije rukovanje velikim brojevima, umjesto da se koriste pojedinačni bajtovi ili drugačije strukture podataka. Na primjer, rad s uint128_t omogućuje da blok podataka tretiramo kao jednu cjelinu, što pojednostavljuje manipulaciju tijekom enkripcije i dekripcije.

3. Deklaracije funkcija (void i uint8_t)

Sljedeći dio koda sadrži deklaracije funkcija koje će se kasnije definirati u programu. Evo pregleda svake funkcije, detaljnije će biti objašnjene u nastavku:

- *void generateRoundKeys(const uint256_t& masterKey, std::array<uint128_t, 10>& roundKeys);*

- *void F(const uint8_t* key, uint8_t* a1, uint8_t* a0);*
- *void X(uint8_t* state, const uint8_t* key);*
- *void L(uint8_t* state);*
- *void L_inv(uint8_t* state);*
- *void S(uint8_t* state);*
- *void S_inv(uint8_t* state);*
- *void R(uint8_t* state);*
- *void R_inv(uint8_t* state);*
- *uint8_t gf_mul(uint8_t a, uint8_t b);*

```
24  const uint8_t Pi[256] = {
25      252, 238, 221,  17, 207, 110,  49,  22, 251, 196, 250, 218,  35, 197,   4,  77,
26      233, 119, 240, 219, 147,  46, 153, 186,  23,  54, 241, 187,  20, 205,  95, 193,
27      249,  24, 101,  90, 226,  92, 239,  33, 129,  28,  60,  66, 139,   1, 142,  79,
28        5, 132,   2, 174, 227, 106, 143, 160,   6,  11, 237, 152, 127, 212, 211,  31,
29      235,  52,  44,  81, 234, 200,  72, 171, 242,  42, 104, 162, 253,  58, 206, 204,
30      181, 112,  14,  86,   8,  12, 118,  18, 191, 114,  19,  71, 156, 183,  93, 135,
31       21, 161, 150,  41,  16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177,
32       50, 117,  25,  61, 255,  53, 138, 126, 109,  84, 198, 128, 195, 189,  13,  87,
33      223, 245,  36, 169,  62, 168,  67, 201, 215, 121, 214, 246, 124,  34, 185,   3,
34      224,  15, 236, 222, 122, 148, 176, 188, 220, 232,  40,  80,  78,  51,  10,  74,
35      167, 151,  96, 115,  30,   0,  98,  68,  26, 184,  56, 130, 100, 159,  38,  65,
36      173,  69,  70, 146,  39,  94,  85,  47, 140, 163, 165, 125, 105, 213, 149,  59,
37        7,  88, 179,  64, 134, 172,  29, 247,  48,  55, 107, 228, 136, 217, 231, 137,
38      225,  27, 131,  73,  76,  63, 248, 254, 141,  83, 170, 144, 202, 216, 133,  97,
39       32, 113, 103, 164,  45,  43,   9,  91, 203, 155,  37, 208, 190, 229, 108,  82,
40       89, 166, 116, 210, 230, 244, 180, 192, 209, 102, 175, 194,  57,  75,  99, 182
41  };
42
43  const uint8_t Pi_inv[256] = {
44      165,  45,  50, 143,  14,  48,  56, 192,  84, 230, 158,  57,  85, 126,  82, 145,
45      100,   3,  87,  90,  28,  96,   7,  24,  33, 114, 168, 209,  41, 198, 164,  63,
46      224,  39, 141,  12, 130, 234, 174, 180, 154,  99,  73, 229,  66, 228,  21, 183,
47      200,   6, 112, 157,  65, 117,  25, 201, 170, 252,  77, 191,  42, 115, 132, 213,
48      195, 175,  43, 134, 167, 177, 178,  91,  70, 211, 159, 253, 212,  15, 156,  47,
49      155,  67, 239, 217, 121, 182,  83, 127, 193, 240,  35, 231,  37,  94, 181,  30,
50      162, 223, 166, 254, 172,  34, 249, 226,  74, 188,  53, 202, 238, 120,   5, 107,
51       81, 225,  89, 163, 242, 113,  86,  17, 106, 137, 148, 101, 140, 187, 119,  60,
52      123,  40, 171, 210,  49, 222, 196,  95, 204, 207, 118,  44, 184, 216,  46,  54,
53      219, 105, 179,  20, 149, 190,  98, 161,  59,  22, 102, 233,  92, 108, 109, 173,
54       55,  97,  75, 185, 227, 186, 241, 160, 133, 131, 218,  71, 197, 176,  51, 250,
55      150, 111, 110, 194, 246,  80, 255,  93, 169, 142,  23,  27, 151, 125, 236,  88,
56      247,  31, 251, 124,   9,  13, 122, 103,  69, 135, 220, 232,  79,  29,  78,   4,
57      235, 248, 243,  62,  61, 189, 138, 136, 221, 205,  11,  19, 152,   2, 147, 128,
58      144, 208,  36,  52, 203, 237, 244, 206, 153,  16,  68,  64, 146,  58,   1,  38,
59       18,  26,  72, 104, 245, 129, 139, 199, 214,  32,  10,   8,   0,  76, 215, 116
60  };
```

Slika 4 - S-box i inverzna S-Box tablica, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

4. S-box i inverzna S-Box tablica

S-Box (Substitution Box) je unaprijed definirana tablica koja se koristi za zamjenu (substituciju) svakog ulaznog bajta s odgovarajućim izlaznim bajtom. U Kuznyechik algoritmu, S-Box tablica *Pi* sadrži 256 unaprijed definiranih vrijednosti, pri čemu svaka ulazna vrijednost od 0 do 255 ima jedinstveni izlaz. Ova zamjena osigurava da algoritam postane nelinearan, što povećava njegovu otpornost na različite oblike kriptoanalize, poput linearne i diferencijalne analize.

Primjerice, ako je ulazni bajt $a_i = 17$, funkcija *S* će ga zamijeniti s vrijednošću iz S-Box tablice koja se nalazi na indeksu 17, a to je broj 207. Ova zamjena se vrši za svaki bajt unutar 128-bitnog bloka.

Nelinearnost koju uvodi S-Box ključna je za zaštitu algoritma od napada koji koriste linearne aproksimacije ili predvidljive obrasce. Bez S-Boxa, transformacije unutar algoritma bile bi linearne i stoga lako podložne analitičkim napadima. S-Box uvodi složenost u algoritam tako da male promjene u ulaznim podacima rezultiraju velikim promjenama u izlaznim podacima, što povećava sigurnost.

```
63  const uint16_t field_polynomial = 0x1C3; // x^8 + x^7 + x^6 + x + 1
64
65  uint8_t gf_mul(uint8_t a, uint8_t b) {
66      uint8_t p = 0;
67      uint8_t carry;
68
69      for (int i = 0; i < 8; i++) {
70          if (b & 1) {
71              p ^= a;
72          }
73          carry = a & 0x80;
74          a <<= 1;
75          if (carry) {
76              a ^= 0xC3;
77          }
78          b >>= 1;
79      }
80      return p;
81  }
```

Slika 5 - Množenje u konačnom polju GF(2^8), isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

5. Množenje u konačnom polju GF(2^8)

Prikazani dio koda implementira funkciju *gf_mul*, koja provodi množenje dvaju elemenata unutar konačnog polja GF(2^8) koristeći polinomsku aritmetiku. Budući da standardne aritmetičke operacije ne vrijede u konačnim poljima na isti način kao u klasičnoj matematici, potrebno je provesti množenje uz redukciju prema fiksnom polinomu.

Polje GF(2^8) koristi polinom reda osam definiran kao $x^8 + x^7 + x^6 + x + 1$, koji je zapisan u obliku heksadecimalne vrijednosti 0x1C3 (gdje 0x1C3 predstavlja binarni zapis 111000011, pri čemu svaki bit označava prisutnost odgovarajuće potencije polinoma: x^8, x^7, x^6, x, 1). Ovaj polinom osigurava da svi rezultati množenja ostaju unutar 8-bitnog raspona. Funkcija *gf_mul* uzima dva ulazna bajta, *a* i *b*, te ih množi primjenom binarne aritmetike i redukcijskog polinoma.

Množenje se provodi iterativno kroz osam koraka unutar for petlje. Svaki korak provjerava je li najmanje značajan bit broja *b* postavljen. Ako jest, vrijednost *a* se dodaje trenutnom proizvodu *p* korištenjem XOR operacije, jer zbrajanje u GF(2^8) odgovara XOR-u. Zatim se broj *a* pomiče ulijevo, što odgovara množenju s *x*. Ako pomak uzrokuje prelazak preko granice od osam bita, provodi se redukcija pomoću polinoma 0xC3 kako bi se rezultat vratio unutar polja.

Funkcija na kraju vraća konačni proizvod *p*. Ova metoda omogućuje učinkovito množenje u konačnom polju, koje je ključno za linearne transformacije unutar Kuznyechik algoritma. Množenje u GF(2^8) osigurava ravnomjerno širenje informacija kroz blok podataka, čime se povećava sigurnost algoritma.

```
84  void S(uint8_t* state) {
85      for (int i = 0; i < 16; i++) {
86          state[i] = Pi[state[i]];
87      }
88  }
89
90  void L(uint8_t* state) {
91      for (int i = 0; i < 16; i++) {
92          R(state);
93      }
94  }
```

Slika 6 - Funkcije S-Transformacije i L-Transformacije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

6. Funkcije S-Transformacije i L-Transformacije

Prikazane funkcije *S* i *L* provode ključne transformacije nad trenutnim stanjem podataka unutar algoritma Kuznyechik. Funkcija *S* provodi nelinearnu transformaciju pomoću S-Box tablice, dok funkcija *L* osigurava difuziju podataka kroz višestruke linearne transformacije. Obje funkcije djeluju na 128-bitnom bloku podataka podijeljenom u 16 bajtova, što odgovara internom stanju algoritma.

Funkcija *S* implementira nelinearnu zamjenu svakog bajta u bloku koristeći S-Box tablicu *Pi*. Svaki element stanja, označen kao *state[i]*, zamjenjuje se odgovarajućom vrijednošću iz S-Box tablice na indeksu *state[i]*.

S druge strane, funkcija *L* provodi višestruku primjenu linearne transformacije pomoću funkcije *R*. Petlja unutar funkcije *L* poziva funkciju *R* 16 puta, pri čemu svaki poziv primjenjuje pomak i množenje nad trenutnim stanjem.

```
96  void R(uint8_t* state) {
97      static const uint8_t c[16] = {
98          0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01, 0xfb,
99          0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01
100     };
101     uint8_t a15_new = 0;
102
103     for (int i = 0; i < 16; i++) {
104         a15_new ^= gf_mul(state[i], c[i]);
105     }
106
107     memmove(state + 1, state, 15);
108     state[0] = a15_new;
109 }
```

Slika 7 - Funkcija R-Transformacije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

7. Funkcija R-Transformacije

Ova funkcija uzima stanje predstavljeno nizom od 16 bajtova i provodi permutaciju koja uvodi difuziju podataka, što je ključan korak u osiguravanju kriptografske sigurnosti.

Prvo što funkcija radi jest definiranje statičkog niza konstantnih vrijednosti *c*, koji sadrži 16 unaprijed određenih vrijednosti. Ovaj niz konstantnih vrijednosti koristi se za množenje s odgovarajućim bajtovima iz trenutnog stanja podataka. Svaka od ovih konstanti doprinosi procesu izračuna novog prvog bajta bloka, što osigurava da svi dijelovi bloka podataka utječu na konačni rezultat.

Nakon što je definiran niz konstanti, funkcija inicijalizira varijablu *a15_new* s vrijednošću nula. Ta varijabla predstavlja novu vrijednost koja će biti postavljena na prvo mjesto u bloku nakon što se završi permutacija. Kako bi se izračunala ova nova vrijednost, funkcija prolazi kroz svih 16 bajtova u trenutnom stanju. Svaki bajt se množi odgovarajućom konstantom iz niza *c* pomoću funkcije *gf_mul*, koja provodi množenje u Galoisovom polju. Rezultat svakog množenja zatim se XOR-om akumulira u varijabli *a15_new*.

Nakon što je izračunata nova vrijednost za prvi bajt, funkcija provodi pomicanje svih bajtova u bloku ulijevo za jednu poziciju. Ova operacija se provodi pomoću funkcije *memmove*, koja kopira svih 15 preostalih bajtova na njihovu novu poziciju unutar niza. Na kraju, izračunata vrijednost *a15_new* postavlja se na prvo mjesto u bloku.

```
141  void X(uint8_t* state, const uint8_t* key) {
142      for (int i = 0; i < 16; i++) {
143          state[i] ^= key[i];
144      }
145  }
146
147  void F(const uint8_t* key, uint8_t* a1, uint8_t* a0) {
148      uint8_t temp[16];
149
150      for (int i = 0; i < 16; i++) {
151          temp[i] = a1[i] ^ key[i];
152      }
153
154      S(temp);
155
156      L(temp);
157
158      for (int i = 0; i < 16; i++) {
159          temp[i] ^= a0[i];
160      }
161
162      memcpy(a0, a1, 16);
163      memcpy(a1, temp, 16);
```

Slika 8 - Funkcije X-Transformacije i F-Transformacije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

8. Funkcije X-Transformacije i F-Transformacije

Funkcija *X* predstavlja osnovnu operaciju XOR između dva 128-bitna bloka podataka. Ova funkcija uzima trenutno stanje *state*, koje se sastoji od niza od 16 bajtova, i ključ *key* iste veličine. Petlja koja se nalazi unutar funkcije prolazi kroz svih 16 bajtova trenutnog stanja, a zatim svaki bajt u stanju XOR-ira s odgovarajućim bajtom iz ključa. Rezultat ove operacije pohranjuje se u varijabli *state*. XOR je u kriptografiji česta operacija jer jednostavno i učinkovito miješa podatke i ključ, pri čemu svaka promjena u ključa ili podacima značajno utječe na izlaz.

Funkcija *F* predstavlja složeniju transformaciju koja uključuje niz ključnih operacija algoritma Kuznyechik, kombinirajući funkcije *X*, *S* i *L*. Ova funkcija ima tri ulazna parametra: trenutni ključ *key* te dva niza od 16 bajtova, *a1* i *a0*. Funkcija započinje inicijalizacijom privremenog niza temp, koji će se koristiti za privremeno pohranjivanje rezultata transformacija. Nakon obavljanja svih prethodno objašnjenih ključnih operacija, funkcija kopira sadržaj niza *a1* u *a0*, a sadržaj temp u *a1*, čime završava proces transformacije.

```
167  void uint128_to_bytes(const uint128_t& value, uint8_t* bytes) {
168      for (int i = 0; i < 8; i++) {
169          bytes[i] = (value[0] >> (56 - 8 * i)) & 0xFF;
170          bytes[i + 8] = (value[1] >> (56 - 8 * i)) & 0xFF;
171      }
172  }
173
174  void bytes_to_uint128(const uint8_t* bytes, uint128_t& value) {
175      value[0] = 0;
176      value[1] = 0;
177      for (int i = 0; i < 8; i++) {
178          value[0] |= static_cast<uint64_t>(bytes[i]) << (56 - 8 * i);
179          value[1] |= static_cast<uint64_t>(bytes[i + 8]) << (56 - 8 * i);
180      }
181  }
182
183  void int_to_vec128(uint32_t i, uint8_t* vec) {
184      memset(vec, 0, 16);
185      vec[15] = (i >> 0) & 0xFF;
186      vec[14] = (i >> 8) & 0xFF;
187      vec[13] = (i >> 16) & 0xFF;
188      vec[12] = (i >> 24) & 0xFF;
189  }
```

Slika 9 - Funkcije za pretvorbu podataka, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

9. Funkcije za pretvorbu podataka

Funkcije *uint128_to_bytes*, *bytes_to_uint128* i *int_to_vec128* služe za pretvorbu podataka između različitih formata koji se koriste u algoritmu Kuznyechik. Ove funkcije omogućuju konverziju između 128-bitnih cjelobrojnih vrijednosti, nizova bajtova i 128-bitnih vektora. Pretvorba podataka u odgovarajuće formate ključna je za ispravnu obradu podataka tijekom generiranja ključeva, enkripcije i dekripcije.

Prva funkcija, *uint128_to_bytes*, uzima 128-bitnu vrijednost koja je pohranjena kao niz od dva 64-bitna broja i pretvara je u niz od 16 bajtova. Funkcija koristi petlju kako bi iterirala kroz prvih osam bajtova i izračunala njihove vrijednosti pomoću pomaka bitova (bit-shifting) i logičke operacije AND s maskom *0xFF* (maska 0xFF predstavlja binarnu vrijednost 11111111, koja služi za izdvajanje samo donjih 8 bita određene vrijednosti, odnosno vrijednosti jednog bajta). Ova operacija osigurava da se svaki segment 64-bitne vrijednosti pravilno razbije na pojedinačne bajtove. Bajtovi se zatim pohranjuju u izlazni niz *bytes*. Funkcija koristi isti postupak za drugi 64-bitni segment 128-bitne vrijednosti, čime se kompletira pretvorba svih 128 bitova u bajtove.

Druga funkcija, *bytes_to_uint128*, obavlja inverznu operaciju u odnosu na prethodnu funkciju. Uzima niz od 16 bajtova i pretvara ga u 128-bitnu vrijednost pohranjenu kao dva

64-bitna broja. Na početku funkcije, obje komponente 128-bitne vrijednosti (*value[0]* i *value[1]*) inicijaliziraju se na nulu. Petlja zatim iterira kroz prvih osam bajtova i izračunava njihov doprinos 64-bitnoj vrijednosti pomoću pomaka bitova ulijevo (left-shifting) i logičke operacije OR. Na taj način, svaki bajt se postupno dodaje u odgovarajući segment 128-bitne vrijednosti. Funkcija ponavlja isti postupak za drugih osam bajtova kako bi se završila pretvorba.

Treća funkcija, *int_to_vec128*, pretvara 32-bitnu cijelu vrijednost u 128-bitni vektor koji se koristi unutar algoritma. Funkcija prvo koristi *memset* kako bi inicijalizirala svih 16 bajtova vektora na nulu. Zatim koristi niz operacija pomaka bitova i logičkih operacija AND kako bi 32-bitnu vrijednost podijelila na četiri bajta i smjestila ih u odgovarajuće pozicije unutar vektora. Najmanje značajni bajt pohranjuje se na posljednju poziciju u vektoru (*vec[15]*), dok se najznačajniji bajt pohranjuje na poziciju *vec[12]*. Ova funkcija je važna jer omogućuje generiranje konstanti koje se koriste tijekom generiranja rundnih ključeva.

```
191  void generateRoundKeys(const uint256_t& masterKey,
192                         array<uint128_t, 10>& roundKeys)
193  {
194      roundKeys[0] = { masterKey[0], masterKey[1] };
195      roundKeys[1] = { masterKey[2], masterKey[3] };
196
197      vector<array<uint8_t, 16>> constants(32);
198      for (int i = 0; i < 32; i++) {
199          uint8_t vec[16] = { 0 };
200          int_to_vec128(i + 1, vec);
201          L(vec);
202          copy(vec, vec + 16, constants[i].begin());
203      }
204
205      for (int i = 1; i <= 4; i++) {
206
207          uint8_t a1[16], a0[16];
208          uint128_to_bytes(roundKeys[2 * i - 2], a1);
209          uint128_to_bytes(roundKeys[2 * i - 1], a0);
210
211          for (int j = 0; j < 8; j++) {
212              uint8_t k[16];
213              copy(constants[8 * (i - 1) + j].begin(),
214                   constants[8 * (i - 1) + j].end(),
215                   k);
216
217              F(k, a1, a0);
218          }
219
220          bytes_to_uint128(a1, roundKeys[2 * i]);
221          bytes_to_uint128(a0, roundKeys[2 * i + 1]);
222      }
223  }
```

Slika 10 - Funkcija za generiranje ključeva, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

10. Funkcija za generiranje ključeva

Funkcija *generateRoundKeys* jedna je od najvažnijih funkcija unutar algoritma Kuznyechik jer ima ključnu ulogu u procesu generiranja rundnih ključeva koji se koriste tijekom svakog kruga šifriranja. Ova funkcija uzima 256-bitni master ključ kao ulaz i generira niz od deset 128-bitnih rundnih ključeva koji se koriste u procesu enkripcije i dekripcije.

Na početku funkcije, prvi i drugi rundni ključ jednostavno se kopiraju iz master ključa. Master ključ je podijeljen na četiri 64-bitna segmenta, a prvi rundni ključ postavljen je na prva dva segmenta master ključa, dok je drugi rundni ključ postavljen na preostala dva segmenta. Ove početne postavke ključne su za početak procesa generiranja dodatnih rundnih ključeva.

Sljedeći korak funkcije uključuje stvaranje vektora konstanti koje se koriste tijekom generiranja novih rundnih ključeva. Ovaj vektor konstanti sadrži 32 elementa, pri čemu je svaki element niz od 16 bajtova. Petlja koja slijedi iterira kroz svih 32 konstante i generira ih pomoću funkcije *int_to_vec128*, koja pretvara cijeli broj u 128-bitni vektor. Svaki generirani vektor zatim prolazi kroz funkciju *L*, koja provodi linearnu transformaciju. Nakon primjene linearne transformacije, vektor se kopira u odgovarajuću poziciju unutar vektora konstanti.

Nakon što su sve konstante generirane, funkcija prelazi na stvaranje preostalih rundnih ključeva. Ovaj dio funkcije koristi dvije ugniježđene petlje kako bi postupno generirao preostale rundne ključeve u četiri kruga. U svakom krugu uzimaju se prethodna dva rundna ključa i pretvaraju u nizove bajtova pomoću funkcije *uint128_to_bytes*. Nakon što su svi potrebni izračuni završeni putem *F* funkcije, novi rundni ključevi pretvaraju se natrag u 128-bitne vrijednosti pomoću funkcije *bytes_to_uint128* i pohranjuju u niz rundnih ključeva.

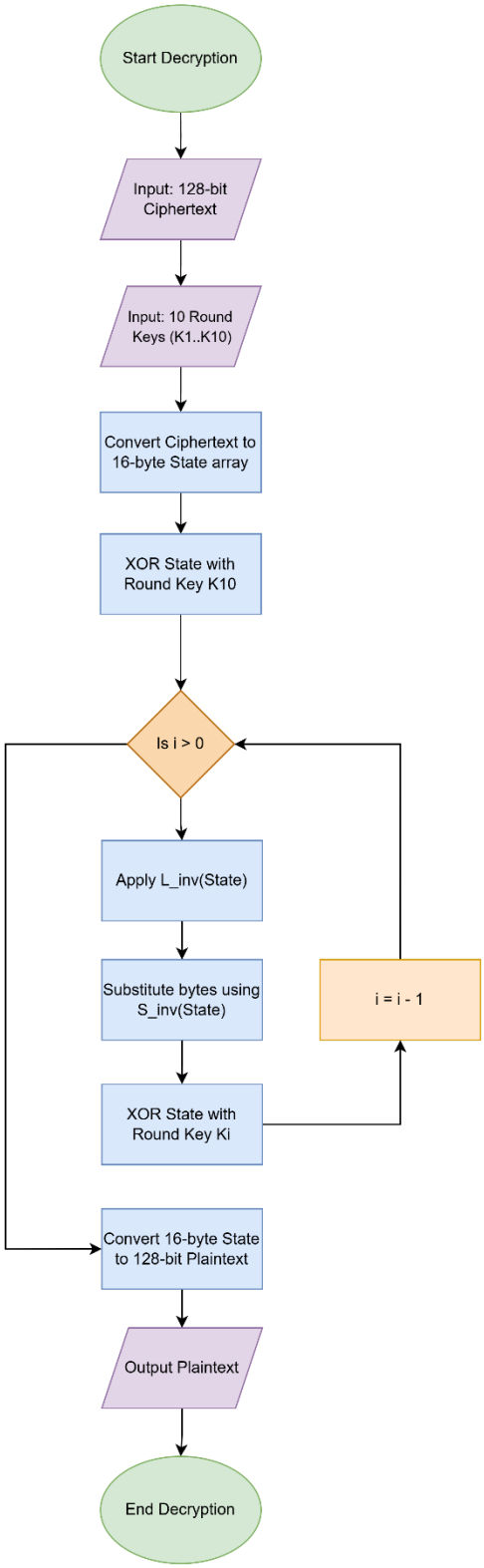
```
uint128_t encryptBlock(const uint128_t& plaintext, const array<uint128_t, 10>& roundKeys) {
    uint8_t state[16];

    for (int i = 0; i < 8; i++) {
        state[i] = (plaintext[0] >> (56 - 8 * i)) & 0xFF;
        state[i + 8] = (plaintext[1] >> (56 - 8 * i)) & 0xFF;
    }

    uint8_t key[16];


    for (int i = 0; i < 9; i++) {

        for (int j = 0; j < 8; j++) {
            key[j] = (roundKeys[i][0] >> (56 - 8 * j)) & 0xFF;
            key[j + 8] = (roundKeys[i][1] >> (56 - 8 * j)) & 0xFF;
        }
        X(state, key);

        S(state);

        L(state);
    }

    for (int j = 0; j < 8; j++) {
        key[j] = (roundKeys[9][0] >> (56 - 8 * j)) & 0xFF;
        key[j + 8] = (roundKeys[9][1] >> (56 - 8 * j)) & 0xFF;
    }
    X(state, key);

    uint128_t ciphertext;
    ciphertext[0] = 0;
    ciphertext[1] = 0;
    for (int i = 0; i < 8; i++) {
        ciphertext[0] |= static_cast<uint64_t>(state[i]) << (56 - 8 * i);
        ciphertext[1] |= static_cast<uint64_t>(state[i + 8]) << (56 - 8 * i);
    }
    return ciphertext;
}
```

Slika 11 - Funkcija za enkripciju, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

11. Funkcija za enkripciju

Funkcija *encryptBlock* implementira osnovni proces enkripcije jednog 128-bitnog bloka podataka. Ova funkcija prima kao ulaz čisti tekst (*plaintext*), koji je 128-bitna vrijednost, i niz rundnih ključeva (*roundKeys*), koji je prethodno generiran pomoću funkcije generateRoundKeys. Cilj ove funkcije je pretvoriti ulazni blok podataka u šifrirani tekst, poznat kao *ciphertext*, primjenom serije kriptografskih transformacija.

Dijagram toka za dekripciju



Slika 12 - Dijagram toka za dekripciju (Samostalna izrada u alatu draw.io, 2025.)

Programski kod za dekripciju

```
111 void S_inv(uint8_t* state) {
112     for (int i = 0; i < 16; i++) {
113         state[i] = Pi_inv[state[i]];
114     }
115 }
116
117 void L_inv(uint8_t* state) {
118     for (int i = 0; i < 16; i++) {
119         R_inv(state);
120     }
121 }
```

Slika 13 - Inverzne funkcije S-Transformacije i L-Transformacije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

1. Inverzne funkcije S-Transformacije i L-Transformacije

Funkcija *S_inv* provodi inverznu zamjenu bajtova koristeći inverznu S-box tablicu (*Pi_inv*). Bez ove inverzne operacije, kao i ostalih operacija za dekripciju, šifrirani tekst bi ostao u svom iskrivljenom obliku, jer bi podaci bili nepovratno izmiješani nelinearnim zamjenama.

Funkcija *L_inv* provodi inverznu linearnu transformaciju nad stanjem podataka. Funkcija *L_inv* poništava ovaj efekt primjenom inverzne permutacije, koja se provodi kroz 16 uzastopnih poziva funkcije *R_inv*. Svaki poziv funkcije *R_inv* pomiče stanje unatrag za jedan korak, postupno vraćajući podatke u njihovo početno stanje prije primjene linearne transformacije.

```
123 void R_inv(uint8_t* state) {
124     static const uint8_t c[16] = {
125         0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01, 0xfb,
126         0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01
127     };
128
129     uint8_t b0 = state[0];
130
131     memmove(state, state + 1, 15);
132
133     uint8_t partial = 0;
134     for (int i = 0; i < 15; i++) {
135         partial ^= gf_mul(state[i], c[i]);
136     }
137
138     state[15] = b0 ^ partial;
139 }
```

Slika 14 - Inverzna funkcija R-Transformacije, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

2. Inverzna funkcija R-Transformacije

Funkcija *R_inv* poništava difuziju, pomiče bajtove udesno i vraća originalnu vrijednost posljednjeg bajta bloka, čime obrće transformaciju uvedenu funkcijom *R*. Prvi korak funkcije uključuje definiranje statičkog niza konstantnih vrijednosti *c*, koji sadrži 16 unaprijed definiranih bajtova. Nakon definiranja konstanti, funkcija sprema trenutnu vrijednost prvog bajta stanja u varijablu *b0*. Ovaj bajt će kasnije biti potreban za izračun nove vrijednosti posljednjeg bajta bloka. Sljedeći korak je pomicanje svih bajtova u stanju udesno za jednu poziciju. To se postiže pomoću funkcije *memmove*, koja kopira 15 preostalih bajtova na njihovu novu poziciju unutar niza. Na ovaj način, bajtovi se pomiču prema kraju bloka, ostavljajući prvu poziciju praznom.

Nakon pomicanja bajtova, funkcija započinje izračunavanje nove vrijednosti posljednjeg bajta bloka. Varijabla *partial* inicijalizira se na nulu, a zatim funkcija prolazi kroz prvih 15 bajtova u stanju. Svaki od ovih bajtova množi se odgovarajućom konstantom iz niza *c* koristeći funkciju *gf_mul*, koja provodi množenje unutar Galoisovog polja. Rezultat svakog množenja akumulira se u varijabli *partial* pomoću XOR operacije.

Na kraju, funkcija postavlja novu vrijednost posljednjeg bajta bloka. Ova vrijednost dobiva se pomoću XOR operacije između spremljenog bajta *b0* i akumulirane vrijednosti *partial*. Ova operacija osigurava da posljednji bajt reflektira sve promjene koje su se dogodile u preostalim bajtovima tijekom pomicanja i množenja.

```
271  uint128_t decryptBlock(const uint128_t& ciphertext, const array<uint128_t, 10>& roundKeys) {
272      uint8_t state[16];
273      uint128_to_bytes(ciphertext, state);
274
275      {
276          uint8_t key[16];
277          uint128_to_bytes(roundKeys[9], key);
278          X(state, key);
279      }
280
281      for (int i = 9; i > 0; i--) {
282          L_inv(state);
283          S_inv(state);
284
285          uint8_t key[16];
286          uint128_to_bytes(roundKeys[i - 1], key);
287          X(state, key);
288      }
289
290      uint128_t plaintext{0, 0};
291      bytes_to_uint128(state, plaintext);
292      return plaintext;
293  }
```

Slika 15 - Funkcija za dekripciju, isječak programskog koda (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

3. Funkcija za dekripciju

Cilj ove funkcije je pretvoriti šifrirani tekst (*ciphertext*) natrag u izvorni čisti tekst (plaintext) primjenom inverznih transformacija koje poništavaju učinke enkripcije. Funkcija koristi niz rundnih ključeva (*roundKeys*) koji su prethodno generirani i koriste se obrnutim redoslijedom kako bi se vratili izvorni podaci.

5. Analiza i testiranje

Testni slučajevi za enkripciju

Testni input uzet je iz službene dokumentacije za GOST R 34.12-2015, te je output moga algoritma uspoređen s outputom iz iste (Dolmatov, 2016, Section 5). Za testiranje ispravnosti implementacije funkcije za enkripciju korišten je početni ključ i 128-bitni čisti tekst (plaintext). U ovom primjeru uzet je ulazni čisti tekst *1122334455667700ffeeddccbbaa9988*.

Nakon što su primijenjene sve transformacije u devet rundi, rezultat enkripcije bio je šifrirani tekst *7f679d90bebc24305a468d42b9d4edcd*. Ovaj izlaz pokazuje ispravnost implementacije algoritma i njegovu otpornost na povratno dekodiranje bez poznavanja ključeva.

```
Odaberite opciju:
1. Testiraj enkripciju
2. Testiraj dekripciju
Unesite svoj odabir: 1
Plaintext:  1122334455667700ffeeddccbbaa9988
Ciphertext: 7f679d90bebc24305a468d42b9d4edcd
```

Slika 16 – Testni slučaj enkripcije, ispis u VS Code Ubuntu terminalu (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

Testni slučajevi za dekripciju

Za provjeru ispravnosti funkcije za dekripciju korišten je šifrirani tekst generiran tijekom enkripcijskog testa, zajedno s istim nizom rundnih ključeva. Dakle, zapravo je za provjeru (input) dekripcije korišten output enkripcije, što ima smisla s obzirom da su funkcije inverzne. Proces dekripcije vratio je izvorni čisti tekst, potvrđujući reverzibilnost algoritma.

Šifrirani tekst *7f679d90bebc24305a468d42b9d4edcd* prošao je kroz deset uzastopnih inverznih transformacija. Nakon završetka svih dekripcijskih rundi, dobiveni rezultat bio je izvorni čisti tekst *1122334455667700ffeeddccbbaa9988*, što dokazuje točnost implementacije dekripcije.

```
Odaberite opciju:
1. Testiraj enkripciju
2. Testiraj dekripciju
Unesite svoj odabir: 2
Ciphertext: 7f679d90bebc24305a468d42b9d4edcd
Decrypted plaintext: 1122334455667700ffeeddccbbaa9988
```

Slika 17 - Testni slučaj dekripcije, ispis u VS Code Ubuntu terminalu (Samostalna izrada u VS Code - WSL: Ubuntu, C++)

Vremenska složenost algoritma

1. *gf_mul(a, b)*
    - Vrti se petlja od 8 koraka, u svakoj se obavlja nekoliko bitnih operacija (pomaci, XOR) i provjerava je li bit postavljen. Sve to je fiksni broj operacija.
    - Složenost: $O(1)$.
2. *R(state)*
    - Uključuje 16 poziva funkcije *gf_mul* (za svaki od 16 bajtova). Nakon toga pomiče polje za jedan bajt i postavlja izračunatu vrijednost na početak. Broj poziva i veličina polja su fiksni.
    - Složenost: $O(1)$.
3. *R_inv(state)*
    - Inverzna operacija od R, ali i dalje s 16 poziva *gf_mul* i sličnim pomakom polja od 16 bajtova. Sve je opet fiksni broj koraka.
    - Složenost: $O(1)$.
4. *L(state)*
    - Petlja od 16 iteracija, u svakoj se poziva *R*.
    - Svaka iteracija ima fiksni broj poziva *gf_mul*. Ukupno i dalje fiksna količina posla.
    - Složenost: $O(1)$).
5. *L_inv(state)*
    - Analogno *L*, samo poziva *R_inv* 16 puta. Broj operacija ostaje fiksan.
    - Složenost: $O(1)$).
6. *S(state)*
    - Prolazi se kroz 16 bajtova i za svaki bajt radi tablično preslikavanje (lookup) u *Pi*. Petlja se uvijek “vrti” 16 puta.
    - Složenost: $O(1)$.

7. *S_inv(state)*

- Isto kao *S*, samo koristi inverznu tablicu (*Pi_inv*). Također 16 preslikavanja, dakle fiksno.
- Složenost: $O(1)$.

8. *X(state, key)*

- XOR nad 16 bajtova stanja i 16 bajtova ključa. Petlja s 16 koraka, fiksno.
- Složenost: $O(1)$.

9. *F(key, a1, a0)*

- XOR nad 16 bajtova (*a1* i *key*), zatim *S*, zatim *L*, te još jedan XOR (s *a0*). Na kraju se 16 bajtova *a1* kopira u *a0*, a dobiveni temp kopira u *a1*. Sve je na razini 16-bajtnih polja, s fiksnim brojem operacija.
- Složenost: $O(1)$.

10. *generateRoundKeys(masterKey, roundKeys)*

- Inicijalizira se 32 konstante, svaka obradi 16 bajtova i poziva *L*. Zatim se 4 puta (po 8 poziva *F*) dobiju ostali rundski ključevi. Sve radi s fiksnim duljinama (16 bajtova) i fiksnim brojem petlji.
- Složenost: $O(1)$.

11. *encryptBlock(plaintext, roundKeys)*

- Pretvaranje 128-bitnog plaintexta u state polje (16 bajtova). 9 rundi, svaka radi: *X*(state, ključ), *S*(state), *L*(state). Nakon toga finalni XOR sa zadnjim ključem. Broj rundi (9) i operacije unutar njih su fiksne (izračunato iznad).
- Složenost: $O(1)$.

12. *decryptBlock(ciphertext, roundKeys)*

- Početni XOR sa zadnjim rundskim ključem, zatim 9 obrnutih rundi (*L_inv*, *S_inv*, *X*). Kao i kod enkripcije, sve je vezano uz 16 bajtova i 9 rundi.
- Složenost: $O(1)$.

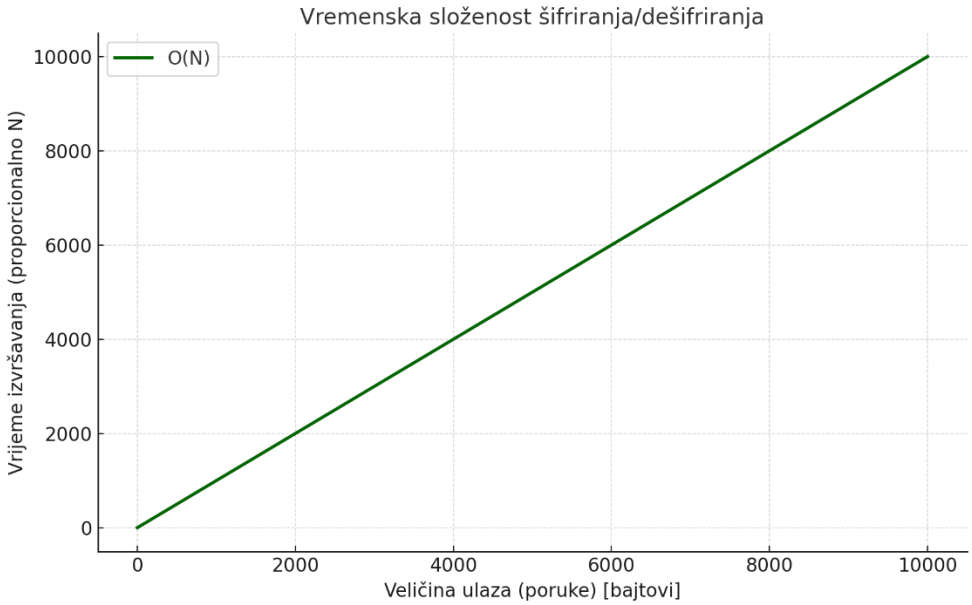
Svaka navedena funkcija radi nad fiksnom veličinom podataka (128-bitni blok, odnosno 16 bajtova) i ima unaprijed određen broj koraka/rundi, što sveukupno dovodi do konstantne vremenske složenosti $O(1)$ za svaku funkciju.

S obzirom na to da algoritam ima linearnu vremensku složenost $O(1)$ u odnosu na veličinu ulazne poruke, nije moguće razlikovati best i worst case scenarij u smislu Big-O notacije. Razlog je što algoritam uvijek obrađuje blok fiksne veličine na isti način, bez obzira na njezin sadržaj, te stoga svi ulazi imaju identičnu složenost.

Međutim, ako se algoritam koristi za šifriranje cijele poruke, potrebno je uzeti u obzir dodatni korak pripreme podataka. Budući da se radi o blok šifri, ulazna poruka mora se podijeliti na blokove od 128 bita (16 bajtova). Ako duljina poruke nije višekratnik veličine bloka, potrebno je primijeniti tehniku ispunjavanja (padding), odnosno dodavanje podataka kako bi se zadnji blok poravnao.

U slučaju poruke duljine N bajtova, broj potrebnih blokova za šifriranje je $\lceil N/16 \rceil$. Svaki blok tada prolazi kroz proces šifriranja, što znači da se funkcije poput encryptBlock pozivaju za svaki pojedini blok. Ukupna vremenska složenost šifriranja cijele poruke, kada se uključuje i ovaj korak raspodjele po blokovima, postaje proporcionalna broju blokova, tj. $\mathrm{O}(N/16)$, što je linearna složenost $\mathrm{O}(N)$ u odnosu na veličinu poruke N.

Dakle, iako su pojedine funkcije algoritma konstantne složenosti, složenost primjene cijelog algoritma na poruku ovisi o duljini poruke. Kada se algoritam primjenjuje na poruke proizvoljne duljine, ukupna vremenska složenost postaje linearna u odnosu na veličinu ulazne poruke, tj. $\mathrm{O}(N)$.



Slika 18 - Graf vremenske složenosti (vrijeme izvršavanja u odnosu na veličinu ulaza), (Samostalna izrada, Python3 - matplotlib)

6. Praktična primjena Kuznyechik algoritma u stvarnom svijetu

Kuznyechik blok šifra nalazi svoju praktičnu primjenu u širokom spektru područja gdje su sigurnost podataka i otpornost na napade od ključne važnosti. Kao dio ruskog nacionalnog standarda za šifriranje (GOST R 34.12-2015), Kuznyechik se koristi u financijskim sustavima, osiguravajući povjerljivost i integritet transakcija. Njegova implementacija u bankarskim sustavima omogućuje sigurno šifriranje osjetljivih podataka, poput podataka o korisnicima i financijskih transakcija, čime se sprječavaju napadi poput presretanja podataka ili manipulacije sadržajem.

U području državne sigurnosti, Kuznyechik se koristi za šifriranje komunikacija unutar državnih agencija i vojnih sustava, osiguravajući da ključne informacije ostanu nedostupne neovlaštenim osobama.

Kuznyechik se također sve više koristi u ugrađenim sustavima (embedded systems), poput uređaja za Internet stvari (IoT), gdje je ključna optimizacija resursa uz zadržavanje visoke razine sigurnosti. Na primjer, u pametnim mrežama (smart grids) koristi se za zaštitu podataka između senzora i centralnih sustava, čime se sprječavaju napadi na

kritičnu infrastrukturu. Njegovdiz ajn omogućuje prilagodbu specifičnim hardverskim platformama, čime se povećava učinkovitost i smanjuje potrošnja energije, što je ključno za resursno ograničene uređaje.

Dodatno, standardizacija kroz RFC 7801 otvara vrata za integraciju Kuznyechik algoritma u međunarodne sigurnosne protokole, poput VPN-a i TLS-a, omogućujući njegovu primjenu u globalnim sigurnosnim sustavima. Ova interoperabilnost pruža dodatnu fleksibilnost organizacijama koje žele uskladiti lokalne standarde s globalnim praksama, dok istovremeno osiguravaju neovisnost od zapadnih sigurnosnih standarda poput AES-a.

7. Zaključak

Kuznyechik blok šifra, kao dio GOST R 34.12-2015 standarda, ističe se kao vrhunski primjer modernog kriptografskog dizajna koji spaja sigurnost, performanse i praktičnost implementacije. Njena struktura, utemeljena na SPN mreži i transformacijama poput S-Boxa, R-transformacije i L-transformacije, omogućuje otpornost na napredne oblike kriptoanalize i osigurava široku primjenjivost u različitim sigurnosnim sustavima. Duljina ključa od 256 bita i blokovi od 128 bita čine Kuznyechik izuzetno otpornim na brute-force napade i prilagođenim za rad u okruženjima s visokim sigurnosnim zahtjevima.

Kroz praktičnu implementaciju algoritma u jeziku C++, ovaj rad pokazuje kako se matematički principi konačnih polja i modularnih transformacija mogu pretočiti u učinkovito softversko rješenje. Analizom vremenske složenosti algoritma, kao i testiranjem funkcija za enkripciju i dekripciju, potvrđena je njegova tehnička ispravnost i prikladnost za stvarne kriptografske izazove

.

Literatura

1. Dolmatov, V. (Editor). (2016). GOST R 34.12-2015: Block Cipher "Kuznyechik". RFC 7801. Retrieved from https://www.rfc-editor.org/info/rfc7801
2. ISO/IEC. (2006). Information technology — Security techniques — Modes of operation for an n-bit block cipher, ISO/IEC 10116:2006.
3. ISO/IEC. (2015). Information technology — Security techniques — Encryption algorithms — Part 1: General, ISO/IEC 18033-1:2015.

Popis slika