

RTU.jpg

---

**МИНОБРНАУКИ РОССИИ**

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

**Институт Кибернетики**

**Курсовая работа**

по дисциплине

"Методы программирования"

Тема курсовой работы

**Реализация шифратора на основе алгоритма ГОСТ**

**28147-89 в режиме CBC**

**Выполнил:**

студент группы ККСО-04-19

Курбатов В.М.

**Научный руководитель:**

Кирюхин Виталий Александрович

Москва — 2021

# Оглавление

# Введение

В данной курсовой работе представлена реализация ГОСТ 28147-89 (являющийся примером DES-подобных криптосистем, созданных по классической итерационной схеме Фейстеля) - стандарт симметричного шифрования в режиме CBC(Cipher Block Chaining) - режим сцепления блоков шифротекста, на языке программирования C++. Алгоритм шифрования данных представляет собой 64-битовый блочный алгоритм с 256-битовым ключом.

Согласно извещению ФСБ о порядке использования алгоритма блочного шифрования ГОСТ 28147-89 данный алгоритм базового блочного шифрования применяется в криптографических методах обработки и защиты информации, не содержащей сведений, составляющих государственную тайну.

## 0.1 Теоретическая часть

Клод Шеннон в ряде своих основополагающих работ по теории шифрования сформулировал условия стойкости современного блочного шифра. Такой шифр должен обладать свойствами перемешивания и рассеивания:

- **рассеивание** - это свойство шифра, при котором один символ (бит) исходного текста влияет на несколько символов (битов) шифротекста, оптимально - на все символы в пределах одного блока. Если данное условие выполняется, то при шифровании двух блоков данных с минимальными отличиями между ними должны получаться совершенно непохожие друг на друга блоки шифротекста. Точно такая же картина должна иметь место и для зависимости шифротекста от ключа: один символ (бит) ключа должен влиять на несколько символов (битов) шифротекста.
- **перемешивание** - это свойство шифра скрывать зависимость между символами исходного текста и шифротекста. Если шифр достаточно хорошо "перемешивает" биты исходного текста, то соответствующий шифротекст не содержит никаких статистических и тем более функциональных закономерностей для стороннего наблюдателя, обладающего лишь ограниченными вычислительными ресурсами.

### 0.1.1 Общие сведения о блочных шифрах

Характерной особенностью блочных криптоалгоритмов является тот факт, что в ходе своей работы они осуществляют преобразование блока входной информации фиксированной длины и получают результирующий блок той же длины, но недоступный для прочтения сторонним лицам, не владеющим

КЛЮЧОМ.

Таким образом, схему работы блочного шифра можно описать функциями

$$Z = EnCrypt(X, Key)$$

и

$$X = DeCrypt(Z, Key)$$

Ключ  $Key$  является параметром блочного криптоалгоритма и представляет собой некоторый блок двоичной информации фиксированного размера. Исходный  $X$  и зашифрованный  $Z$  блоки данных также имеют фиксированную разрядность, равную между собой, но необязательно равную длине ключа.

ГОСТ 28147-89 признан стойким алгоритмом. Криптоалгоритм именуется идеально стойким, если зашифрованный блок данных можно прочесть только перебрав все возможные ключи, пока сообщение не окажется осмысленным.

Поскольку по теории вероятности искомый ключ будет найден с вероятностью  $\frac{1}{2}$  после перебора половины всех ключей, постольку на взлом идеально стойкого криптоалгоритма с ключом длиной  $N$  потребуется в среднем  $2^{N-1}$  проверок. Таким образом, в общем случае стойкость блочного шифра зависит только от длины ключа и возрастает экспоненциально с ее ростом.

Характерным признаком блочных алгоритмов является много кратное и косвенное использование материала ключа. Это диктуется, в первую очередь, требованием невозможности обратного декодирования в отношении ключа при известных исходном и зашифрованном текстах.

## 0.1.2 Сеть Фейстеля

Приведем описание сети Фейстеля, опираясь на [9]. Сеть Фейстеля получила широкое распространение, поскольку обеспечивает выполнение требования о многократном использовании материала ключа и исходного блока информации.

Классическая сеть Фейстеля имеет структуру, представленную на рис.1

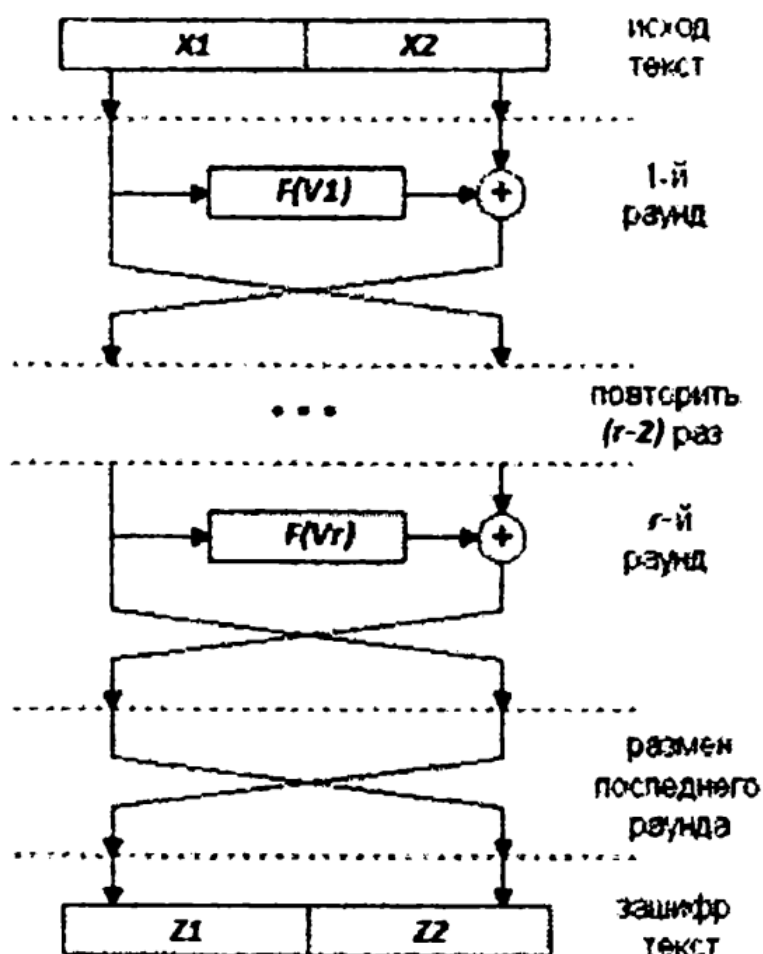


Рис. 1: Классическая сеть Фейстеля

Независимые потоки информации, появляющиеся из исходного блока, называются ветвями сети. В классической схеме их две.

Величины  $V_i$ , именуются параметрами сети, обычно это функции от материала ключа. Функция  $F$  является образующей. Действие, состоящее из однократного вычисления образующей функции и последующего наложения ее результата на другую ветвь с обменом их местами, называется циклом или раундом (англ. round) сети Фейстеля.

Оптимальное число раундов  $K = 8 - 32$ . Важно то, что увеличение количества раундов значительно увеличивает крипто стойкость любого блочного шифра к криптоанализу.

В современных алгоритмах обычно применяют модификацию сети Фейстеля для большего числа ветвей. Это в первую очередь связано с тем, что при больших размерах кодируемых блоков (128 и более битов) становится неудобно работать с математическими функциями по модулю 64 и выше.

Сеть Фейстеля надежно зарекомендовала себя как криптостойкая схема произведения преобразований, и ее можно найти практически в любом современном блочном шифре. Незначительные модификации касаются обычно дополнительных начальных и конечных преобразований (англ. - whitening) шифруемого блока. Подобные преобразования, выполняемые обычно также (исключаящим или) или сложением, имеют целью повысить начальную рандомизацию входного текста.

Таким образом, криптостойкость блочного шифра, использующего сеть Фейстеля, определяется на 95 % функцией  $F$  и правилом вычисления  $V_i$ , из ключа.

### 0.1.3 ГОСТ 28147-89

В описываемом алгоритме блок длиной 64 бита, подлежащий зашифрованию, разделяется на две равные части по 32 бита - правую и левую. Затем выполняется 32 итерации с использованием итерационных ключей, получаемых из исходного 256-битного ключа шифрования.

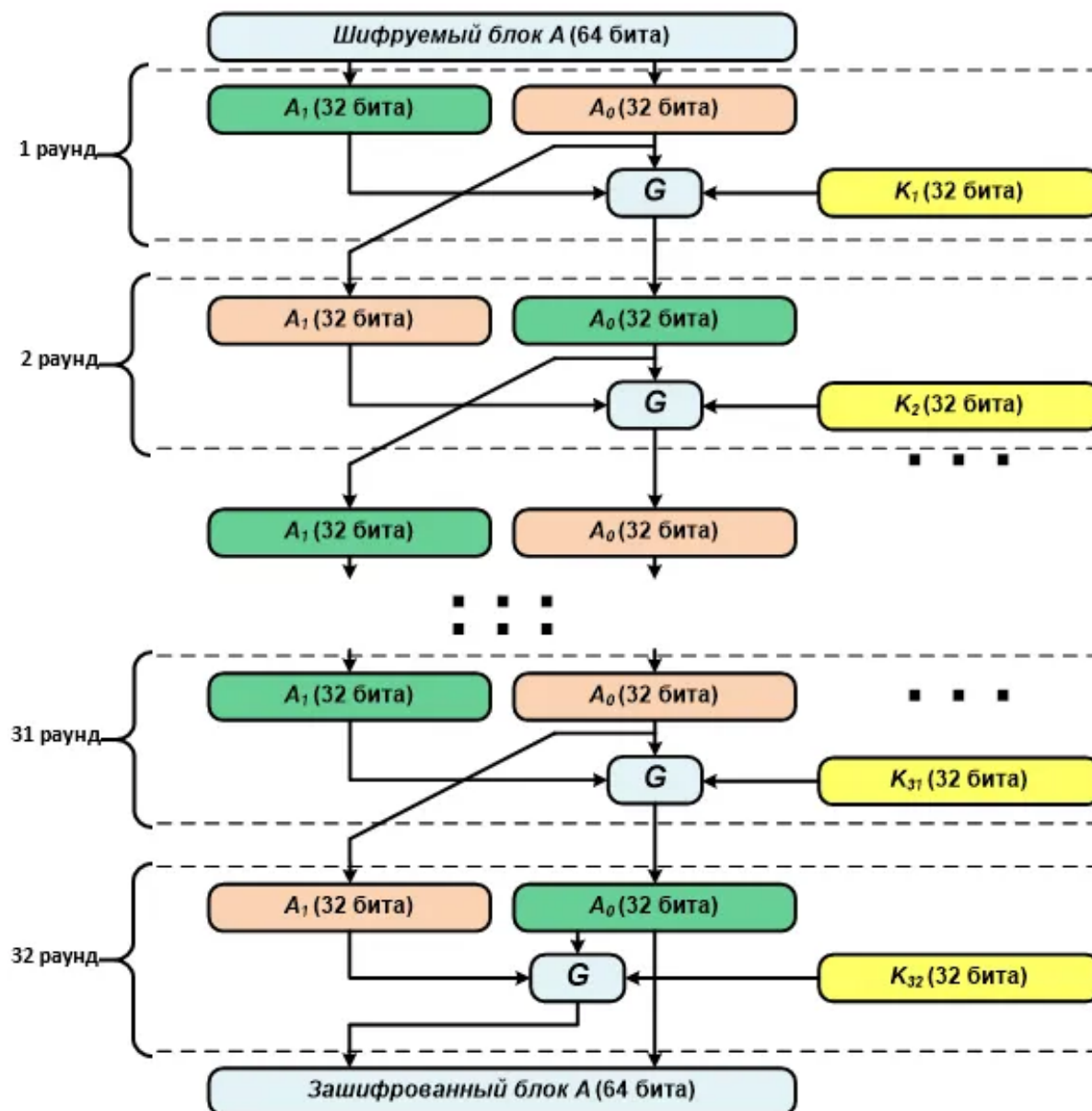


Рис. 2: Схема работы алгоритма при зашифровании



Во время каждой итерации, кроме 32, с правой и левой половиной зашифруемого блока производится одно преобразование, основанное на сети Фейстеля. Сначала правая часть складывается по модулю 32 с текущим итерационным ключом, затем полученное 32-битное число делится на восемь 4-битных и каждое из них с использованием таблицы перестановки преобразуется в другое 4-битное число (нелинейное биективное преобразование).

После этого преобразования полученное число циклически сдвигается влево на одиннадцать разрядов. Далее результат XORится с левой половиной блока. Получившееся 32-битное число записывается в правую половину блока, а старое содержимое правой половины переносится в левую половину блока.

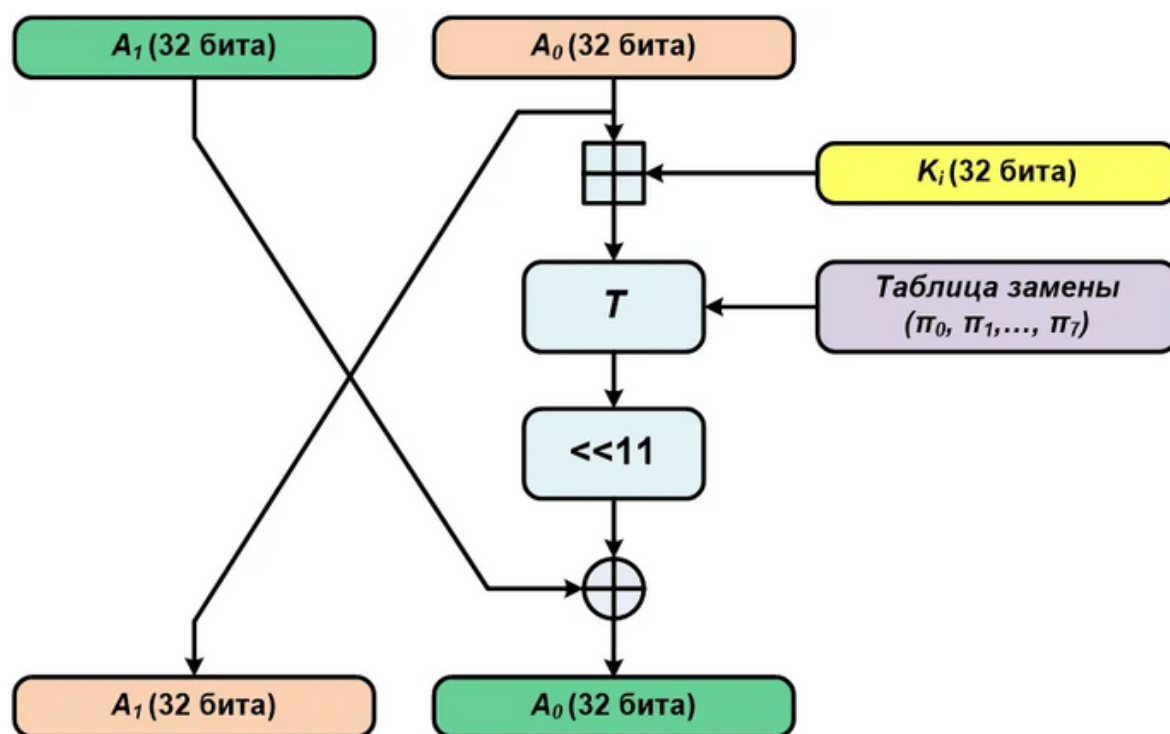


Рис. 3: Схема одной итерации

В ходе последней (тридцать второй) итерации так же, как описано выше, преоб-

разуется правая половина, после чего полученный результат пишется в левую часть исходного блока, а правая половина сохраняет свое значение.

Итерационные ключи получаются из исходного 256-битного ключа. Исходный ключ делится на восемь 32-битных подключей, и далее они используются в следующем порядке: три раза с первого по восьмой и один раз с восьмого по первый.

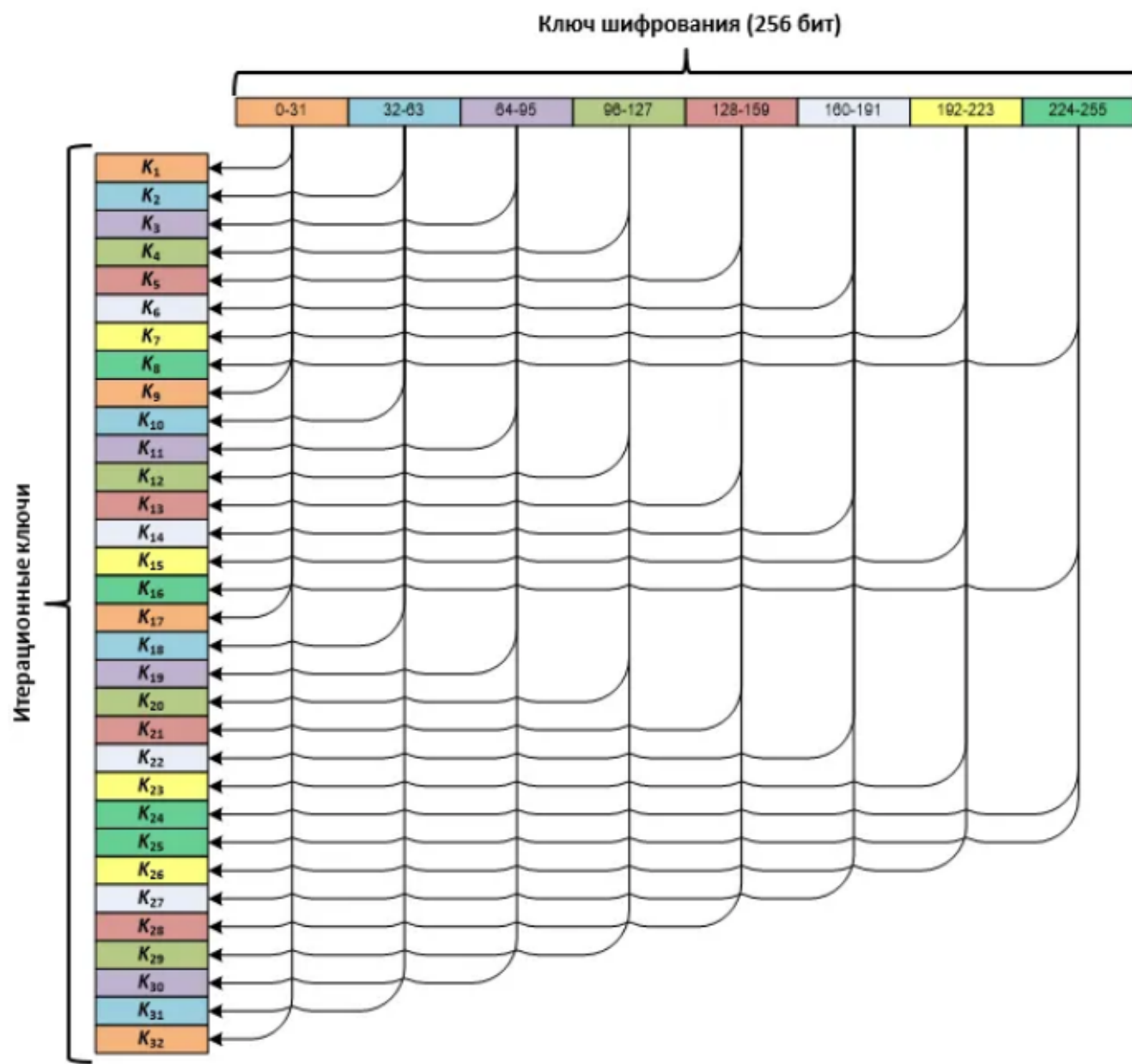


Рис. 4: Схема получения итерационных ключей

Для расшифровывания используется такая же последовательность итераций, как и при зашифровывании, но порядок следования ключей изменяется на обратный.

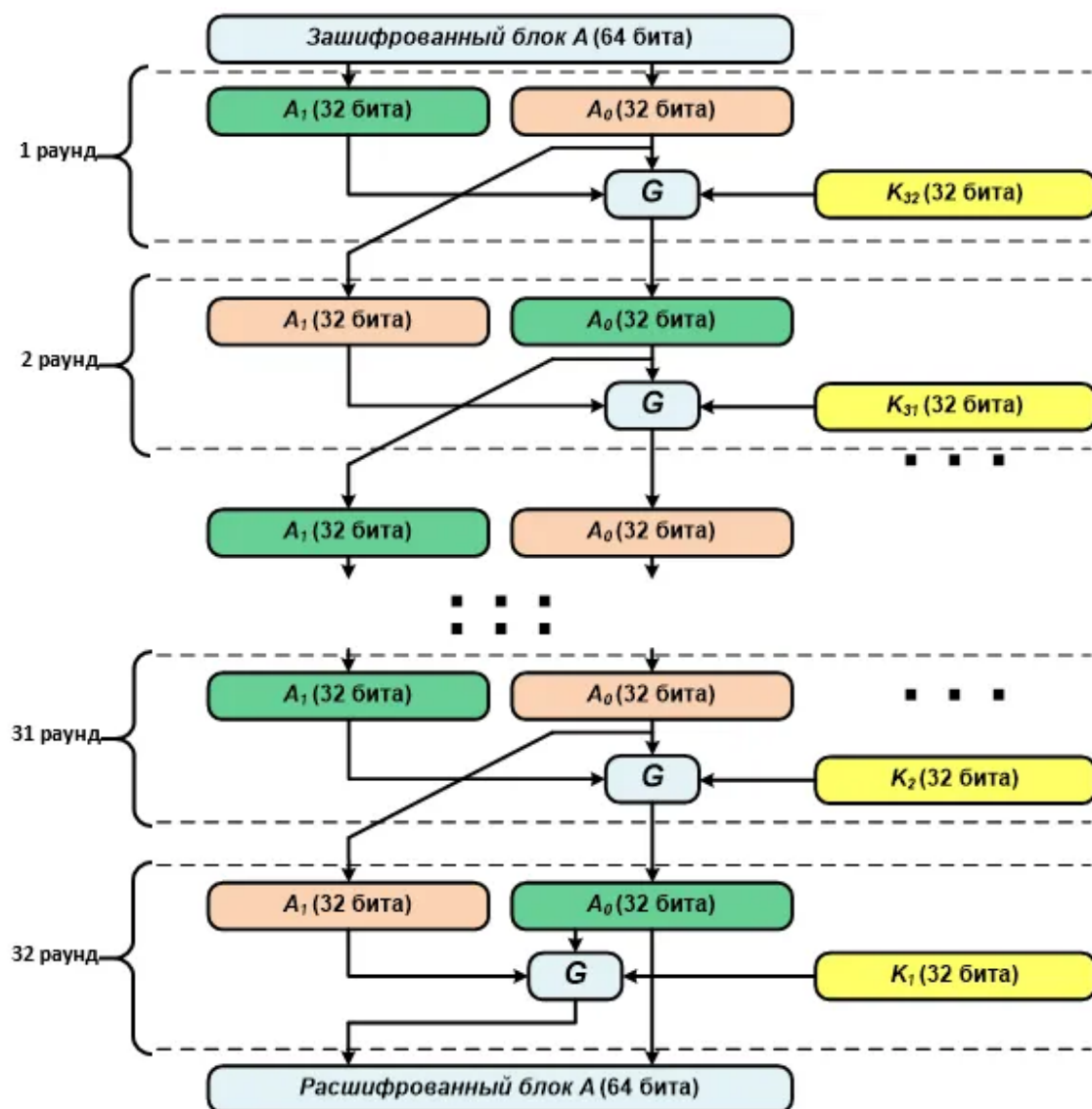


Рис. 5: Схема работы алгоритма при расшифровывании

Для шифровки исходного текста произвольной длины блочные шифры могут быть использованы в нескольких режимах:

- электронной кодировочной книги (ECB - Electronic Code Book);
- сцепления блоков шифрованного текста (CBC - Cipher Block Chaining);
- обратной связи по шифрованному тексту (CFB - Cipher Feedback);
- обратной связи по выходу (OFB - Output Feedback).

В режиме сцепления блоков шифрованного текста (CBC) каждый блок исходного текста складывается поразрядно по модулю 2 с предыдущим блоком шифрованного текста, а затем шифруется. Для начала процесса шифрования используется синхропосылка (или начальный вектор), которая передается в канал связи в открытом виде.

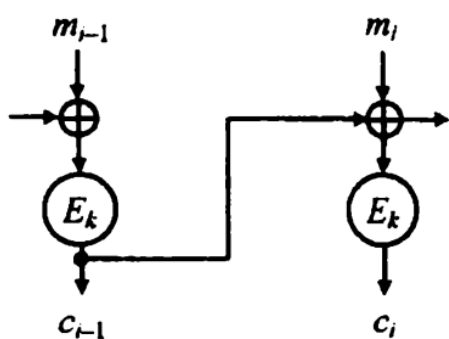


Рис. 6: Режим сцепления блоков шифрованного текста

$$c_i = E_k(m_i \oplus c_{i-1})$$

и

$$m_i = D_k(c_i) \oplus c_{i-1}$$

То есть, схема работы при шифровании будет иметь вид:

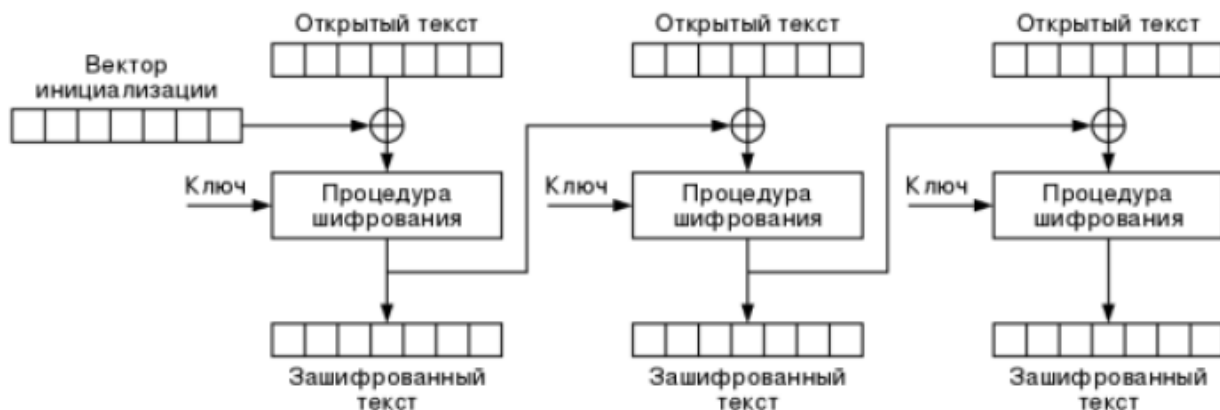


Рис. 7: Схема работы при шифровании в режиме CBC

А при расшифровании будет иметь вид:

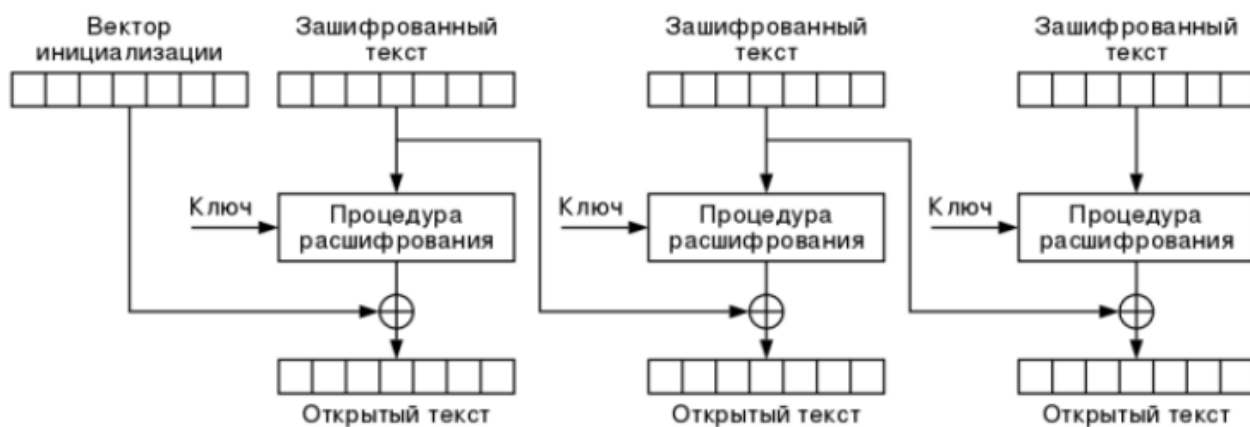


Рис. 8: Схема работы при расшифровании в режиме CBC

Стойкость режима CBC равна стойкости блочного шифра, лежащего в его основе. Кроме того, структура исходного текста скрывается за счет сложения

предыдущего блока шифрованного текста с очередным блоком открытого текста. Стойкость шифрованного текста увеличивается, поскольку становится невозможной прямая манипуляция исходным текстом, кроме как путем удаления блоков из начала или конца шифрованного текста.

К недостаткам CBC стоит отнести возможность определения начала изменения данных по изменению шифротекста (если сравнить шифротексты двух сообщений с одним и тем же ключом, то номер первого блока, в котором шифротексты различаются, будет соответствовать номеру первого блока, в котором различаются исходные сообщения).

К достоинствам CBC стоит отнести:

- постоянная скорость обработки блоков (скорость определяется эффективностью реализации шифра; время выполнения операции «хор» пренебрежимо мало);
- отсутствие статистических особенностей, характерных для режима ECB (поскольку каждый блок открытого текста «смешивается» с блоком шифротекста, полученным на предыдущем шаге шифрования);

#### **0.1.4 Сравнительными характеристики методов шифровки блочных шифров**

Приведу таблицу с сравнительными характеристиками методов шифровки блочных шифров:

Характеристика	ECB	CBC	CFB	OFB
Блоки, от которых зависит шифрование блока	Текущий	Все предыдущие	Все предыдущие	Позиция блока в файле
Результат искажения одного бита при передаче	Порча всего текущего блока	Порча всего текущего и всех последующих блоков	Порча одного бита текущего блока и всех последующих блоков	Порча одного бита текущего блока
Возможность кодирования без дополнения числа байтов, некратных блоку	Нет	Нет	Да	Да
Поступление на выход криптосистемы	Выход криптоалгоритма	Выход криптоалгоритма	XOR-маска с исходным текстом	XOR-маска с исходным текстом

## 0.2 Практическая часть

В тексте стандарта ГОСТ 28147-89 указывается, что поставка заполнения узлов замены (S-блоков) производится в установленном порядке, то есть разработчиком алгоритма.

В своей реализации я использовал узел замены определенные документом RFC 4357. Идентификатор: id-Gost28147-89-CryptoPro-A-ParamSet. OID: 1.2.643.2.2.31.1.

```

10 static unsigned char S box[8][16]=
11 {
12     {9,6,3,2,8,11,1,7,10,4,14,15,12,0,13,5},
13     {3,7,14,9,8,10,15,0,5,2,6,12,11,4,13,1},
14     {14,4,6,2,11,3,13,8,12,15,5,10,0,7,1,9},
15     {14,7,10,12,13,1,3,9,0,2,11,4,15,8,5,6},
16     {11,5,1,9,8,13,15,0,14,4,2,3,12,7,10,6},
17     {3,10,13,12,1,2,0,11,7,5,9,4,8,15,14,6},
18     {1,13,2,9,7,10,6,0,8,12,4,5,15,3,11,14},
19     {11,10,15,5,0,12,14,8,6,2,3,9,1,7,13,4}
20 };
21

```

Рис. 9: Таблица замены

Данный узел замен используется криптопровайдером CryptoPRO CSP по умолчанию. Так же данный узел замен используется в ПО "Верба-О".

Далее опишу используемые мною функции:

*Add\_mod2* - функция, реализующая сложение двух двоичных векторов по модулю 2. Каждый байт первого вектора  $\oplus$  с соответствующим байтом второго вектора, результат операции записываем в третий вектор.

```

211
212 → void Add mod2(uint8_t* A, uint8_t* B, uint8_t* C)
213 {
214     for(int i = 0; i < 8; i++) C[i] = A[i]^B[i];
215 }
216

```

Рис. 10: Функция *Add\_mod2*

*add\_mod32* - функция, реализующая сложение двух двоичных векторов по модулю 32. Два исходных 4-байтовых вектора представляются как два 32-битных



числа и затем складываются. Если появляется переполнение - отбрасывается.

Стоит заметить, что данная функция аналогична сложению в кольце вычетов по модулю 2 в степени n.

```
220 }
221 void add_mod32(uint8_t* A, uint8_t* B, uint8_t* C)
222 {
223     uint8_t inner = 0;
224     for(int i = 3; i >= 0; i--)
225     {
226         inner = A[i] + B[i] + (inner >> 8);
227         C[i] = inner & 0xff;
228     }
229 }
230
```

Рис. 11: Функция *add\_mod32*

*GOST\_28147\_89\_T* - функция, выполняющая нелинейное биективное преобразование. Выполняется на основе таблицы подстановок в *S\_box* следующим образом: исходный 32-битный вектор разбивается на 4-х битные подпоследовательности, номер следования которых определяет строчку в таблице замен, а значение номер индекса в этой ячейке, после чего значение в этой ячейке становится значением 4-х битного вектора. В конце 4-х битные вектора складываются обратно в 32-х битный.

*GOST\_28147\_89\_g* - функция, выполняющая преобразование *g*. Это преобразование включает в себя сложение правой части блока с итерационным ключом по модулю 32, нелинейное биективное преобразование и сдвиг влево на одиннадцать разрядов.

```

272 void GOST 28147 89 T (uint8_t *in data, uint8_t *out data)
273 {
274     uint8_t first section Byte, sec section Byte;
275     for(int i = 0; i < 4; i++)
276     {
277         first section Byte = (in data[i] & 0xf0) >> 4;
278         sec section Byte = (in data[i] & 0x0f);
279         first section Byte = S box[i*2][first section Byte];
280         sec section Byte = S box[i*2 + 1][sec section Byte];
281         out data[i] = (first section Byte << 4) | sec section Byte;
282     }
283 }

```

Рис. 12: Функция *GOST\_28147\_89\_T*

```

297 void GOST 28147 89 g(uint8_t* Key, uint8_t* B, uint8_t* out data)
298 {
299     uint8_t inner[4];
300     uint32_t out data 32 = 0;
301     add mod32(B, Key, inner);
302     GOST 28147 89 T(inner, inner);
303     out data 32 = inner[0];
304     out data 32 = (out data 32 << 8) + inner[1];
305     out data 32 = (out data 32 << 8) + inner[2];
306     out data 32 = (out data 32 << 8) + inner[3];
307     out data 32 = (out data 32 << 11) | (out data 32 >> 21);
308     for(int i = 3; i >= 0; i--) out data[i] = out data 32 >> ((3 - i)*8);
309 }

```

Рис. 13: Функция *GOST\_28147\_89\_g*

*GOST\_28147\_89\_G* - функция, выполняющая преобразование *G*. Данное преобразование представляет собой одну итерацию цикла зашифровывания или расшифровывания (с первой по тридцать первую). Включает в себя преобразование *g*, сложение по модулю 2 результата преобразования *g* с правой половиной блока и обмен содержимым между правой и левой частью блока.

*GOST\_28147\_89\_G\_Finally* - функция, выполняющая финальное преобразование *G*. Это последняя (тридцать вторая) итерация цикла зашифровывания или расшифровывания. От простого преобразования *G* отличается отсутствием

```

311 void GOST 28147 89 G(uint8_t* Key, uint8_t* A, uint8_t* out data)
312 {
313     uint8_t a 0[4];
314     uint8_t a 1[4];
315     uint8_t G[4];
316     for(int i = 0; i < 4; i++)
317     {
318         a 0[i] = A[i + 4];
319         a 1[i] = A[i];
320     }
321     GOST 28147 89 g(Key, a 0, G);
322     addition mod2(a 1, G, G);
323     for(int i = 0; i < 4; i++)
324     {
325         a 1[i] = a 0[i];
326         a 0[i] = G[i];
327     }
328     for(int i = 0; i < 4; i++)
329     {
330         out data[i] = a 1[i];
331         out data[4 + i] = a 0[i];
332     }
333 }

```

Рис. 14: Функция *GOST\_28147\_89\_G*

обмена значениями между правой и левой частью исходного блока.

*Produce\_IV* - функция, отвечающая за синхропысылку(*IV*).

*GOST\_28147\_89\_encrypt* - функция, отвечающая за шифрование. Шифрование производится путем тридцати двух итераций, с первой по тридцать первую с применением преобразования *G* и тридцать вторую с применением *G\_Finally*.

*GOST\_28147\_89\_decrypt* - функция, отвечающая за расшифрование.

```

335 void GOST 28147 89 G Finally(uint8_t* Key, uint8_t* A, uint8_t *out data)
336 {
337     uint8_t a 0[4];
338     uint8_t a 1[4];
339     uint8_t G[4];
340     for(int i = 0; i < 4; i++)
341     {
342         a 0[i] = A[i + 4];
343         a 1[i] = A[i];
344     }
345     GOST 28147 89 g(Key, a 0, G);
346     addition mod2(a 1, G, G);
347     for(int i = 0; i < 4; i++) a 1[i] = G[i];
348     for(int i = 0; i < 4; i++)
349     {
350         out data[i] = a 1[i];
351         out data[4 + i] = a 0[i];
352     }
353 }
354

```

Рис. 15: Функция *GOST\_28147\_89\_G\_Finally*

```

262 void Produce IV(uint8_t* IV)
263 {
264     random device rd;
265     uniform int distribution<uint32_t> distr;
266     for(int i = 0; i < 8; i++)
267     {
268         IV[i] = distr(rd);
269     }
270 }
271

```

Рис. 16: Функция IV

```

354
355 void GOST 28147 89 encrypt(uint8_t *blk, uint8_t *OUTPUT block)
356 {
357     GOST 28147 89 G(Key Iter[0], blk, OUTPUT block);
358     for(int i = 1; i < 31; i++) GOST 28147 89 G(Key Iter[i], OUTPUT block, OUTPUT block);
359     GOST 28147 89 G Finally(Key Iter[31], OUTPUT block, OUTPUT block);
360 }
361

```

Рис. 17: Функция шифрования

Расшифровыва- ние выполняется аналогично зашифровыванию с использова- нием итераци- онных ключей в обратном порядке.

```

361
362 void GOST 28147 89 decrypt(uint8_t *blk, uint8_t *OUTPUT block)
363 {
364     GOST 28147 89 G(Key Iter[31], blk, OUTPUT block);
365     for(int i = 30; i > 0; i--) GOST 28147 89 G(Key Iter[i], OUTPUT block, OUTPUT block);
366     GOST 28147 89 G Finally(Key Iter[0], OUTPUT block, OUTPUT block);
367 }
368

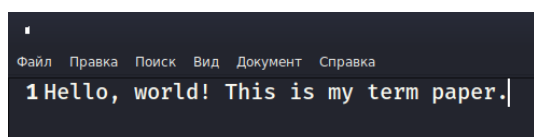
```

Рис. 18: Функция расшифрования

## 0.2.1 Демонстрация работы программы

Продemonстрируем работу программы.

Содержание файла *test.txt* :



```

Файл  Правка  Поиск  Вид  Документ  Справка
1Hello, world! This is my term paper.|

```

Рис. 19: Соержание файла test.txt

После того, как я ввел файл, который необходимо зашифровать с созданием ключа, то в директории появились файлы *key.txt* - созданный ключ, *ENCRYPTED.txt* - зашифрованный текст.

Содержание файла *ENCRYPTED.txt*:

Затем попробуем расшифровать файл *ENCRYPTED.txt*, введя имя файла, содержащий ключ:

Видим, что появился файл *test\_2.txt*, откроем его:

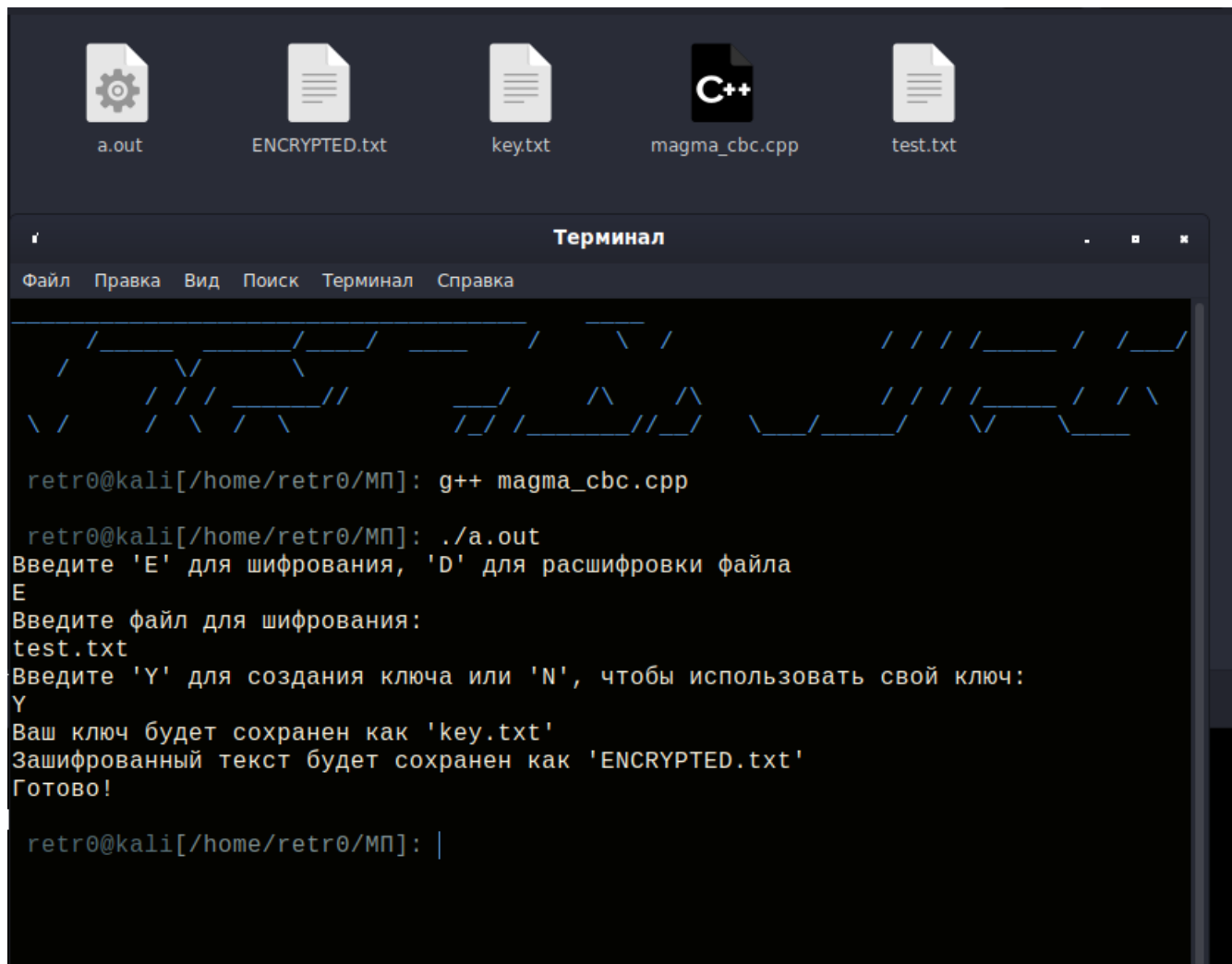


Рис. 20: В директории появились файла key.txt и ENCRYPTED.txt

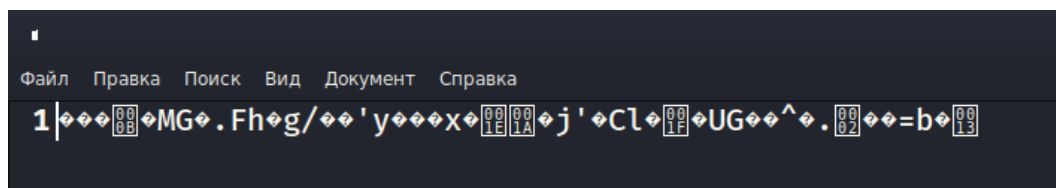


Рис. 21: Содержание файла ENCRYPTED.txt

## 0.2.2 Тест скорости работы программы

Проверим скорость работы алгоритма на примере шифрования 2 файлов размером *5MB* и *163MB*

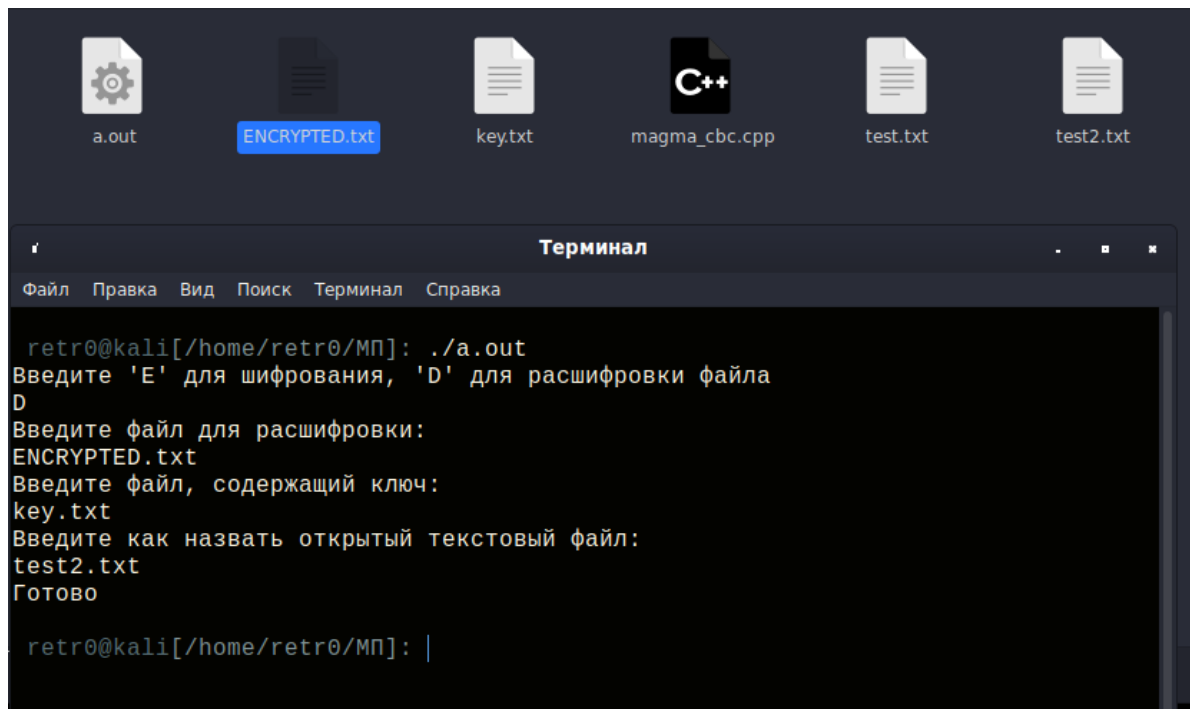


Рис. 22: Расшифрование ENCRYPTED.txt

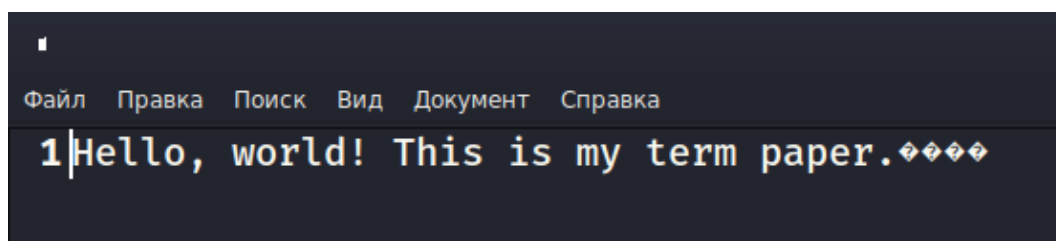


Рис. 23: Результат расшифрования файла ENCRYPTED.txt

Шифрование файла размером  $1MB$ :

Шифрование файла, размером  $100MB$ :

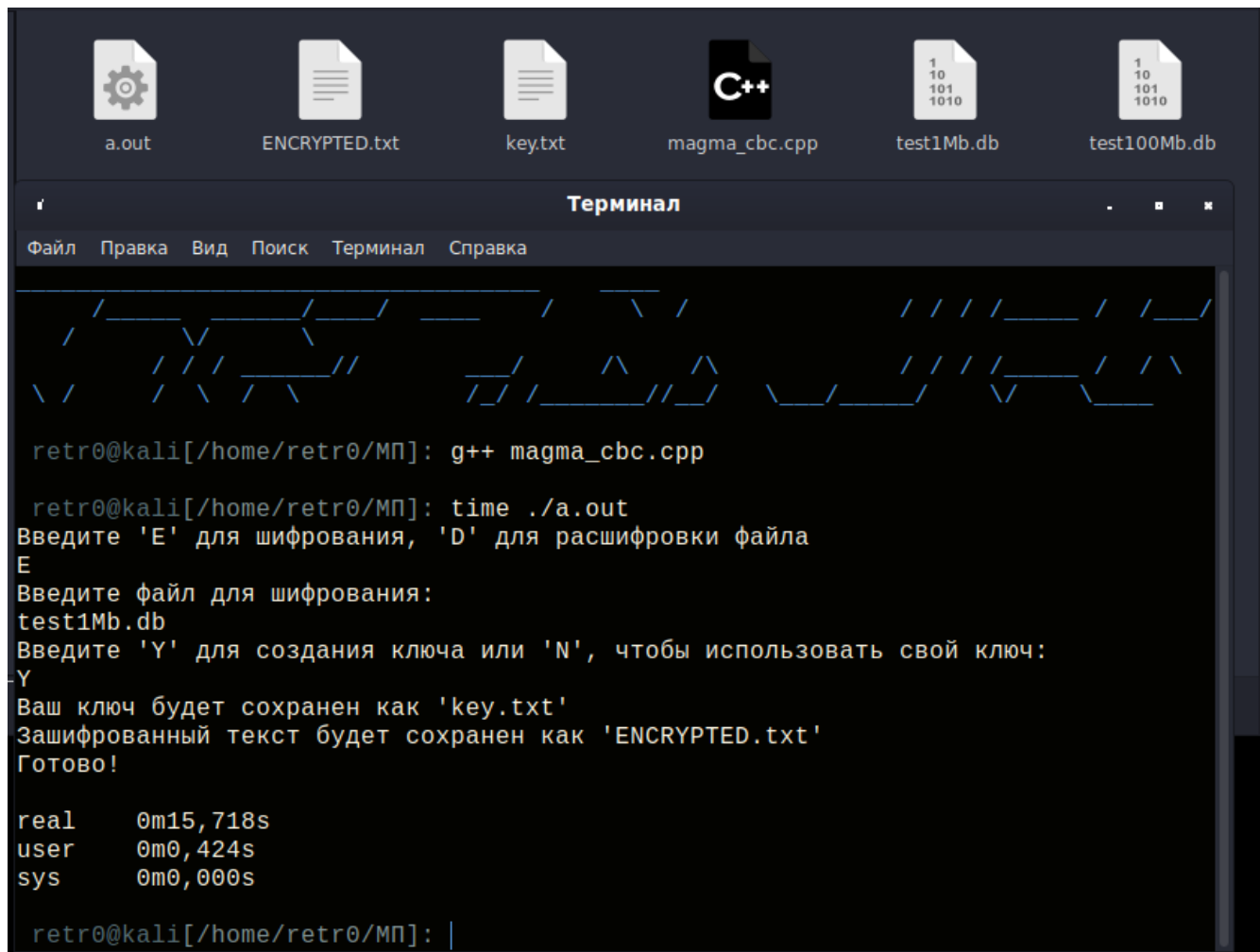


Рис. 24: Шифрование файла размером 1MB

## Заключение

В данной работе я реализовал алгоритм шифрования ГОСТ 28147–89, а также провел исследования скорости шифрования реализованного мной алгоритма. Шифр является устойчивым к атакам путем полного перебора.



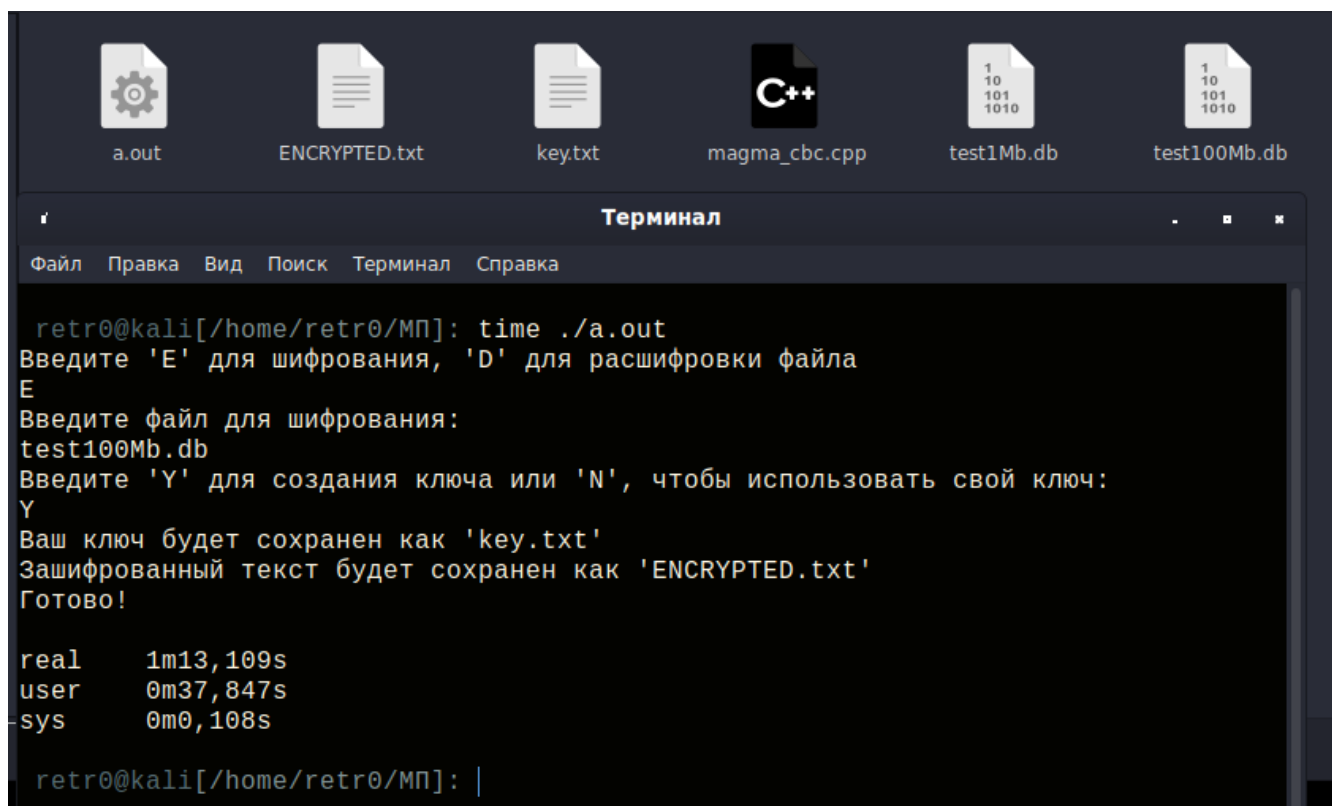


Рис. 25: Шифрование файла размером 100MB

# Литература

- [1] Шнайер Б. – Прикладная криптография. Протоколы, алгоритмы и исходные тексты на языке С – М.:Издательский дом «Вильямс» – 2016. – 1040 с.
- [2] А. Б. Лось, А. Ю. Нестеренко, М. И. Рожков. – Криптографические методы защиты информации: учебник для академического бакалавриата – 2-е изд., испр. – М. : Издательство Юрайт, 2018. – 473 с.
- [3] Панасенко С.П. – Алгоритмы шифрования. Специальный справочник. – СПб.: БХВ-Петербург, 2009. – 506 с.: ил.
- [4] Popov, V., Kurepkin, I., and S. Leontiev. Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms (англ.) // RFC 4357. — IETF, January 2006.
- [5] Романец Ю.В., Панасенко С.П., Заботин И.А., Петров С.В., Ракитин В.В., Дударев Д.А., Сырчин В.К., Салманова Ш.А. Глава 3. История создания алгоритма ГОСТ 28147-89 и принципы, заложенные в его основу // Фирма «АНКАД» – 25 лет на службе обеспечения информационной безопасности России (рус.) / под ред. Романца Ю.В.. — М.: Техносфера, 2016. — С. 9—19. — 256 с. — ISBN 978-5-94836-429-2.
- [6] Саломаа А. Криптография с открытым ключом. - М., 1995.

- [7] Винокуров А. Алгоритм шифрования ГОСТ 28147-89, его использование и реализация для компьютеров платформы Intel x86.
- [8] ГОСТ 28147-89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования.
- [9] Файстель Хорст. Криптография и компьютерная безопасность. Перевод А.Винокурова по изданию Horst Feistel. Cryptography and Computer Privacy, Scientific American, May 1973, Vol. 228, No. 5, pp. 15-23.
- [10] Информационная технология. Криптографическая защита информации. Функция хэширования ГОСТ Р34.11-94, Госстандарт РФ, М., 1994.

## 0.3 Приложение

### Исходный код:

```
#include <iostream>
#include <random>
#include <ctime>
#include <cmath>
#include <fstream>
#include <string>

using namespace std;

static unsigned char S_box[8][16]=
{
    {9,6,3,2,8,11,1,7,10,4,14,15,12,0,13,5},
    {3,7,14,9,8,10,15,0,5,2,6,12,11,4,13,1},
    {14,4,6,2,11,3,13,8,12,15,5,10,0,7,1,9},
    {14,7,10,12,13,1,3,9,0,2,11,4,15,8,5,6},
    {11,5,1,9,8,13,15,0,14,4,2,3,12,7,10,6},
    {3,10,13,12,1,2,0,11,7,5,9,4,8,15,14,6},
    {1,13,2,9,7,10,6,0,8,12,4,5,15,3,11,14},
    {11,10,15,5,0,12,14,8,6,2,3,9,1,7,13,4}
};

uint32_t File_Size(char*);
uint32_t Blocks(uint32_t);

static uint8_t Key[32];
static uint8_t Key_Iter[32][3];

void Add_mod2(uint8_t*, uint8_t*, uint8_t*);
void addition_mod2(uint8_t*, uint8_t*, uint8_t*);
void add_mod32(uint8_t*, uint8_t*, uint8_t*);

void Produce_key();
void Produce_IV(uint8_t*);

void GOST_28147_89_T(uint8_t*, uint8_t*);
void GOST_28147_89_Extend_Key(uint8_t [32][3]);
void GOST_28147_89_g(uint8_t*, uint8_t*, uint8_t*);
void GOST_28147_89_G(uint8_t*, uint8_t*, uint8_t*);
void GOST_28147_89_G_Finally(uint8_t*, uint8_t*, uint8_t*);
void GOST_28147_89_encrypt(uint8_t *, uint8_t*);
void GOST_28147_89_decrypt(uint8_t*, uint8_t*);
```

```

int main(int argc, char* argv[])
{
    char INPUT_DATA_1[32], INPUT_DATA_2[32], E, K, T = 'N';
    uint8_t Buff[8], Enc[8], IV[8];
    cout<<"Введите 'E' для шифрования, 'D' для расшифровки файла"<<endl;
    cin>>E;
    if(E == 'E')
    {
        cout<<"Введите файл для шифрования: "<<endl;
        cin>>INPUT_DATA_1;
        ifstream fs(INPUT_DATA_1, ios_base::binary);
        while(!fs.is_open())
        {
            cout<<"Файл не существует. Пожалуйста, попробуйте еще раз: "<<endl;
            cin>>INPUT_DATA_1;
            fs.open(INPUT_DATA_1, ios_base::binary);
        }
        cout<<"Введите 'Y' для создания ключа или 'N', чтобы использовать свой ключ: "<<endl;
        cin>>K;
        if(K == 'Y')
        {
            cout<<"Ваш ключ будет сохранен как 'key.txt'"<<endl;
            Produce_key();
        }
        else if(K == 'N')
        {
            cout<<"Введите свой ключ файл: "<<endl;
            cin>>INPUT_DATA_2;
            ifstream Key_file(INPUT_DATA_2, ios_base::binary);
            while(!Key_file.is_open())
            {
                cout<<"Что-то не так с вашим ключевым файлом. Пожалуйста, попробуйте еще раз: "<<endl;
                cin>>INPUT_DATA_2;
                Key_file.open(INPUT_DATA_2, ios_base::binary);
            }
            Key_file.read((char*)Key, 32);
            Key_file.close();
        }
        GOST_28147_89_Extend_Key(Key_Iter);
        Produce_IV(IV);

        cout<<"Зашифрованный текст будет сохранен как 'ENCRYPTED.txt'"<<endl;
        ofstream OUT("ENCRYPTED.txt", ios_base::binary);

        for(int i = 0; i < 8; i++)
        {
            OUT<<IV[i];
        }
    }
}

```

```

uint8_t Buff[8];
for(int i = 0; i < 8; i++)
{
    Buff[i] = 0;
}

uint32_t size = File_Size(INPUT_DATA_1);
uint32_t Blocks_32 = Blocks(size);
int cursor = 0, temp = 8;
for(int k = 0; k < Blocks_32; k++)
{
    if((cursor + temp) <= size)
    {
        fs.read((char*)Buff, temp);
        Add_mod2(IV, Buff, IV);
        GOST_28147_89_encrypt(IV, IV);
        for(int i = 0; i < 8; i++) OUT<<IV[i];
        cursor += temp;
    }
    else
    {
        temp = size - cursor;
        if(temp > 0)
        {
            for(int i = 0; i < 8; i++) Buff[i] = 0;
            fs.read((char*)Buff, temp);
            Add_mod2(IV, Buff, IV);
            GOST_28147_89_encrypt(IV, IV);
            for(int i = 0; i < 8; i++) OUT<<IV[i];
        }
    }
}
fs.close();
OUT.close();
cout<<"Готово!"<<endl;
}
else if(E == 'D')
{
    cout<<"Введите_файл_для_расшифровки:_"<<endl;
    cin>>INPUT_DATA_1;
    ifstream CT(INPUT_DATA_1, ios_base::binary);

    while(!CT.is_open())
    {
        cout<<"Файл_не_найден. _Пожалуйста, _попробуйте_еще_раз:_"<<endl;
        cin>>INPUT_DATA_1;
        CT.open(INPUT_DATA_1, ios_base::binary);
    }
}

```

```

cout<<"Введите_файл,_содержащий_ключ: "<<endl;
cin>>INPUT_DATA_2;
ifstream Key_file(INPUT_DATA_2, ios_base::binary);

while(!Key_file.is_open())
{
    cout<<"Что-то_не_так_с_файлом,_содержащим_ключ._Пожалуйста,_попробуйте_еще_раз: "<<endl;
    cin>>INPUT_DATA_2;
    Key_file.open(INPUT_DATA_2, ios_base::binary);
}

Key_file.read((char*)Key, 32);
Key_file.close();
GOST_28147_89_Extend_Key(Key_Iter);
CT.read((char*)IV, 8);

cout<<"Введите_как_назвать_открытый_текстовый_файл: "<<endl;
cin>>INPUT_DATA_2;
uint8_t Gamma[8];
long size = File_Size(INPUT_DATA_1) - 8;
long Blocks_32 = Blocks(size);

ofstream Open_text(INPUT_DATA_2, ios_base::binary);
long cursor = 0, temp = 8;
for(int i = 0; i < 8; i++)
{
    Enc[i] = 0;
}
for(int k = 0; k < Blocks_32; k++)
{
    if((cursor + temp) <= size)
    {
        CT.read((char*)Enc, temp);
        for(int i = 0; i < 8; i++)
        {
            Gamma[i] = Enc[i];
        }
        GOST_28147_89_decrypt(Enc, Enc);
        Add_mod2(IV, Enc, Enc);
        for(int i = 0; i < 8; i++)
        {
            Open_text<<Enc[i];
            IV[i] = Gamma[i];
        }
        cursor += temp;
    }
    else

```

```

        {
            temp = size - cursor;
            if (temp > 0)
            {
                for (int i = 0; i < 8; i++) Enc[i] = 0;
                CT.read((char*)Enc, temp);
                for (int i = 0; i < 8; i++)
                {
                    Gamma[i] = Enc[i];
                }
                GOST_28147_89_decrypt(Enc, Enc);
                Add_mod2(IV, Enc, Enc);
                for (int i = 0; i < 8; i++) Open_text<<Enc[i];
            }
        }
    }
    CT.close();
    Open_text.close();
    cout<<"Готово"<<endl;
}

return 0;
}

void Add_mod2(uint8_t* A, uint8_t* B, uint8_t* C)
{
    for (int i = 0; i < 8; i++) C[i] = A[i]^B[i];
}

void addition_mod2(uint8_t* A, uint8_t* B, uint8_t* C)
{
    for (int i = 0; i < 4; i++) C[i] = A[i]^B[i];
}

void add_mod32(uint8_t* A, uint8_t* B, uint8_t* C)
{
    uint8_t inner = 0;
    for (int i = 3; i >= 0; i--)
    {
        inner = A[i] + B[i] + (inner >> 8);
        C[i] = inner & 0xff;
    }
}

uint32_t File_Size(char* file)
{
    ifstream stream(file, ios_base::binary);
    uint32_t cursor, length;
    cursor = stream.tellg();
    stream.seekg(0, ios_base::end);

```



```

        length = stream.tellg();
        stream.seekg(0,ios_base::beg);
        stream.close();
        return length;
}

uint32_t Blocks(uint32_t size)
{
    if(size % 8 == 0) return size/8;
    else return size/8 + 1;
}

void Produce_key()
{
    ofstream out;
    out.open("key.txt");
    random_device rd;
    uniform_int_distribution<uint32_t> distr;
    for(int i = 0; i < 32; i++)
    {
        Key[i] = distr(rd);
        out<<Key[i];
    }
    out.close();
}

void Produce_IV(uint8_t* IV)
{
    random_device rd;
    uniform_int_distribution<uint32_t> distr;
    for(int i = 0; i < 8; i++)
    {
        IV[i] = distr(rd);
    }
}

void GOST_28147_89_T (uint8_t *in_data, uint8_t *out_data)
{
    uint8_t first_section_Byte, sec_section_Byte;
    for(int i = 0; i < 4; i++)
    {
        first_section_Byte = (in_data[i] & 0xf0) >> 4;
        sec_section_Byte = (in_data[i] & 0x0f);
        first_section_Byte = S_box[i*2][first_section_Byte];
        sec_section_Byte = S_box[i*2 + 1][sec_section_Byte];
        out_data[i] = (first_section_Byte << 4) | sec_section_Byte;
    }
}

```

```

void GOST_28147_89_Extend_Key(uint8_t Key_Iter[32][3])
{
    for(int i = 0; i < 24; i++)
    {
        for(int j = 0; j < 4; j++) Key_Iter[i][j] = Key[(i%8)*4+j];
    }
    for(int i = 24; i < 32; i++)
    {
        for(int j = 0; j < 4; j++) Key_Iter[i][j] = Key[28-(i%8)*4+j];
    }
}

void GOST_28147_89_g(uint8_t* Key, uint8_t* B, uint8_t* out_data)
{
    uint8_t inner[4];
    uint32_t out_data_32 = 0;
    add_mod32(B, Key, inner);
    GOST_28147_89_T(inner, inner);
    out_data_32 = inner[0];
    out_data_32 = (out_data_32 << 8) + inner[1];
    out_data_32 = (out_data_32 << 8) + inner[2];
    out_data_32 = (out_data_32 << 8) + inner[3];
    out_data_32 = (out_data_32 << 11) | (out_data_32 >> 21);
    for(int i = 3; i >= 0; i--) out_data[i] = out_data_32 >> ((3 - i)*8);
}

void GOST_28147_89_G(uint8_t* Key, uint8_t* A, uint8_t* out_data)
{
    uint8_t a_0[4];
    uint8_t a_1[4];
    uint8_t G[4];
    for(int i = 0; i < 4; i++)
    {
        a_0[i] = A[i + 4];
        a_1[i] = A[i];
    }
    GOST_28147_89_g(Key, a_0, G);
    addition_mod2(a_1, G, G);
    for(int i = 0; i < 4; i++)
    {
        a_1[i] = a_0[i];
        a_0[i] = G[i];
    }
    for(int i = 0; i < 4; i++)
    {
        out_data[i] = a_1[i];
        out_data[4 + i] = a_0[i];
    }
}

```

```

}

void GOST_28147_89_G_Finally(uint8_t* Key, uint8_t* A, uint8_t *out_data)
{
    uint8_t a_0[4];
    uint8_t a_1[4];
    uint8_t G[4];
    for(int i = 0; i < 4; i++)
    {
        a_0[i] = A[i + 4];
        a_1[i] = A[i];
    }
    GOST_28147_89_g(Key, a_0, G);
    addition_mod2(a_1, G, G);
    for(int i = 0; i < 4; i++) a_1[i] = G[i];
    for(int i = 0; i < 4; i++)
    {
        out_data[i] = a_1[i];
        out_data[4 + i] = a_0[i];
    }
}

void GOST_28147_89_encrypt(uint8_t *blk, uint8_t *OUTPUT_block)
{
    GOST_28147_89_G(Key_Iter[0], blk, OUTPUT_block);
    for(int i = 1; i < 31; i++) GOST_28147_89_G(Key_Iter[i], OUTPUT_block, OUTPUT_block);
    GOST_28147_89_G_Finally(Key_Iter[31], OUTPUT_block, OUTPUT_block);
}

void GOST_28147_89_decrypt(uint8_t *blk, uint8_t *OUTPUT_block)
{
    GOST_28147_89_G(Key_Iter[31], blk, OUTPUT_block);
    for(int i = 30; i > 0; i--) GOST_28147_89_G(Key_Iter[i], OUTPUT_block, OUTPUT_block);
    GOST_28147_89_G_Finally(Key_Iter[0], OUTPUT_block, OUTPUT_block);
}

```